Network File System (NFS) Version 4 Protocol
draft-ietf-nfsv4-rfc3530bis-27.txt

Abstract

   The Network File System (NFS) version 4 is a distributed file system
   protocol which builds on the heritage of NFS protocol version 2, RFC
   1094, and version 3, RFC 1813. Unlike earlier versions, the NFS
   version 4 protocol supports traditional file access while integrating
   support for file locking and the mount protocol. In addition,
   support for strong security (and its negotiation), compound
   operations, client caching, and internationalization have been added.
   Of course, attention has been applied to making NFS version 4 operate
   well in an Internet environment.

   This document, together with the companion XDR description document,
   RFCNFSv4XDR, obsoletes RFC 3530 as the definition of the NFS version
   4 protocol.

Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [RFC2119].

Copyright Notice

Table of Contents

1. Introduction

1.1. NFS Version 4 Goals

   The Network Filesystem version 4 (NFSv4) protocol is a further
   revision of the NFS protocol defined already by versions 2 [RFC1094]
   and 3 [RFC1813]. It retains the essential characteristics of
   previous versions: design for easy recovery, independent of transport
   protocols, operating systems and file systems, simplicity, and good
   performance. The NFSv4 revision has the following goals:

   o Improved access and good performance on the Internet.

      The protocol is designed to transit firewalls easily, perform well
      where latency is high and bandwidth is low, and scale to very
      large numbers of clients per server.

   o Strong security with negotiation built into the protocol.

      The protocol builds on the work of the Open Network Computing
      (ONC) Remote Procedure Call (RPC) working group in supporting the
      RPCSEC_GSS protocol (see both [RFC2203] and [RFC5403]).
      Additionally, the NFS version 4 protocol provides a mechanism to
      allow clients and servers the ability to negotiate security and
      require clients and servers to support a minimal set of security
      schemes.

   o Good cross-platform interoperability.

      The protocol features a file system model that provides a useful,
      common set of features that does not unduly favor one file system
      or operating system over another.

   o Designed for protocol extensions.

      The protocol is designed to accept standard extensions that do not
      compromise backward compatibility.

   This document, together with the companion XDR description document
   [I-D.ietf-nfsv4-rfc3530bis-dot-x], obsoletes RFC 3530 [RFC3530] as
   the authoritative document describing NFSv4. It does not introduce
   any over-the-wire protocol changes, in the sense that previously
   valid requests remain valid.

1.2. Definitions in the companion document NFS Version 4 Protocol are Authoritative

   [I-D.ietf-nfsv4-rfc3530bis-dot-x], NFS Version 4 Protocol, contains

the definitions in XDR description language of the constructs used by
the protocol. Inside this document, several of the constructs are
reproduced for purposes of explanation. The reader is warned of the
possibility of errors in the reproduced constructs outside of
[I-D.ietf-nfsv4-rfc3530bis-dot-x]. For any part of the document that
is inconsistent with [I-D.ietf-nfsv4-rfc3530bis-dot-x],
[I-D.ietf-nfsv4-rfc3530bis-dot-x] is to be considered authoritative.

1.3. Overview of NFSv4 Features

To provide a reasonable context for the reader, the major features of
NFSv4 protocol will be reviewed in brief. This will be done to
provide an appropriate context for both the reader who is familiar
with the previous versions of the NFS protocol and the reader who is
new to the NFS protocols. For the reader new to the NFS protocols,
some fundamental knowledge is still expected. The reader should be
familiar with the XDR and RPC protocols as described in [RFC5531] and
[RFC4506]. A basic knowledge of file systems and distributed file
systems is expected as well.

1.3.1. RPC and Security

As with previous versions of NFS, the External Data Representation
(XDR) and RPC mechanisms used for the NFSv4 protocol are those
defined in [RFC5531] and [RFC4506]. To meet end to end security
requirements, the RPCSEC_GSS framework (both version 1 in [RFC2203]
and version 2 in [RFC5403]) will be used to extend the basic RPC
security. With the use of RPCSEC_GSS, various mechanisms can be
provided to offer authentication, integrity, and privacy to the NFS
version 4 protocol. Kerberos V5 will be used as described in
[RFC4121] to provide one security framework. With the use of
RPCSEC_GSS, other mechanisms may also be specified and used for NFS
version 4 security.

To enable in-band security negotiation, the NFSv4 protocol has added
a new operation which provides the client with a method of querying
the server about its policies regarding which security mechanisms
must be used for access to the server's file system resources. With
this, the client can securely match the security mechanism that meets
the policies specified at both the client and server.

1.3.2. Procedure and Operation Structure

A significant departure from the previous versions of the NFS
protocol is the introduction of the COMPOUND procedure. For the
NFSv4 protocol, there are two RPC procedures, NULL and COMPOUND. The
COMPOUND procedure is defined in terms of operations and these
operations correspond more closely to the traditional NFS procedures.

With the use of the COMPOUND procedure, the client is able to build
simple or complex requests. These COMPOUND requests allow for a
reduction in the number of RPCs needed for logical file system
operations. For example, without previous contact with a server a
client will be able to read data from a file in one request by
combining LOOKUP, OPEN, and READ operations in a single COMPOUND RPC.
With previous versions of the NFS protocol, this type of single
request was not possible.

The model used for COMPOUND is very simple. There is no logical OR
or ANDing of operations. The operations combined within a COMPOUND
request are evaluated in order by the server. Once an operation
returns a failing result, the evaluation ends and the results of all
evaluated operations are returned to the client.

The NFSv4 protocol continues to have the client refer to a file or
directory at the server by a "filehandle". The COMPOUND procedure
has a method of passing a filehandle from one operation to another
within the sequence of operations. There is a concept of a "current
filehandle" and "saved filehandle". Most operations use the "current
filehandle" as the file system object to operate upon. The "saved
filehandle" is used as temporary filehandle storage within a COMPOUND
procedure as well as an additional operand for certain operations.

1.3.3. Filesystem Model

The general file system model used for the NFSv4 protocol is the same
as previous versions. The server file system is hierarchical with
the regular files contained within being treated as opaque byte
streams. In a slight departure, file and directory names are encoded
with UTF-8 to deal with the basics of internationalization.

The NFSv4 protocol does not require a separate protocol to provide
for the initial mapping between path name and filehandle. Instead of
using the older MOUNT protocol for this mapping, the server provides
a ROOT filehandle that represents the logical root or top of the file
system tree provided by the server. The server provides multiple
file systems by gluing them together with pseudo file systems. These
pseudo file systems provide for potential gaps in the path names
between real file systems.

1.3.3.1. Filehandle Types

In previous versions of the NFS protocol, the filehandle provided by
the server was guaranteed to be valid or persistent for the lifetime
of the file system object to which it referred. For some server
implementations, this persistence requirement has been difficult to
meet. For the NFSv4 protocol, this requirement has been relaxed by

introducing another type of filehandle, volatile. With persistent
and volatile filehandle types, the server implementation can match
the abilities of the file system at the server along with the
operating environment. The client will have knowledge of the type of
filehandle being provided by the server and can be prepared to deal
with the semantics of each.

1.3.3.2. Attribute Types

The NFSv4 protocol has a rich and extensible file object attribute
structure, which is divided into REQUIRED, RECOMMENDED, and named
attributes (see Section 5).

Several (but not all) of the REQUIRED attributes are derived from the
attributes of NFSv3 (see definition of the fattr3 data type in
[RFC1813]). An example of a REQUIRED attribute is the file object's
type (Section 5.8.1.2) so that regular files can be distinguished
from directories (also known as folders in some operating
environments) and other types of objects. REQUIRED attributes are
discussed in Section 5.1.

An example of the RECOMMENDED attributes is an acl (Section 6.2.1).
This attribute defines an Access Control List (ACL) on a file object.
An ACL provides file access control beyond the model used in NFSv3.
The ACL definition allows for specification of specific sets of
permissions for individual users and groups. In addition, ACL
inheritance allows propagation of access permissions and restriction
down a directory tree as file system objects are created.
RECOMMENDED attributes are discussed in Section 5.2.

A named attribute is an opaque byte stream that is associated with a
directory or file and referred to by a string name. Named attributes
are meant to be used by client applications as a method to associate
application-specific data with a regular file or directory. NFSv4.1
modifies named attributes relative to NFSv4.0 by tightening the
allowed operations in order to prevent the development of non-
interoperable implementations. Named attributes are discussed in
Section 5.3.

1.3.3.3. Multi-server Namespace

NFSv4 contains a number of features to allow implementation of
namespaces that cross server boundaries and that allow and facilitate
a non-disruptive transfer of support for individual file systems
between servers. They are all based upon attributes that allow one
file system to specify alternate or new locations for that file
system.

These attributes may be used together with the concept of absent file
systems, which provide specifications for additional locations but no
actual file system content. This allows a number of important
facilities:

o Location attributes may be used with absent file systems to
   implement referrals whereby one server may direct the client to a
   file system provided by another server. This allows extensive
   multi-server namespaces to be constructed.

o Location attributes may be provided for present file systems to
   provide the locations of alternate file system instances or
   replicas to be used in the event that the current file system
   instance becomes unavailable.

o Location attributes may be provided when a previously present file
   system becomes absent. This allows non-disruptive migration of
   file systems to alternate servers.

1.3.4. OPEN and CLOSE

The NFSv4 protocol introduces OPEN and CLOSE operations. The OPEN
operation provides a single point where file lookup, creation, and
share semantics can be combined. The CLOSE operation also provides
for the release of state accumulated by OPEN.

1.3.5. File Locking

With the NFSv4 protocol, the support for byte range file locking is
part of the NFS protocol. The file locking support is structured so
that an RPC callback mechanism is not required. This is a departure
from the previous versions of the NFS file locking protocol, Network
Lock Manager (NLM). The state associated with file locks is
maintained at the server under a lease-based model. The server
defines a single lease period for all state held by a NFS client. If
the client does not renew its lease within the defined period, all
state associated with the client's lease may be released by the
server. The client may renew its lease with use of the RENEW
operation or implicitly by use of other operations (primarily READ).

1.3.6. Client Caching and Delegation

The file, attribute, and directory caching for the NFSv4 protocol is
similar to previous versions. Attributes and directory information
are cached for a duration determined by the client. At the end of a
predefined timeout, the client will query the server to see if the
related file system object has been updated.

For file data, the client checks its cache validity when the file is
opened. A query is sent to the server to determine if the file has
been changed. Based on this information, the client determines if
the data cache for the file should kept or released. Also, when the
file is closed, any modified data is written to the server.

If an application wants to serialize access to file data, file
locking of the file data ranges in question should be used.

The major addition to NFSv4 in the area of caching is the ability of
the server to delegate certain responsibilities to the client. When
the server grants a delegation for a file to a client, the client is
guaranteed certain semantics with respect to the sharing of that file
with other clients. At OPEN, the server may provide the client
either a OPEN_DELEGATE_READ or OPEN_DELEGATE_WRITE delegation for the
file. If the client is granted a OPEN_DELEGATE_READ delegation, it
is assured that no other client has the ability to write to the file
for the duration of the delegation. If the client is granted a
OPEN_DELEGATE_WRITE delegation, the client is assured that no other
client has read or write access to the file.

Delegations can be recalled by the server. If another client
requests access to the file in such a way that the access conflicts
with the granted delegation, the server is able to notify the initial
client and recall the delegation. This requires that a callback path
exist between the server and client. If this callback path does not
exist, then delegations cannot be granted. The essence of a
delegation is that it allows the client to locally service operations
such as OPEN, CLOSE, LOCK, LOCKU, READ, or WRITE without immediate
interaction with the server.

1.4. General Definitions

The following definitions are provided for the purpose of providing
an appropriate context for the reader.

Absent File System: A file system is "absent" when a namespace
   component does not have a backing file system.

Byte: In this document, a byte is an octet, i.e., a datum exactly 8
   bits in length.

Client: The client is the entity that accesses the NFS server's
   resources. The client may be an application that contains the
   logic to access the NFS server directly. The client may also be
   the traditional operating system client that provides remote file
   system services for a set of applications.

With reference to byte-range locking, the client is also the
entity that maintains a set of locks on behalf of one or more
applications. This client is responsible for crash or failure
recovery for those locks it manages.

Note that multiple clients may share the same transport and
connection and multiple clients may exist on the same network
node.

Client ID: A 64-bit quantity used as a unique, short-hand reference
   to a client supplied Verifier and ID. The server is responsible
   for supplying the Client ID.

File System: The file system is the collection of objects on a
   server that share the same fsid attribute (see Section 5.8.1.9).

Lease: An interval of time defined by the server for which the
   client is irrevocably granted a lock. At the end of a lease
   period the lock may be revoked if the lease has not been extended.
   The lock must be revoked if a conflicting lock has been granted
   after the lease interval.

   All leases granted by a server have the same fixed interval. Note
   that the fixed interval was chosen to alleviate the expense a
   server would have in maintaining state about variable length
   leases across server failures.

Lock: The term "lock" is used to refer to both record (byte-range)
   locks as well as share reservations unless specifically stated
   otherwise.

Server: The "Server" is the entity responsible for coordinating
   client access to a set of file systems.

Stable Storage: NFSv4 servers must be able to recover without data
   loss from multiple power failures (including cascading power
   failures, that is, several power failures in quick succession),
   operating system failures, and hardware failure of components
   other than the storage medium itself (for example, disk,
   nonvolatile RAM).

   Some examples of stable storage that are allowable for an NFS
   server include:

   (1) Media commit of data, that is, the modified data has been
        successfully written to the disk media, for example, the disk
        platter.

   (2) An immediate reply disk drive with battery-backed on-drive
        intermediate storage or uninterruptible power system (UPS).

   (3) Server commit of data with battery-backed intermediate
        storage and recovery software.

   (4) Cache commit with uninterruptible power system (UPS) and
        recovery software.

   Stateid: A stateid is a 128-bit quantity returned by a server that
      uniquely identifies the open and locking states provided by the
      server for a specific open-owner or lock-owner/open-owner pair for
      a specific file and type of lock.

   Verifier: A 64-bit quantity generated by the client that the server
      can use to determine if the client has restarted and lost all
      previous lock state.

1.5. Changes since RFC 3530

   The main changes from RFC 3530 [RFC3530] are:

   o The XDR definition has been moved to a companion document
      [I-D.ietf-nfsv4-rfc3530bis-dot-x]

   o Updates for the latest IETF intellectual property statements

   o There is a restructured and more complete explanation of multi-
      server namespace features.

   o Updating handling of domain names to reflect Internationalized
      Domain Names in Applications (IDNA) [RFC5891].

   o The previously required LIPKEY and SPKM-3 security mechanisms have
      been removed.

   o Some clarification on a client re-establishing callback
      information to the new server if state has been migrated.

   o A third edge case was added for Courtesy locks and network
      partitions.

   o The definition of stateid was strengthened.

1.6. Changes since RFC 3010

   This definition of the NFSv4 protocol replaces or obsoletes the
   definition present in [RFC3010]. While portions of the two documents
   have remained the same, there have been substantive changes in
   others. The changes made between [RFC3010] and this document
   represent implementation experience and further review of the
   protocol. While some modifications were made for ease of
   implementation or clarification, most updates represent errors or
   situations where the [RFC3010] definition were untenable.

   The following list is not all inclusive of all changes but presents
   some of the most notable changes or additions made:

   o The state model has added an open_owner4 identifier. This was
      done to accommodate Posix based clients and the model they use for
      file locking. For Posix clients, an open_owner4 would correspond
      to a file descriptor potentially shared amongst a set of processes
      and the lock_owner4 identifier would correspond to a process that
      is locking a file.

   o Clarifications and error conditions were added for the handling of
      the owner and group attributes. Since these attributes are string
      based (as opposed to the numeric uid/gid of previous versions of
      NFS), translations may not be available and hence the changes
      made.

   o Clarifications for the ACL and mode attributes to address
      evaluation and partial support.

   o For identifiers that are defined as XDR opaque, limits were set on
      their size.

   o Added the mounted_on_fileid attribute to allow Posix clients to
      correctly construct local mounts.

   o Modified the SETCLIENTID/SETCLIENTID_CONFIRM operations to deal
      correctly with confirmation details along with adding the ability
      to specify new client callback information. Also added
      clarification of the callback information itself.

   o Added a new operation RELEASE_LOCKOWNER to enable notifying the
      server that a lock_owner4 will no longer be used by the client.

   o RENEW operation changes to identify the client correctly and allow
      for additional error returns.

   o Verify error return possibilities for all operations.

   o Remove use of the pathname4 data type from LOOKUP and OPEN in
      favor of having the client construct a sequence of LOOKUP
      operations to achieve the same effect.

2. Protocol Data Types

   The syntax and semantics to describe the data types of the NFS
   version 4 protocol are defined in the XDR [RFC4506] and RPC [RFC5531]
   documents. The next sections build upon the XDR data types to define
   types and structures specific to this protocol.

2.1. Basic Data Types

                  These are the base NFSv4 data types.

   +----------------+--+
   | Data Type | Definition |
   +----------------+--+
int32_t	typedef int int32_t;
uint32_t	typedef unsigned int uint32_t;
int64_t	typedef hyper int64_t;
uint64_t	typedef unsigned hyper uint64_t;
attrlist4	typedef opaque attrlist4<>;
	Used for file/directory attributes.
bitmap4	typedef uint32_t bitmap4<>;
	Used in attribute array encoding.
changeid4	typedef uint64_t changeid4;
	Used in the definition of change_info4.
clientid4	typedef uint64_t clientid4;
	Shorthand reference to client identification.
count4	typedef uint32_t count4;
	Various count parameters (READ, WRITE, COMMIT).
length4	typedef uint64_t length4;
	Describes LOCK lengths.
mode4	typedef uint32_t mode4;
	Mode attribute data type.
nfs_cookie4	typedef uint64_t nfs_cookie4;
	Opaque cookie value for READDIR.
nfs_fh4	typedef opaque nfs_fh4<NFS4_FHSIZE>;
	Filehandle definition.
nfs_ftype4	enum nfs_ftype4;
	Various defined file types.
nfsstat4	enum nfsstat4;
	Return value for operations.
offset4	typedef uint64_t offset4;

	Various offset designations (READ, WRITE, LOCK, COMMIT).
qop4	typedef uint32_t qop4; Quality of protection designation in SECINFO.
sec_oid4	typedef opaque sec_oid4<>; Security Object Identifier. The sec_oid4 data type is not really opaque. Instead it contains an ASN.1 OBJECT IDENTIFIER as used by GSS-API in the mech_type argument to GSS_Init_sec_context. See [RFC2743] for details.
seqid4	typedef uint32_t seqid4; Sequence identifier used for file locking.
utf8string	typedef opaque utf8string<>; UTF-8 encoding for strings.
utf8str_cis	typedef utf8string utf8str_cis; Case-insensitive UTF-8 string.
utf8str_cs	typedef utf8string utf8str_cs; Case-sensitive UTF-8 string.
utf8str_mixed	typedef utf8string utf8str_mixed; UTF-8 strings with a case-sensitive prefix and a case-insensitive suffix.
component4	typedef utf8str_cs component4; Represents pathname components.
linktext4	typedef utf8str_cs linktext4; Symbolic link contents ("symbolic link" is defined in an Open Group [openg_symlink] standard).
ascii_REQUIRED4	typedef utf8string ascii_REQUIRED4; String MUST be sent as ASCII and thus is automatically UTF-8.
pathname4	typedef component4 pathname4<>; Represents path name for fs_locations.
nfs_lockid4	typedef uint64_t nfs_lockid4;
verifier4	typedef opaque verifier4[NFS4_VERIFIER_SIZE]; Verifier used for various operations (COMMIT, CREATE, OPEN, READDIR, WRITE) NFS4_VERIFIER_SIZE is defined as 8.

End of Base Data Types

Table 1

[2.2](#). Structured Data Types

[2.2.1](#). nfstime4

```
struct nfstime4 {
        int64_t         seconds;
        uint32_t        nseconds;
};
```

The nfstime4 structure gives the number of seconds and nanoseconds
since midnight or 0 hour January 1, 1970 Coordinated Universal Time
(UTC). Values greater than zero for the seconds field denote dates
after the 0 hour January 1, 1970. Values less than zero for the
seconds field denote dates before the 0 hour January 1, 1970. In
both cases, the nseconds field is to be added to the seconds field
for the final time representation. For example, if the time to be
represented is one-half second before 0 hour January 1, 1970, the
seconds field would have a value of negative one (-1) and the
nseconds fields would have a value of one-half second (500000000).
Values greater than 999,999,999 for nseconds are considered invalid.

This data type is used to pass time and date information. A server
converts to and from its local representation of time when processing
time values, preserving as much accuracy as possible. If the
precision of timestamps stored for a file system object is less than
defined, loss of precision can occur. An adjunct time maintenance
protocol is recommended to reduce client and server time skew.

[2.2.2](#). time_how4

```
enum time_how4 {
        SET_TO_SERVER_TIME4 = 0,
        SET_TO_CLIENT_TIME4 = 1
};
```

[2.2.3](#). settime4

```
union settime4 switch (time_how4 set_it) {
 case SET_TO_CLIENT_TIME4:
        nfstime4        time;
 default:
        void;
};
```

The above definitions are used as the attribute definitions to set
time values. If set_it is SET_TO_SERVER_TIME4, then the server uses
its local representation of time for the time value.

2.2.4. **specdata4**

```
struct specdata4 {
 uint32_t specdata1; /* major device number */
 uint32_t specdata2; /* minor device number */
};
```

This data type represents additional information for the device file
types NF4CHR and NF4BLK.

2.2.5. **fsid4**

```
struct fsid4 {
        uint64_t        major;
        uint64_t        minor;
};
```

This type is the file system identifier that is used as a mandatory
attribute.

2.2.6. **fs_location4**

```
struct fs_location4 {
        utf8str_cis             server<>;
        pathname4               rootpath;
};
```

2.2.7. **fs_locations4**

```
struct fs_locations4 {
        pathname4       fs_root;
        fs_location4    locations<>;
};
```

The fs_location4 and fs_locations4 data types are used for the
fs_locations recommended attribute which is used for migration and
replication support.

2.2.8. **fattr4**

```
struct fattr4 {
        bitmap4         attrmask;
        attrlist4       attr_vals;
};
```

The fattr4 structure is used to represent file and directory
attributes.

The bitmap is a counted array of 32 bit integers used to contain bit
values. The position of the integer in the array that contains bit n
can be computed from the expression (n / 32) and its bit within that
integer is (n mod 32).

```
                      0              1
      +-----------+-----------+-----------+--
      |  count    | 31  ..  0 | 63  .. 32 |
      +-----------+-----------+-----------+--
```

2.2.9. change_info4

```
   struct change_info4 {
           bool            atomic;
           changeid4       before;
           changeid4       after;
   };
```

This structure is used with the CREATE, LINK, REMOVE, RENAME
operations to let the client know the value of the change attribute
for the directory in which the target file system object resides.

2.2.10. clientaddr4

```
   struct clientaddr4 {
           /* see struct rpcb in RFC 1833 */
           string r_netid<>;    /* network id */
           string r_addr<>;     /* universal address */
   };
```

The clientaddr4 structure is used as part of the SETCLIENTID
operation to either specify the address of the client that is using a
client ID or as part of the callback registration. The r_netid and
r_addr fields respectively contain a netid and uaddr. The netid and
uaddr concepts are defined in [RFC5665]. The netid and uaddr formats
for TCP over IPv4 and TCP over IPv6 are defined in [RFC5665],
specifically Tables 2 and 3 and Sections 5.2.3.3 and 5.2.3.4.

2.2.11. cb_client4

```
   struct cb_client4 {
           unsigned int    cb_program;
           clientaddr4     cb_location;
   };
```

This structure is used by the client to inform the server of its call
back address; includes the program number and client address.

2.2.12. nfs_client_id4

```
struct nfs_client_id4 {
        verifier4       verifier;
        opaque          id<NFS4_OPAQUE_LIMIT>;
};
```

   This structure is part of the arguments to the SETCLIENTID operation.

2.2.13. open_owner4

```
struct open_owner4 {
        clientid4       clientid;
        opaque          owner<NFS4_OPAQUE_LIMIT>;
};
```

   This structure is used to identify the owner of open state.

2.2.14. lock_owner4

```
struct lock_owner4 {
        clientid4       clientid;
        opaque          owner<NFS4_OPAQUE_LIMIT>;
};
```

   This structure is used to identify the owner of file locking state.

2.2.15. open_to_lock_owner4

```
struct open_to_lock_owner4 {
        seqid4          open_seqid;
        stateid4        open_stateid;
        seqid4          lock_seqid;
        lock_owner4     lock_owner;
};
```

   This structure is used for the first LOCK operation done for an
   open_owner4. It provides both the open_stateid and lock_owner such
   that the transition is made from a valid open_stateid sequence to
   that of the new lock_stateid sequence. Using this mechanism avoids
   the confirmation of the lock_owner/lock_seqid pair since it is tied
   to established state in the form of the open_stateid/open_seqid.

2.2.16. **stateid4**

```
struct stateid4 {
        uint32_t         seqid;
        opaque           other[NFS4_OTHER_SIZE];
};
```

This structure is used for the various state sharing mechanisms
between the client and server. For the client, this data structure
is read-only. The server is required to increment the seqid field
monotonically at each transition of the stateid. This is important
since the client will inspect the seqid in OPEN stateids to determine
the order of OPEN processing done by the server.

3. **RPC and Security Flavor**

The NFSv4 protocol is a RPC application that uses RPC version 2 and
the XDR as defined in [RFC5531] and [RFC4506]. The RPCSEC_GSS
security flavors as defined in version 1 ([RFC2203]) and version 2
([RFC5403]) MUST be implemented as the mechanism to deliver stronger
security for the NFSv4 protocol. However, deployment of RPCSEC_GSS
is optional.

3.1. **Ports and Transports**

Historically, NFSv2 and NFSv3 servers have resided on port 2049. The
registered port 2049 [RFC3232] for the NFS protocol SHOULD be the
default configuration. Using the registered port for NFS services
means the NFS client will not need to use the RPC binding protocols
as described in [RFC1833]; this will allow NFS to transit firewalls.

Where an NFSv4 implementation supports operation over the IP network
protocol, the supported transport layer between NFS and IP MUST be an
IETF standardised transport protocol that is specified to avoid
network congestion; such transports include TCP and SCTP. To enhance
the possibilities for interoperability, an NFSv4 implementation MUST
support operation over the TCP transport protocol, at least until
such time as a standards track RFC revises this requirement to use a
different IETF standardised transport protocol with appropriate
congestion control.

If TCP is used as the transport, the client and server SHOULD use
persistent connections. This will prevent the weakening of TCP's
congestion control via short lived connections and will improve
performance for the Wide Area Network (WAN) environment by
eliminating the need for SYN handshakes.

To date, all NFSv4 implementations are TCP based, i.e., there are
none for SCTP nor UDP. UDP by itself is not sufficient as a
transport for NFSv4, neither is UDP in combination with some other
mechanism (e.g., DCCP [RFC4340], NORM [RFC5740]).

As noted in Section 17, the authentication model for NFSv4 has moved
from machine-based to principal-based. However, this modification of
the authentication model does not imply a technical requirement to
move the TCP connection management model from whole machine-based to
one based on a per user model. In particular, NFS over TCP client
implementations have traditionally multiplexed traffic for multiple
users over a common TCP connection between an NFS client and server.
This has been true, regardless of whether the NFS client is using
AUTH_SYS, AUTH_DH, RPCSEC_GSS or any other flavor. Similarly, NFS
over TCP server implementations have assumed such a model and thus
scale the implementation of TCP connection management in proportion
to the number of expected client machines. It is intended that NFSv4
will not modify this connection management model. NFSv4 clients that
violate this assumption can expect scaling issues on the server and
hence reduced service.

3.1.1. Client Retransmission Behavior

When processing a NFSv4 request received over a reliable transport
such as TCP, the NFSv4 server MUST NOT silently drop the request,
except if the established transport connection has been broken.
Given such a contract between NFSv4 clients and servers, clients MUST
NOT retry a request unless one or both of the following are true:

o The transport connection has been broken

o The procedure being retried is the NULL procedure

Since reliable transports, such as TCP, do not always synchronously
inform a peer when the other peer has broken the connection (for
example, when an NFS server reboots), the NFSv4 client may want to
actively "probe" the connection to see if has been broken. Use of
the NULL procedure is one recommended way to do so. So, when a
client experiences a remote procedure call timeout (of some arbitrary
implementation specific amount), rather than retrying the remote
procedure call, it could instead issue a NULL procedure call to the
server. If the server has died, the transport connection break will
eventually be indicated to the NFSv4 client. The client can then
reconnect, and then retry the original request. If the NULL
procedure call gets a response, the connection has not broken. The
client can decide to wait longer for the original request's response,
or it can break the transport connection and reconnect before re-
sending the original request.

For callbacks from the server to the client, the same rules apply,
but the server doing the callback becomes the client, and the client
receiving the callback becomes the server.

3.2. Security Flavors

Traditional RPC implementations have included AUTH_NONE, AUTH_SYS,
AUTH_DH, and AUTH_KRB4 as security flavors. With [RFC2203] an
additional security flavor of RPCSEC_GSS has been introduced which
uses the functionality of GSS-API [RFC2743]. This allows for the use
of various security mechanisms by the RPC layer without the
additional implementation overhead of adding RPC security flavors.
For NFSv4, the RPCSEC_GSS security flavor MUST be used to enable the
mandatory security mechanism. Other flavors, such as, AUTH_NONE,
AUTH_SYS, and AUTH_DH MAY be implemented as well.

3.2.1. Security mechanisms for NFSv4

RPCSEC_GSS, via GSS-API, normalizes access to mechanisms that provide
security services. Therefore, NFSv4 clients and servers MUST support
the Kerberos V5 security mechanism.

The use of RPCSEC_GSS requires selection of mechanism, quality of
protection (QOP), and service (authentication, integrity, privacy).
For the mandated security mechanisms, NFSv4 specifies that a QOP of
zero is used, leaving it up to the mechanism or the mechanism's
configuration to map QOP zero to an appropriate level of protection.
Each mandated mechanism specifies a minimum set of cryptographic
algorithms for implementing integrity and privacy. NFSv4 clients and
servers MUST be implemented on operating environments that comply
with the REQUIRED cryptographic algorithms of each REQUIRED
mechanism.

3.2.1.1. Kerberos V5 as a Security Triple

The Kerberos V5 GSS-API mechanism as described in [RFC4121] MUST be
implemented with the RPCSEC_GSS services as specified in the
following table:

```
column descriptions:
1 == number of pseudo flavor
2 == name of pseudo flavor
3 == mechanism's OID
4 == RPCSEC_GSS service
5 == NFSv4 clients MUST support
6 == NFSv4 servers MUST support

1       2       3                               4                    5   6
-------------------------------------------------------------------------
390003 krb5     1.2.840.113554.1.2.2 rpc_gss_svc_none        yes yes
390004 krb5i    1.2.840.113554.1.2.2 rpc_gss_svc_integrity yes yes
390005 krb5p    1.2.840.113554.1.2.2 rpc_gss_svc_privacy     no yes
```

Note that the pseudo flavor is presented here as a mapping aid to the
implementor. Because this NFS protocol includes a method to
negotiate security and it understands the GSS-API mechanism, the
pseudo flavor is not needed. The pseudo flavor is needed for NFSv3
since the security negotiation is done via the MOUNT protocol as
described in [RFC2623].

At the time this document was specified, the Advanced Encryption
Standard (AES) with HMAC-SHA1 was a REQUIRED algorithm set for
Kerberos V5. In contrast, when NFSv4.0 was first specified in
[RFC3530], weaker algorithm sets were REQUIRED for Kerberos V5, and
were REQUIRED in the NFSv4.0 specification, because the Kerberos V5
specification at the time did not specify stronger algorithms. The
NFSv4 specification does not specify REQUIRED algorithms for Kerberos
V5, and instead, the implementor is expected to track the evolution
of the Kerberos V5 standard if and when stronger algorithms are
specified.

3.2.1.1.1. Security Considerations for Cryptographic Algorithms in Kerberos V5

When deploying NFSv4, the strength of the security achieved depends
on the existing Kerberos V5 infrastructure. The algorithms of
Kerberos V5 are not directly exposed to or selectable by the client
or server, so there is some due diligence required by the user of
NFSv4 to ensure that security is acceptable where needed. Guidance
is provided in [RFC6649] as to why weak algorithms should be disabled
by default.

3.3. Security Negotiation

With the NFSv4 server potentially offering multiple security
mechanisms, the client needs a method to determine or negotiate which
mechanism is to be used for its communication with the server. The

NFS server may have multiple points within its file system name space
that are available for use by NFS clients. In turn the NFS server
may be configured such that each of these entry points may have
different or multiple security mechanisms in use.

The security negotiation between client and server SHOULD be done
with a secure channel to eliminate the possibility of a third party
intercepting the negotiation sequence and forcing the client and
server to choose a lower level of security than required or desired.
See Section 17 for further discussion.

3.3.1. SECINFO

The new SECINFO operation will allow the client to determine, on a
per filehandle basis, what security triple (see [RFC2743]) is to be
used for server access. In general, the client will not have to use
the SECINFO operation except during initial communication with the
server or when the client crosses policy boundaries at the server.
It is possible that the server's policies change during the client's
interaction therefore forcing the client to negotiate a new security
triple.

3.3.2. Security Error

Based on the assumption that each NFSv4 client and server MUST
support a minimum set of security (i.e., Kerberos-V5 under
RPCSEC_GSS), the NFS client will start its communication with the
server with one of the minimal security triples. During
communication with the server, the client may receive an NFS error of
NFS4ERR_WRONGSEC. This error allows the server to notify the client
that the security triple currently being used is not appropriate for
access to the server's file system resources. The client is then
responsible for determining what security triples are available at
the server and choose one which is appropriate for the client. See
Section 15.33 for further discussion of how the client will respond
to the NFS4ERR_WRONGSEC error and use SECINFO.

3.3.3. Callback RPC Authentication

Except as noted elsewhere in this section, the callback RPC
(described later) MUST mutually authenticate the NFS server to the
principal that acquired the client ID (also described later), using
the security flavor the original SETCLIENTID operation used.

For AUTH_NONE, there are no principals, so this is a non-issue.

AUTH_SYS has no notions of mutual authentication or a server
principal, so the callback from the server simply uses the AUTH_SYS

credential that the user used when he set up the delegation.

For AUTH_DH, one commonly used convention is that the server uses the credential corresponding to this AUTH_DH principal:

    unix.host@domain

where host and domain are variables corresponding to the name of server host and directory services domain in which it lives such as a Network Information System domain or a DNS domain.

Regardless of what security mechanism under RPCSEC_GSS is being used, the NFS server MUST identify itself in GSS-API via a GSS_C_NT_HOSTBASED_SERVICE name type. GSS_C_NT_HOSTBASED_SERVICE names are of the form:

    service@hostname

For NFS, the "service" element is

    nfs

Implementations of security mechanisms will convert nfs@hostname to various different forms. For Kerberos V5, the following form is RECOMMENDED:

nfs/hostname

For Kerberos V5, nfs/hostname would be a server principal in the Kerberos Key Distribution Center database. This is the same principal the client acquired a GSS-API context for when it issued the SETCLIENTID operation, therefore, the realm name for the server principal must be the same for the callback as it was for the SETCLIENTID.

[4](). Filehandles

The filehandle in the NFS protocol is a per server unique identifier for a file system object. The contents of the filehandle are opaque to the client. Therefore, the server is responsible for translating the filehandle to an internal representation of the file system object.

[4.1](). Obtaining the First Filehandle

The operations of the NFS protocol are defined in terms of one or more filehandles. Therefore, the client needs a filehandle to

initiate communication with the server. With the NFSv2 protocol
[RFC1094] and the NFSv3 protocol [RFC1813], there exists an ancillary
protocol to obtain this first filehandle. The MOUNT protocol, RPC
program number 100005, provides the mechanism of translating a string
based file system path name to a filehandle which can then be used by
the NFS protocols.

The MOUNT protocol has deficiencies in the area of security and use
via firewalls. This is one reason that the use of the public
filehandle was introduced in [RFC2054] and [RFC2055]. With the use
of the public filehandle in combination with the LOOKUP operation in
the NFSv2 and NFSv3 protocols, it has been demonstrated that the
MOUNT protocol is unnecessary for viable interaction between NFS
client and server.

Therefore, the NFSv4 protocol will not use an ancillary protocol for
translation from string based path names to a filehandle. Two
special filehandles will be used as starting points for the NFS
client.

4.1.1. Root Filehandle

The first of the special filehandles is the ROOT filehandle. The
ROOT filehandle is the "conceptual" root of the file system name
space at the NFS server. The client uses or starts with the ROOT
filehandle by employing the PUTROOTFH operation. The PUTROOTFH
operation instructs the server to set the "current" filehandle to the
ROOT of the server's file tree. Once this PUTROOTFH operation is
used, the client can then traverse the entirety of the server's file
tree with the LOOKUP operation. A complete discussion of the server
name space is in Section 7.

4.1.2. Public Filehandle

The second special filehandle is the PUBLIC filehandle. Unlike the
ROOT filehandle, the PUBLIC filehandle may be bound or represent an
arbitrary file system object at the server. The server is
responsible for this binding. It may be that the PUBLIC filehandle
and the ROOT filehandle refer to the same file system object.
However, it is up to the administrative software at the server and
the policies of the server administrator to define the binding of the
PUBLIC filehandle and server file system object. The client may not
make any assumptions about this binding. The client uses the PUBLIC
filehandle via the PUTPUBFH operation.

4.2. Filehandle Types

   In the NFSv2 and NFSv3 protocols, there was one type of filehandle
   with a single set of semantics. This type of filehandle is termed
   "persistent" in NFS Version 4. The semantics of a persistent
   filehandle remain the same as before. A new type of filehandle
   introduced in NFS Version 4 is the "volatile" filehandle, which
   attempts to accommodate certain server environments.

   The volatile filehandle type was introduced to address server
   functionality or implementation issues which make correct
   implementation of a persistent filehandle infeasible. Some server
   environments do not provide a file system level invariant that can be
   used to construct a persistent filehandle. The underlying server
   file system may not provide the invariant or the server's file system
   programming interfaces may not provide access to the needed
   invariant. Volatile filehandles may ease the implementation of
   server functionality such as hierarchical storage management or file
   system reorganization or migration. However, the volatile filehandle
   increases the implementation burden for the client.

   Since the client will need to handle persistent and volatile
   filehandles differently, a file attribute is defined which may be
   used by the client to determine the filehandle types being returned
   by the server.

4.2.1. General Properties of a Filehandle

   The filehandle contains all the information the server needs to
   distinguish an individual file. To the client, the filehandle is
   opaque. The client stores filehandles for use in a later request and
   can compare two filehandles from the same server for equality by
   doing a byte-by-byte comparison. However, the client MUST NOT
   otherwise interpret the contents of filehandles. If two filehandles
   from the same server are equal, they MUST refer to the same file
   system object. Servers SHOULD try to maintain a one-to-one
   correspondence between filehandles and file system objects but this
   is not required. Clients MUST use filehandle comparisons only to
   improve performance, not for correct behavior. All clients need to
   be prepared for situations in which it cannot be determined whether
   two filehandles denote the same object and in such cases, avoid
   making invalid assumptions which might cause incorrect behavior.
   Further discussion of filehandle and attribute comparison in the
   context of data caching is presented in Section 10.3.4.

   As an example, in the case that two different path names when
   traversed at the server terminate at the same file system object, the
   server SHOULD return the same filehandle for each path. This can

occur if a hard link is used to create two file names which refer to
the same underlying file object and associated data. For example, if
paths /a/b/c and /a/d/c refer to the same file, the server SHOULD
return the same filehandle for both path names traversals.

4.2.2. Persistent Filehandle

A persistent filehandle is defined as having a fixed value for the
lifetime of the file system object to which it refers. Once the
server creates the filehandle for a file system object, the server
MUST accept the same filehandle for the object for the lifetime of
the object. If the server restarts or reboots the NFS server must
honor the same filehandle value as it did in the server's previous
instantiation. Similarly, if the file system is migrated, the new
NFS server must honor the same filehandle as the old NFS server.

The persistent filehandle will be become stale or invalid when the
file system object is removed. When the server is presented with a
persistent filehandle that refers to a deleted object, it MUST return
an error of NFS4ERR_STALE. A filehandle may become stale when the
file system containing the object is no longer available. The file
system may become unavailable if it exists on removable media and the
media is no longer available at the server or the file system in
whole has been destroyed or the file system has simply been removed
from the server's name space (i.e., unmounted in a UNIX environment).

4.2.3. Volatile Filehandle

A volatile filehandle does not share the same longevity
characteristics of a persistent filehandle. The server may determine
that a volatile filehandle is no longer valid at many different
points in time. If the server can definitively determine that a
volatile filehandle refers to an object that has been removed, the
server should return NFS4ERR_STALE to the client (as is the case for
persistent filehandles). In all other cases where the server
determines that a volatile filehandle can no longer be used, it
should return an error of NFS4ERR_FHEXPIRED.

The mandatory attribute "fh_expire_type" is used by the client to
determine what type of filehandle the server is providing for a
particular file system. This attribute is a bitmask with the
following values:

FH4_PERSISTENT: The value of FH4_PERSISTENT is used to indicate a
   persistent filehandle, which is valid until the object is removed
   from the file system. The server will not return
   NFS4ERR_FHEXPIRED for this filehandle. FH4_PERSISTENT is defined
   as a value in which none of the bits specified below are set.

FH4_VOLATILE_ANY: The filehandle may expire at any time, except as
   specifically excluded (i.e., FH4_NOEXPIRE_WITH_OPEN).

FH4_NOEXPIRE_WITH_OPEN: May only be set when FH4_VOLATILE_ANY is
   set. If this bit is set, then the meaning of FH4_VOLATILE_ANY is
   qualified to exclude any expiration of the filehandle when it is
   open.

FH4_VOL_MIGRATION: The filehandle will expire as a result of
   migration. If FH4_VOLATILE_ANY is set, FH4_VOL_MIGRATION is
   redundant.

FH4_VOL_RENAME: The filehandle will expire during rename. This
   includes a rename by the requesting client or a rename by any
   other client. If FH4_VOLATILE_ANY is set, FH4_VOL_RENAME is
   redundant.

Servers which provide volatile filehandles that may expire while open
(i.e., if FH4_VOL_MIGRATION or FH4_VOL_RENAME is set or if
FH4_VOLATILE_ANY is set and FH4_NOEXPIRE_WITH_OPEN not set), should
deny a RENAME or REMOVE that would affect an OPEN file of any of the
components leading to the OPEN file. In addition, the server should
deny all RENAME or REMOVE requests during the grace period upon
server restart.

Note that the bits FH4_VOL_MIGRATION and FH4_VOL_RENAME allow the
client to determine that expiration has occurred whenever a specific
event occurs, without an explicit filehandle expiration error from
the server. FH4_VOLATILE_ANY does not provide this form of
information. In situations where the server will expire many, but
not all filehandles upon migration (e.g., all but those that are
open), FH4_VOLATILE_ANY (in this case with FH4_NOEXPIRE_WITH_OPEN) is
a better choice since the client may not assume that all filehandles
will expire when migration occurs, and it is likely that additional
expirations will occur (as a result of file CLOSE) that are separated
in time from the migration event itself.

4.2.4. One Method of Constructing a Volatile Filehandle

A volatile filehandle, while opaque to the client could contain:

   [volatile bit = 1 | server boot time | slot | generation number]

o slot is an index in the server volatile filehandle table

o generation number is the generation number for the table entry/
   slot

When the client presents a volatile filehandle, the server makes the
following checks, which assume that the check for the volatile bit
has passed. If the server boot time is less than the current server
boot time, return NFS4ERR_FHEXPIRED. If slot is out of range, return
NFS4ERR_BADHANDLE. If the generation number does not match, return
NFS4ERR_FHEXPIRED.

When the server reboots, the table is gone (it is volatile).

If volatile bit is 0, then it is a persistent filehandle with a
different structure following it.

4.3. Client Recovery from Filehandle Expiration

If possible, the client should recover from the receipt of an
NFS4ERR_FHEXPIRED error. The client must take on additional
responsibility so that it may prepare itself to recover from the
expiration of a volatile filehandle. If the server returns
persistent filehandles, the client does not need these additional
steps.

For volatile filehandles, most commonly the client will need to store
the component names leading up to and including the file system
object in question. With these names, the client should be able to
recover by finding a filehandle in the name space that is still
available or by starting at the root of the server's file system name
space.

If the expired filehandle refers to an object that has been removed
from the file system, obviously the client will not be able to
recover from the expired filehandle.

It is also possible that the expired filehandle refers to a file that
has been renamed. If the file was renamed by another client, again
it is possible that the original client will not be able to recover.
However, in the case that the client itself is renaming the file and
the file is open, it is possible that the client may be able to
recover. The client can determine the new path name based on the
processing of the rename request. The client can then regenerate the
new filehandle based on the new path name. The client could also use
the compound operation mechanism to construct a set of operations
like:

    RENAME A B
    LOOKUP B
    GETFH

Note that the COMPOUND procedure does not provide atomicity. This

example only reduces the overhead of recovering from an expired
filehandle.

5. **Attributes**

To meet the requirements of extensibility and increased
interoperability with non-UNIX platforms, attributes need to be
handled in a flexible manner. The NFSv3 fattr3 structure contains a
fixed list of attributes that not all clients and servers are able to
support or care about. The fattr3 structure cannot be extended as
new needs arise and it provides no way to indicate non-support. With
the NFSv4.0 protocol, the client is able to query what attributes the
server supports and construct requests with only those supported
attributes (or a subset thereof).

To this end, attributes are divided into three groups: REQUIRED,
RECOMMENDED, and named. Both REQUIRED and RECOMMENDED attributes are
supported in the NFSv4.0 protocol by a specific and well-defined
encoding and are identified by number. They are requested by setting
a bit in the bit vector sent in the GETATTR request; the server
response includes a bit vector to list what attributes were returned
in the response. New REQUIRED or RECOMMENDED attributes may be added
to the NFSv4 protocol as part of a new minor version by publishing a
Standards Track RFC which allocates a new attribute number value and
defines the encoding for the attribute. See Section 11 for further
discussion.

Named attributes are accessed by the OPENATTR operation, which
accesses a hidden directory of attributes associated with a file
system object. OPENATTR takes a filehandle for the object and
returns the filehandle for the attribute hierarchy. The filehandle
for the named attributes is a directory object accessible by LOOKUP
or READDIR and contains files whose names represent the named
attributes and whose data bytes are the value of the attribute. For
example:

```
      +----------+-----------+-------------------------------+
      | LOOKUP   | "foo"     | ; look up file                |
      | GETATTR  | attrbits  |                               |
      | OPENATTR |           | ; access foo's named attributes |
      | LOOKUP   | "x11icon" | ; look up specific attribute  |
      | READ     | 0,4096    | ; read stream of bytes        |
      +----------+-----------+-------------------------------+
```

Named attributes are intended for data needed by applications rather
than by an NFS client implementation. NFS implementors are strongly
encouraged to define their new attributes as RECOMMENDED attributes

by bringing them to the IETF Standards Track process.

The set of attributes that are classified as REQUIRED is deliberately
small since servers need to do whatever it takes to support them. A
server should support as many of the RECOMMENDED attributes as
possible but, by their definition, the server is not required to
support all of them. Attributes are deemed REQUIRED if the data is
both needed by a large number of clients and is not otherwise
reasonably computable by the client when support is not provided on
the server.

Note that the hidden directory returned by OPENATTR is a convenience
for protocol processing. The client should not make any assumptions
about the server's implementation of named attributes and whether or
not the underlying file system at the server has a named attribute
directory. Therefore, operations such as SETATTR and GETATTR on the
named attribute directory are undefined.

5.1. REQUIRED Attributes

These MUST be supported by every NFSv4.0 client and server in order
to ensure a minimum level of interoperability. The server MUST store
and return these attributes, and the client MUST be able to function
with an attribute set limited to these attributes. With just the
REQUIRED attributes some client functionality may be impaired or
limited in some ways. A client may ask for any of these attributes
to be returned by setting a bit in the GETATTR request, and the
server MUST return their value.

5.2. RECOMMENDED Attributes

These attributes are understood well enough to warrant support in the
NFSv4.0 protocol. However, they may not be supported on all clients
and servers. A client MAY ask for any of these attributes to be
returned by setting a bit in the GETATTR request but must handle the
case where the server does not return them. A client MAY ask for the
set of attributes the server supports and SHOULD NOT request
attributes the server does not support. A server should be tolerant
of requests for unsupported attributes and simply not return them
rather than considering the request an error. It is expected that
servers will support all attributes they comfortably can and only
fail to support attributes that are difficult to support in their
operating environments. A server should provide attributes whenever
they don't have to "tell lies" to the client. For example, a file
modification time should be either an accurate time or should not be
supported by the server. At times this will be difficult for
clients, but a client is better positioned to decide whether and how
to fabricate or construct an attribute or whether to do without the

attribute.

5.3. Named Attributes

These attributes are not supported by direct encoding in the NFSv4
protocol but are accessed by string names rather than numbers and
correspond to an uninterpreted stream of bytes that are stored with
the file system object. The name space for these attributes may be
accessed by using the OPENATTR operation. The OPENATTR operation
returns a filehandle for a virtual "named attribute directory", and
further perusal and modification of the name space may be done using
operations that work on more typical directories. In particular,
READDIR may be used to get a list of such named attributes, and
LOOKUP and OPEN may select a particular attribute. Creation of a new
named attribute may be the result of an OPEN specifying file
creation.

Once an OPEN is done, named attributes may be examined and changed by
normal READ and WRITE operations using the filehandles and stateids
returned by OPEN.

Named attributes and the named attribute directory may have their own
(non-named) attributes. Each of these objects must have all of the
REQUIRED attributes and may have additional RECOMMENDED attributes.
However, the set of attributes for named attributes and the named
attribute directory need not be, and typically will not be, as large
as that for other objects in that file system.

Named attributes might be the target of delegations. However, since
granting of delegations is at the server's discretion, a server need
not support delegations on named attributes.

It is RECOMMENDED that servers support arbitrary named attributes. A
client should not depend on the ability to store any named attributes
in the server's file system. If a server does support named
attributes, a client that is also able to handle them should be able
to copy a file's data and metadata with complete transparency from
one location to another; this would imply that names allowed for
regular directory entries are valid for named attribute names as
well.

In NFSv4.0, the structure of named attribute directories is
restricted in a number of ways, in order to prevent the development
of non-interoperable implementations in which some servers support a
fully general hierarchical directory structure for named attributes
while others support a limited but adequate structure for named
attributes. In such an environment, clients or applications might
come to depend on non-portable extensions. The restrictions are:

o CREATE is not allowed in a named attribute directory. Thus, such
   objects as symbolic links and special files are not allowed to be
   named attributes. Further, directories may not be created in a
   named attribute directory, so no hierarchical structure of named
   attributes for a single object is allowed.

o If OPENATTR is done on a named attribute directory or on a named
   attribute, the server MUST return an error.

o Doing a RENAME of a named attribute to a different named attribute
   directory or to an ordinary (i.e., non-named-attribute) directory
   is not allowed.

o Creating hard links between named attribute directories or between
   named attribute directories and ordinary directories is not
   allowed.

Names of attributes will not be controlled by this document or other
IETF Standards Track documents. See Section 18 for further
discussion.

5.4. Classification of Attributes

Each of the REQUIRED and RECOMMENDED attributes can be classified in
one of three categories: per server (i.e., the value of the attribute
will be the same for all file objects that share the same server),
per file system (i.e., the value of the attribute will be the same
for some or all file objects that share the same fsid attribute
(Section 5.8.1.9) and server owner), or per file system object. Note
that it is possible that some per file system attributes may vary
within the file system. Note that it is possible that some per file
system attributes may vary within the file system, depending on the
value of the "homogeneous" (Section 5.8.2.16) attribute. Note that
the attributes time_access_set and time_modify_set are not listed in
this section because they are write-only attributes corresponding to
time_access and time_modify, and are used in a special instance of
SETATTR.

o The per-server attribute is:

       lease_time

o The per-file system attributes are:

       supported_attrs, fh_expire_type, link_support, symlink_support,
       unique_handles, aclsupport, cansettime, case_insensitive,
       case_preserving, chown_restricted, files_avail, files_free,
       files_total, fs_locations, homogeneous, maxfilesize, maxname,

          maxread, maxwrite, no_trunc, space_avail, space_free,
          space_total, time_delta,

   o   The per-file system object attributes are:

          type, change, size, named_attr, fsid, rdattr_error, filehandle,
          acl, archive, fileid, hidden, maxlink, mimetype, mode,
          numlinks, owner, owner_group, rawdev, space_used, system,
          time_access, time_backup, time_create, time_metadata,
          time_modify, mounted_on_fileid

   For quota_avail_hard, quota_avail_soft, and quota_used, see their
   definitions below for the appropriate classification.

5.5. Set-Only and Get-Only Attributes

   Some REQUIRED and RECOMMENDED attributes are set-only; i.e., they can
   be set via SETATTR but not retrieved via GETATTR. Similarly, some
   REQUIRED and RECOMMENDED attributes are get-only; i.e., they can be
   retrieved via GETATTR but not set via SETATTR. If a client attempts
   to set a get-only attribute or get a set-only attribute, the server
   MUST return NFS4ERR_INVAL.

5.6. REQUIRED Attributes - List and Definition References

   The list of REQUIRED attributes appears in Table 2. The meaning of
   the columns of the table are:

   o   Name: The name of attribute

   o   Id: The number assigned to the attribute. In the event of
       conflicts between the assigned number and
       [I-D.ietf-nfsv4-rfc3530bis-dot-x], the latter is authoritative,
       but in such an event, it should be resolved with Errata to this
       document and/or [I-D.ietf-nfsv4-rfc3530bis-dot-x]. See
       [ISEG errata] for the Errata process.

   o   Data Type: The XDR data type of the attribute.

   o   Acc: Access allowed to the attribute. R means read-only (GETATTR
       may retrieve, SETATTR may not set). W means write-only (SETATTR
       may set, GETATTR may not retrieve). R W means read/write (GETATTR
       may retrieve, SETATTR may set).

   o   Defined in: The section of this specification that describes the
       attribute.

```
+------------------+----+------------+-----+-----------------+
| Name             | Id | Data Type  | Acc | Defined in:     |
+------------------+----+------------+-----+-----------------+
| supported_attrs  | 0  | bitmap4    | R   | Section 5.8.1.1 |
| type             | 1  | nfs_ftype4 | R   | Section 5.8.1.2 |
| fh_expire_type   | 2  | uint32_t   | R   | Section 5.8.1.3 |
| change           | 3  | uint64_t   | R   | Section 5.8.1.4 |
| size             | 4  | uint64_t   | R W | Section 5.8.1.5 |
| link_support     | 5  | bool       | R   | Section 5.8.1.6 |
| symlink_support  | 6  | bool       | R   | Section 5.8.1.7 |
| named_attr       | 7  | bool       | R   | Section 5.8.1.8 |
| fsid             | 8  | fsid4      | R   | Section 5.8.1.9 |
| unique_handles   | 9  | bool       | R   | Section 5.8.1.10 |
| lease_time       | 10 | nfs_lease4 | R   | Section 5.8.1.11 |
| rdattr_error     | 11 | nfsstat4   | R   | Section 5.8.1.12 |
| filehandle       | 19 | nfs_fh4    | R   | Section 5.8.1.13 |
+------------------+----+------------+-----+-----------------+
```

                                Table 2

5.7. RECOMMENDED Attributes - List and Definition References

   The RECOMMENDED attributes are defined in Table 3. The meanings of
   the column headers are the same as Table 2; see Section 5.6 for the
   meanings.

```
+--------------------+----+--------------+-----+------------------+
| Name               | Id | Data Type    | Acc | Defined in:      |
+--------------------+----+--------------+-----+------------------+
| acl                | 12 | nfsace4<>    | R W | Section 6.2.1    |
| aclsupport         | 13 | uint32_t     | R   | Section 6.2.1.2  |
| archive            | 14 | bool         | R W | Section 5.8.2.1  |
| cansettime         | 15 | bool         | R   | Section 5.8.2.2  |
| case_insensitive   | 16 | bool         | R   | Section 5.8.2.3  |
| case_preserving    | 17 | bool         | R   | Section 5.8.2.4  |
| chown_restricted   | 18 | bool         | R   | Section 5.8.2.5  |
| fileid             | 20 | uint64_t     | R   | Section 5.8.2.6  |
| files_avail        | 21 | uint64_t     | R   | Section 5.8.2.7  |
| files_free         | 22 | uint64_t     | R   | Section 5.8.2.8  |
| files_total        | 23 | uint64_t     | R   | Section 5.8.2.9  |
| fs_locations       | 24 | fs_locations | R   | Section 5.8.2.10 |
| hidden             | 25 | bool         | R W | Section 5.8.2.11 |
| homogeneous        | 26 | bool         | R   | Section 5.8.2.12 |
| maxfilesize        | 27 | uint64_t     | R   | Section 5.8.2.13 |
| maxlink            | 28 | uint32_t     | R   | Section 5.8.2.14 |
| maxname            | 29 | uint32_t     | R   | Section 5.8.2.15 |
| maxread            | 30 | uint64_t     | R   | Section 5.8.2.16 |
| maxwrite           | 31 | uint64_t     | R   | Section 5.8.2.17 |
```

mimetype	32	utf8<>	R W	Section 5.8.2.18
mode	33	mode4	R W	Section 6.2.2
mounted_on_fileid	55	uint64_t	R	Section 5.8.2.19
no_trunc	34	bool	R	Section 5.8.2.20
numlinks	35	uint32_t	R	Section 5.8.2.21
owner	36	utf8<>	R W	Section 5.8.2.22
owner_group	37	utf8<>	R W	Section 5.8.2.23
quota_avail_hard	38	uint64_t	R	Section 5.8.2.24
quota_avail_soft	39	uint64_t	R	Section 5.8.2.25
quota_used	40	uint64_t	R	Section 5.8.2.26
rawdev	41	specdata4	R	Section 5.8.2.27
space_avail	42	uint64_t	R	Section 5.8.2.28
space_free	43	uint64_t	R	Section 5.8.2.29
space_total	44	uint64_t	R	Section 5.8.2.30
space_used	45	uint64_t	R	Section 5.8.2.31
system	46	bool	R W	Section 5.8.2.32
time_access	47	nfstime4	R	Section 5.8.2.33
time_access_set	48	settime4	W	Section 5.8.2.34
time_backup	49	nfstime4	R W	Section 5.8.2.35
time_create	50	nfstime4	R W	Section 5.8.2.36
time_delta	51	nfstime4	R	Section 5.8.2.37
time_metadata	52	nfstime4	R	Section 5.8.2.38
time_modify	53	nfstime4	R	Section 5.8.2.39
time_modify_set	54	settime4	W	Section 5.8.2.40

                              Table 3

5.8. Attribute Definitions

5.8.1. Definitions of REQUIRED Attributes

5.8.1.1. Attribute 0: supported_attrs

   The bit vector that would retrieve all REQUIRED and RECOMMENDED
   attributes that are supported for this object. The scope of this
   attribute applies to all objects with a matching fsid.

5.8.1.2. Attribute 1: type

   Designates the type of an object in terms of one of a number of
   special constants:

   o NF4REG designates a regular file.

   o NF4DIR designates a directory.

   o NF4BLK designates a block device special file.

   o NF4CHR designates a character device special file.

   o NF4LNK designates a symbolic link.

   o NF4SOCK designates a named socket special file.

   o NF4FIFO designates a fifo special file.

   o NF4ATTRDIR designates a named attribute directory.

   o NF4NAMEDATTR designates a named attribute.

   Within the explanatory text and operation descriptions, the following
   phrases will be used with the meanings given below:

   o The phrase "is a directory" means that the object's type attribute
      is NF4DIR or NF4ATTRDIR.

   o The phrase "is a special file" means that the object's type
      attribute is NF4BLK, NF4CHR, NF4SOCK, or NF4FIFO.

   o The phrase "is an regular file" means that the object's type
      attribute is NF4REG or NF4NAMEDATTR.

5.8.1.3. Attribute 2: fh_expire_type

   Server uses this to specify filehandle expiration behavior to the
   client. See Section 4 for additional description.

5.8.1.4. Attribute 3: change

   A value created by the server that the client can use to determine if
   file data, directory contents, or attributes of the object have been
   modified. The server MAY return the object's time_metadata attribute
   for this attribute's value but only if the file system object cannot
   be updated more frequently than the resolution of time_metadata.

5.8.1.5. Attribute 4: size

   The size of the object in bytes.

5.8.1.6. Attribute 5: link_support

   TRUE, if the object's file system supports hard links.

5.8.1.7. Attribute 6: symlink_support

   TRUE, if the object's file system supports symbolic links.

5.8.1.8. Attribute 7: named_attr

   TRUE, if this object has named attributes. In other words, object
   has a non-empty named attribute directory.

5.8.1.9. Attribute 8: fsid

   Unique file system identifier for the file system holding this
   object. The fsid attribute has major and minor components, each of
   which are of data type uint64_t.

5.8.1.10. Attribute 9: unique_handles

   TRUE, if two distinct filehandles are guaranteed to refer to two
   different file system objects.

5.8.1.11. Attribute 10: lease_time

   Duration of the lease at server in seconds.

5.8.1.12. Attribute 11: rdattr_error

   Error returned from an attempt to retrieve attributes during a
   READDIR operation.

5.8.1.13. Attribute 19: filehandle

   The filehandle of this object (primarily for READDIR requests).

5.8.2. Definitions of Uncategorized RECOMMENDED Attributes

   The definitions of most of the RECOMMENDED attributes follow.
   Collections that share a common category are defined in other
   sections.

5.8.2.1. Attribute 14: archive

   TRUE, if this file has been archived since the time of last
   modification (deprecated in favor of time_backup).

5.8.2.2. Attribute 15: cansettime

   TRUE, if the server is able to change the times for a file system
   object as specified in a SETATTR operation.

[5.8.2.3](#). Attribute 16: case_insensitive

TRUE, if file name comparisons on this file system are case
insensitive.

[5.8.2.4](#). Attribute 17: case_preserving

TRUE, if file name case on this file system is preserved.

[5.8.2.5](#). Attribute 18: chown_restricted

If TRUE, the server will reject any request to change either the
owner or the group associated with a file if the caller is not a
privileged user (for example, "root" in UNIX operating environments
or in Windows 2000, the "Take Ownership" privilege).

[5.8.2.6](#). Attribute 20: fileid

A number uniquely identifying the file within the file system.

[5.8.2.7](#). Attribute 21: files_avail

File slots available to this user on the file system containing this
object -- this should be the smallest relevant limit.

[5.8.2.8](#). Attribute 22: files_free

Free file slots on the file system containing this object - this
should be the smallest relevant limit.

[5.8.2.9](#). Attribute 23: files_total

Total file slots on the file system containing this object.

[5.8.2.10](#). Attribute 24: fs_locations

Locations where this file system may be found. If the server returns
NFS4ERR_MOVED as an error, this attribute MUST be supported.

The server specifies the root path for a given server by returning a
path consisting of zero path components.

[5.8.2.11](#). Attribute 25: hidden

TRUE, if the file is considered hidden with respect to the Windows
API.

5.8.2.12. **Attribute 26: homogeneous**

   TRUE, if this object's file system is homogeneous, i.e., all objects
   in the file system (all objects on the server with the same fsid)
   have common values for all per-file-system attributes.

5.8.2.13. **Attribute 27: maxfilesize**

   Maximum supported file size for the file system of this object.

5.8.2.14. **Attribute 28: maxlink**

   Maximum number of links for this object.

5.8.2.15. **Attribute 29: maxname**

   Maximum file name size supported for this object.

5.8.2.16. **Attribute 30: maxread**

   Maximum amount of data the READ operation will return for this
   object.

5.8.2.17. **Attribute 31: maxwrite**

   Maximum amount of data the WRITE operation will accept for this
   object. This attribute SHOULD be supported if the file is writable.
   Lack of this attribute can lead to the client either wasting
   bandwidth or not receiving the best performance.

5.8.2.18. **Attribute 32: mimetype**

   MIME body type/subtype of this object.

5.8.2.19. **Attribute 55: mounted_on_fileid**

   Like fileid, but if the target filehandle is the root of a file
   system, this attribute represents the fileid of the underlying
   directory.

   UNIX-based operating environments connect a file system into the
   namespace by connecting (mounting) the file system onto the existing
   file object (the mount point, usually a directory) of an existing
   file system. When the mount point's parent directory is read via an
   API like readdir(), the return results are directory entries, each
   with a component name and a fileid. The fileid of the mount point's
   directory entry will be different from the fileid that the stat()
   system call returns. The stat() system call is returning the fileid

of the root of the mounted file system, whereas readdir() is
returning the fileid that stat() would have returned before any file
systems were mounted on the mount point.

Unlike NFSv3, NFSv4.0 allows a client's LOOKUP request to cross other
file systems. The client detects the file system crossing whenever
the filehandle argument of LOOKUP has an fsid attribute different
from that of the filehandle returned by LOOKUP. A UNIX-based client
will consider this a "mount point crossing". UNIX has a legacy
scheme for allowing a process to determine its current working
directory. This relies on readdir() of a mount point's parent and
stat() of the mount point returning fileids as previously described.
The mounted_on_fileid attribute corresponds to the fileid that
readdir() would have returned as described previously.

While the NFSv4.0 client could simply fabricate a fileid
corresponding to what mounted_on_fileid provides (and if the server
does not support mounted_on_fileid, the client has no choice), there
is a risk that the client will generate a fileid that conflicts with
one that is already assigned to another object in the file system.
Instead, if the server can provide the mounted_on_fileid, the
potential for client operational problems in this area is eliminated.

If the server detects that there is no mounted point at the target
file object, then the value for mounted_on_fileid that it returns is
the same as that of the fileid attribute.

The mounted_on_fileid attribute is RECOMMENDED, so the server SHOULD
provide it if possible, and for a UNIX-based server, this is
straightforward. Usually, mounted_on_fileid will be requested during
a READDIR operation, in which case it is trivial (at least for UNIX-
based servers) to return mounted_on_fileid since it is equal to the
fileid of a directory entry returned by readdir(). If
mounted_on_fileid is requested in a GETATTR operation, the server
should obey an invariant that has it returning a value that is equal
to the file object's entry in the object's parent directory, i.e.,
what readdir() would have returned. Some operating environments
allow a series of two or more file systems to be mounted onto a
single mount point. In this case, for the server to obey the
aforementioned invariant, it will need to find the base mount point,
and not the intermediate mount points.

5.8.2.20. **Attribute 34: no_trunc**

If this attribute is TRUE, then if the client uses a file name longer
than name_max, an error will be returned instead of the name being
truncated.

[5.8.2.21](#). **Attribute 35: numlinks**

   Number of hard links to this object.

[5.8.2.22](#). **Attribute 36: owner**

   The string name of the owner of this object.

[5.8.2.23](#). **Attribute 37: owner_group**

   The string name of the group ownership of this object.

[5.8.2.24](#). **Attribute 38: quota_avail_hard**

   The value in bytes that represents the amount of additional disk
   space beyond the current allocation that can be allocated to this
   file or directory before further allocations will be refused. It is
   understood that this space may be consumed by allocations to other
   files or directories.

[5.8.2.25](#). **Attribute 39: quota_avail_soft**

   The value in bytes that represents the amount of additional disk
   space that can be allocated to this file or directory before the user
   may reasonably be warned. It is understood that this space may be
   consumed by allocations to other files or directories though there
   may exist server side rules as to which other files or directories.

[5.8.2.26](#). **Attribute 40: quota_used**

   The value in bytes that represents the amount of disk space used by
   this file or directory and possibly a number of other similar files
   or directories, where the set of "similar" meets at least the
   criterion that allocating space to any file or directory in the set
   will reduce the "quota_avail_hard" of every other file or directory
   in the set.

   Note that there may be a number of distinct but overlapping sets of
   files or directories for which a quota_used value is maintained,
   e.g., "all files with a given owner", "all files with a given group
   owner", etc. The server is at liberty to choose any of those sets
   when providing the content of the quota_used attribute, but should do
   so in a repeatable way. The rule may be configured per file system
   or may be "choose the set with the smallest quota".

5.8.2.27. **Attribute 41: rawdev**

   Raw device number of file of type NF4BLK or NF4CHR. The device
   number is split into major and minor numbers. If the file's type
   attribute is not NF4BLK or NF4CHR, the value returned SHOULD NOT be
   considered useful.

5.8.2.28. **Attribute 42: space_avail**

   Disk space in bytes available to this user on the file system
   containing this object -- this should be the smallest relevant limit.

5.8.2.29. **Attribute 43: space_free**

   Free disk space in bytes on the file system containing this object --
   this should be the smallest relevant limit.

5.8.2.30. **Attribute 44: space_total**

   Total disk space in bytes on the file system containing this object.

5.8.2.31. **Attribute 45: space_used**

   Number of file system bytes allocated to this object.

5.8.2.32. **Attribute 46: system**

   This attribute is TRUE if this file is a "system" file with respect
   to the Windows operating environment.

5.8.2.33. **Attribute 47: time_access**

   The time_access attribute represents the time of last access to the
   object by a READ operation sent to the server. The notion of what is
   an "access" depends on the server's operating environment and/or the
   server's file system semantics. For example, for servers obeying
   Portable Operating System Interface (POSIX) semantics, time_access
   would be updated only by the READ and READDIR operations and not any
   of the operations that modify the content of the object [16], [17],
   [read_api], [readdir_api], [write_api]. Of course, setting the
   corresponding time_access_set attribute is another way to modify the
   time_access attribute.

   Whenever the file object resides on a writable file system, the
   server should make its best efforts to record time_access into stable
   storage. However, to mitigate the performance effects of doing so,
   and most especially whenever the server is satisfying the read of the
   object's content from its cache, the server MAY cache access time

   updates and lazily write them to stable storage. It is also
   acceptable to give administrators of the server the option to disable
   time_access updates.

5.8.2.34. Attribute 48: time_access_set

   Sets the time of last access to the object. SETATTR use only.

5.8.2.35. Attribute 49: time_backup

   The time of last backup of the object.

5.8.2.36. Attribute 50: time_create

   The time of creation of the object. This attribute does not have any
   relation to the traditional UNIX file attribute "ctime" or "change
   time".

5.8.2.37. Attribute 51: time_delta

   Smallest useful server time granularity.

5.8.2.38. Attribute 52: time_metadata

   The time of last metadata modification of the object.

5.8.2.39. Attribute 53: time_modify

   The time of last modification to the object.

5.8.2.40. Attribute 54: time_modify_set

   Sets the time of last modification to the object. SETATTR use only.

5.9. Interpreting owner and owner_group

   The RECOMMENDED attributes "owner" and "owner_group" (and also users
   and groups within the "acl" attribute) are represented in terms of a
   UTF-8 string. To avoid a representation that is tied to a particular
   underlying implementation at the client or server, the use of the
   UTF-8 string has been chosen. Note that section 6.1 of RFC 2624
   [RFC2624] provides additional rationale. It is expected that the
   client and server will have their own local representation of owner
   and owner_group that is used for local storage or presentation to the
   end user. Therefore, it is expected that when these attributes are
   transferred between the client and server, the local representation
   is translated to a syntax of the form "user@dns_domain". This will
   allow for a client and server that do not use the same local

representation the ability to translate to a common syntax that can
be interpreted by both.

Similarly, security principals may be represented in different ways
by different security mechanisms. Servers normally translate these
representations into a common format, generally that used by local
storage, to serve as a means of identifying the users corresponding
to these security principals. When these local identifiers are
translated to the form of the owner attribute, associated with files
created by such principals, they identify, in a common format, the
users associated with each corresponding set of security principals.

The translation used to interpret owner and group strings is not
specified as part of the protocol. This allows various solutions to
be employed. For example, a local translation table may be consulted
that maps a numeric identifier to the user@dns_domain syntax. A name
service may also be used to accomplish the translation. A server may
provide a more general service, not limited by any particular
translation (which would only translate a limited set of possible
strings) by storing the owner and owner_group attributes in local
storage without any translation or it may augment a translation
method by storing the entire string for attributes for which no
translation is available while using the local representation for
those cases in which a translation is available.

Servers that do not provide support for all possible values of the
owner and owner_group attributes SHOULD return an error
(NFS4ERR_BADOWNER) when a string is presented that has no
translation, as the value to be set for a SETATTR of the owner,
owner_group, or acl attributes. When a server does accept an owner
or owner_group value as valid on a SETATTR (and similarly for the
owner and group strings in an acl), it is promising to return that
same string (for which see below) when a corresponding GETATTR is
done. For some internationalization-related exceptions where this is
not possible, see below. Configuration changes (including changes
from the mapping of the string to the local representation) and ill-
constructed name translations (those that contain aliasing) may make
that promise impossible to honor. Servers should make appropriate
efforts to avoid a situation in which these attributes have their
values changed when no real change to ownership has occurred.

The "dns_domain" portion of the owner string is meant to be a DNS
domain name. For example, user@example.org. Servers should accept
as valid a set of users for at least one domain. A server may treat
other domains as having no valid translations. A more general
service is provided when a server is capable of accepting users for
multiple domains, or for all domains, subject to security
constraints.

As an implementation guide, both clients and servers may provide a
means to configure the "dns_domain" portion of the owner string. For
example, the DNS domain name might be "lab.example.org", but the user
names are defined in "example.org". In the absence of such a
configuration, or as a default, the current DNS domain name of the
server should be the value used for the "dns_domain".

As mentioned above, it is desirable that a server when accepting a
string of the form user@domain or group@domain in an attribute,
return this same string when that corresponding attribute is fetched.
Internationalization issues (for a general discussion of which see
Section 12) may make this impossible and the client needs to take
note of the following situations:

o The string representing the domain may be converted to equivalent
   U-label (see [RFC5890]), if presented using a form other than a a
   U-label. See Section 12.4 for details.

o The user or group may be returned in a different form, due to
   normalization issues, although it will always be a canonically
   equivalent string.

In the case where there is no translation available to the client or
server, the attribute value will be constructed without the "@".
Therefore, the absence of the "@" from the owner or owner_group
attribute signifies that no translation was available at the sender
and that the receiver of the attribute should not use that string as
a basis for translation into its own internal format. Even though
the attribute value cannot be translated, it may still be useful. In
the case of a client, the attribute string may be used for local
display of ownership.

To provide a greater degree of compatibility with NFSv3, which
identified users and groups by 32-bit unsigned user identifiers and
group identifiers, owner and group strings that consist of ASCII-
encoded decimal numeric values with no leading zeros can be given a
special interpretation by clients and servers that choose to provide
such support. The receiver may treat such a user or group string as
representing the same user as would be represented by an NFSv3 uid or
gid having the corresponding numeric value.

A server SHOULD reject such a numeric value if the security mechanism
is kerberized. I.e., in such a scenario, the client will already
need to form "user@domain" strings. For any other security
mechanism, the server SHOULD accept such numeric values. As an
implementation note, the server could make such an acceptance be
configurable. If the server does not support numeric values or if it
is configured off, then it MUST return an NFS4ERR_BADOWNER error. If

the security mechanism is kerberized and the client attempts to use
the special form, then the server SHOULD return an NFS4ERR_BADOWNER
error when there is a valid translation for the user or owner
designated in this way. In that case, the client must use the
appropriate user@domain string and not the special form for
compatibility.

The client MUST always accept numeric values if the security
mechanism is not RPCSEC_GSS. A client can determine if a server
supports numeric identifiers by first attempting to provide a numeric
identifier. If this attempt rejected with an NFS4ERR_BADOWNER error,
the the client should only use named identifiers of the form "user@
dns_domain".

The owner string "nobody" may be used to designate an anonymous user,
which will be associated with a file created by a security principal
that cannot be mapped through normal means to the owner attribute.

5.10. Character Case Attributes

With respect to the case_insensitive and case_preserving attributes,
each Universal Multiple-octet coded Character Set-4 (UCS-4)
[ISO.10646-1.1993] character (which UTF-8 encodes) has a "long
descriptive name" RFC1345 [RFC1345] which may or may not include the
word "CAPITAL" or "SMALL". The presence of SMALL or CAPITAL allows
an NFS server to implement unambiguous and efficient table driven
mappings for case insensitive comparisons, and non-case-preserving
storage, although there are variations that occur additional
characters with a name including "SMALL" or "CAPITAL" are added in a
subsequent version of Unicode.

6. Access Control Attributes

Access Control Lists (ACLs) are file attributes that specify fine
grained access control. This chapter covers the "acl", "aclsupport",
"mode", file attributes, and their interactions. Note that file
attributes may apply to any file system object.

6.1. Goals

ACLs and modes represent two well established models for specifying
permissions. This chapter specifies requirements that attempt to
meet the following goals:

o If a server supports the mode attribute, it should provide
   reasonable semantics to clients that only set and retrieve the
   mode attribute.

   o If a server supports ACL attributes, it should provide reasonable
      semantics to clients that only set and retrieve those attributes.

   o On servers that support the mode attribute, if ACL attributes have
      never been set on an object, via inheritance or explicitly, the
      behavior should be traditional UNIX-like behavior.

   o On servers that support the mode attribute, if the ACL attributes
      have been previously set on an object, either explicitly or via
      inheritance:

      * Setting only the mode attribute should effectively control the
         traditional UNIX-like permissions of read, write, and execute
         on owner, owner_group, and other.

      * Setting only the mode attribute should provide reasonable
         security. For example, setting a mode of 000 should be enough
         to ensure that future opens for read or write by any principal
         fail, regardless of a previously existing or inherited ACL.

   o When a mode attribute is set on an object, the ACL attributes may
      need to be modified so as to not conflict with the new mode. In
      such cases, it is desirable that the ACL keep as much information
      as possible. This includes information about inheritance, AUDIT
      and ALARM ACEs, and permissions granted and denied that do not
      conflict with the new mode.

6.2. File Attributes Discussion

6.2.1. Attribute 12: acl

   The NFSv4.0 ACL attribute contains an array of access control entries
   (ACEs) that are associated with the file system object. Although the
   client can read and write the acl attribute, the server is
   responsible for using the ACL to perform access control. The client
   can use the OPEN or ACCESS operations to check access without
   modifying or reading data or metadata.

   The NFS ACE structure is defined as follows:

   typedef uint32_t   acetype4;

   typedef uint32_t aceflag4;

   typedef uint32_t   acemask4;

```
struct nfsace4 {
        acetype4                type;
        aceflag4                flag;
        acemask4                access_mask;
        utf8str_mixed           who;
};
```

To determine if a request succeeds, the server processes each nfsace4
entry in order. Only ACEs which have a "who" that matches the
requester are considered. Each ACE is processed until all of the
bits of the requester's access have been ALLOWED. Once a bit (see
below) has been ALLOWED by an ACCESS_ALLOWED_ACE, it is no longer
considered in the processing of later ACEs. If an ACCESS_DENIED_ACE
is encountered where the requester's access still has unALLOWED bits
in common with the "access_mask" of the ACE, the request is denied.
When the ACL is fully processed, if there are bits in the requester's
mask that have not been ALLOWED or DENIED, access is denied.

Unlike the ALLOW and DENY ACE types, the ALARM and AUDIT ACE types do
not affect a requester's access, and instead are for triggering
events as a result of a requester's access attempt. Therefore, AUDIT
and ALARM ACEs are processed only after processing ALLOW and DENY
ACEs.

The NFSv4.0 ACL model is quite rich. Some server platforms may
provide access control functionality that goes beyond the UNIX-style
mode attribute, but which is not as rich as the NFS ACL model. So
that users can take advantage of this more limited functionality, the
server may support the acl attributes by mapping between its ACL
model and the NFSv4.0 ACL model. Servers must ensure that the ACL
they actually store or enforce is at least as strict as the NFSv4 ACL
that was set. It is tempting to accomplish this by rejecting any ACL
that falls outside the small set that can be represented accurately.
However, such an approach can render ACLs unusable without special
client-side knowledge of the server's mapping, which defeats the
purpose of having a common NFSv4 ACL protocol. Therefore servers
should accept every ACL that they can without compromising security.
To help accomplish this, servers may make a special exception, in the
case of unsupported permission bits, to the rule that bits not
ALLOWED or DENIED by an ACL must be denied. For example, a UNIX-
style server might choose to silently allow read attribute
permissions even though an ACL does not explicitly allow those
permissions. (An ACL that explicitly denies permission to read
attributes should still be rejected.)

The situation is complicated by the fact that a server may have
multiple modules that enforce ACLs. For example, the enforcement for
NFSv4.0 access may be different from, but not weaker than, the

enforcement for local access, and both may be different from the
enforcement for access through other protocols such as Server Message
Block (SMB). So it may be useful for a server to accept an ACL even
if not all of its modules are able to support it.

The guiding principle with regard to NFSv4 access is that the server
must not accept ACLs that appear to make access to the file more
restrictive than it really is.

6.2.1.1. ACE Type

The constants used for the type field (acetype4) are as follows:

```
const ACE4_ACCESS_ALLOWED_ACE_TYPE      = 0x00000000;
const ACE4_ACCESS_DENIED_ACE_TYPE       = 0x00000001;
const ACE4_SYSTEM_AUDIT_ACE_TYPE        = 0x00000002;
const ACE4_SYSTEM_ALARM_ACE_TYPE        = 0x00000003;
```

All four bit types are permitted in the acl attribute.

Value	Abbreviation	Description
ACE4_ACCESS_ALLOWED_ACE_TYPE	ALLOW	Explicitly grants the access defined in acemask4 to the file or directory.
ACE4_ACCESS_DENIED_ACE_TYPE	DENY	Explicitly denies the access defined in acemask4 to the file or directory.
ACE4_SYSTEM_AUDIT_ACE_TYPE	AUDIT	LOG (in a system dependent way) any access attempt to a file or directory which uses any of the access methods specified in acemask4.

```
| ACE4_SYSTEM_ALARM_ACE_TYPE   | ALARM        | Generate a system    |
|                              |              | ALARM (system        |
|                              |              | dependent) when any  |
|                              |              | access attempt is    |
|                              |              | made to a file or    |
|                              |              | directory for the    |
|                              |              | access methods       |
|                              |              | specified in         |
|                              |              | acemask4.            |
+------------------------------+--------------+----------------------+
```

   The "Abbreviation" column denotes how the types will be referred to
                   throughout the rest of this chapter.

6.2.1.2. Attribute 13: aclsupport

   A server need not support all of the above ACE types. This attribute
   indicates which ACE types are supported for the current file system.
   The bitmask constants used to represent the above definitions within
   the aclsupport attribute are as follows:

```
   const ACL4_SUPPORT_ALLOW_ACL   = 0x00000001;
   const ACL4_SUPPORT_DENY_ACL    = 0x00000002;
   const ACL4_SUPPORT_AUDIT_ACL   = 0x00000004;
   const ACL4_SUPPORT_ALARM_ACL   = 0x00000008;
```

   Servers which support either the ALLOW or DENY ACE type SHOULD
   support both ALLOW and DENY ACE types.

   Clients should not attempt to set an ACE unless the server claims
   support for that ACE type. If the server receives a request to set
   an ACE that it cannot store, it MUST reject the request with
   NFS4ERR_ATTRNOTSUPP. If the server receives a request to set an ACE
   that it can store but cannot enforce, the server SHOULD reject the
   request with NFS4ERR_ATTRNOTSUPP.

   Support for any of the ACL attributes is optional (albeit,
   RECOMMENDED).

6.2.1.3. ACE Access Mask

   The bitmask constants used for the access mask field are as follows:

```
const ACE4_READ_DATA            = 0x00000001;
const ACE4_LIST_DIRECTORY       = 0x00000001;
const ACE4_WRITE_DATA           = 0x00000002;
const ACE4_ADD_FILE             = 0x00000002;
const ACE4_APPEND_DATA          = 0x00000004;
const ACE4_ADD_SUBDIRECTORY     = 0x00000004;
const ACE4_READ_NAMED_ATTRS     = 0x00000008;
const ACE4_WRITE_NAMED_ATTRS    = 0x00000010;
const ACE4_EXECUTE              = 0x00000020;
const ACE4_DELETE_CHILD         = 0x00000040;
const ACE4_READ_ATTRIBUTES      = 0x00000080;
const ACE4_WRITE_ATTRIBUTES     = 0x00000100;

const ACE4_DELETE               = 0x00010000;
const ACE4_READ_ACL             = 0x00020000;
const ACE4_WRITE_ACL            = 0x00040000;
const ACE4_WRITE_OWNER          = 0x00080000;
const ACE4_SYNCHRONIZE          = 0x00100000;
```

Note that some masks have coincident values, for example,
ACE4_READ_DATA and ACE4_LIST_DIRECTORY. The mask entries
ACE4_LIST_DIRECTORY, ACE4_ADD_FILE, and ACE4_ADD_SUBDIRECTORY are
intended to be used with directory objects, while ACE4_READ_DATA,
ACE4_WRITE_DATA, and ACE4_APPEND_DATA are intended to be used with
non-directory objects.

6.2.1.3.1. Discussion of Mask Attributes

ACE4_READ_DATA

   Operation(s) affected:

       READ

       OPEN

   Discussion:

       Permission to read the data of the file.

       Servers SHOULD allow a user the ability to read the data of the
       file when only the ACE4_EXECUTE access mask bit is allowed.

ACE4_LIST_DIRECTORY

      Operation(s) affected:

         READDIR

      Discussion:

         Permission to list the contents of a directory.

   ACE4_WRITE_DATA

      Operation(s) affected:

         WRITE

         OPEN

         SETATTR of size

      Discussion:

         Permission to modify a file's data.

   ACE4_ADD_FILE

      Operation(s) affected:

         CREATE

         LINK

         OPEN

         RENAME

      Discussion:

         Permission to add a new file in a directory. The CREATE
         operation is affected when nfs_ftype4 is NF4LNK, NF4BLK,
         NF4CHR, NF4SOCK, or NF4FIFO. (NF4DIR is not listed because it
         is covered by ACE4_ADD_SUBDIRECTORY.) OPEN is affected when
         used to create a regular file. LINK and RENAME are always
         affected.

   ACE4_APPEND_DATA

   Operation(s) affected:

      WRITE

      OPEN

      SETATTR of size

   Discussion:

      The ability to modify a file's data, but only starting at EOF.
      This allows for the notion of append-only files, by allowing
      ACE4_APPEND_DATA and denying ACE4_WRITE_DATA to the same user
      or group. If a file has an ACL such as the one described above
      and a WRITE request is made for somewhere other than EOF, the
      server SHOULD return NFS4ERR_ACCESS.

ACE4_ADD_SUBDIRECTORY

   Operation(s) affected:

      CREATE

      RENAME

   Discussion:

      Permission to create a subdirectory in a directory. The CREATE
      operation is affected when nfs_ftype4 is NF4DIR. The RENAME
      operation is always affected.

ACE4_READ_NAMED_ATTRS

   Operation(s) affected:

      OPENATTR

   Discussion:

      Permission to read the named attributes of a file or to lookup
      the named attributes directory. OPENATTR is affected when it
      is not used to create a named attribute directory. This is
      when 1.) createdir is TRUE, but a named attribute directory
      already exists, or 2.) createdir is FALSE.

   ACE4_WRITE_NAMED_ATTRS

      Operation(s) affected:

         OPENATTR

      Discussion:

         Permission to write the named attributes of a file or to create
         a named attribute directory. OPENATTR is affected when it is
         used to create a named attribute directory. This is when
         createdir is TRUE and no named attribute directory exists. The
         ability to check whether or not a named attribute directory
         exists depends on the ability to look it up, therefore, users
         also need the ACE4_READ_NAMED_ATTRS permission in order to
         create a named attribute directory.

   ACE4_EXECUTE

      Operation(s) affected:

         READ

      Discussion:

         Permission to execute a file.

         Servers SHOULD allow a user the ability to read the data of the
         file when only the ACE4_EXECUTE access mask bit is allowed.
         This is because there is no way to execute a file without
         reading the contents. Though a server may treat ACE4_EXECUTE
         and ACE4_READ_DATA bits identically when deciding to permit a
         READ operation, it SHOULD still allow the two bits to be set
         independently in ACLs, and MUST distinguish between them when
         replying to ACCESS operations. In particular, servers SHOULD
         NOT silently turn on one of the two bits when the other is set,
         as that would make it impossible for the client to correctly
         enforce the distinction between read and execute permissions.

         As an example, following a SETATTR of the following ACL:

         nfsuser:ACE4_EXECUTE:ALLOW

         A subsequent GETATTR of ACL for that file SHOULD return:

         nfsuser:ACE4_EXECUTE:ALLOW

Rather than:

nfsuser:ACE4_EXECUTE/ACE4_READ_DATA:ALLOW

ACE4_EXECUTE

Operation(s) affected:

LOOKUP

OPEN

REMOVE

RENAME

LINK

CREATE

Discussion:

Permission to traverse/search a directory.

ACE4_DELETE_CHILD

Operation(s) affected:

REMOVE

RENAME

Discussion:

Permission to delete a file or directory within a directory.
See [Section 6.2.1.3.2](#) for information on how ACE4_DELETE and
ACE4_DELETE_CHILD interact.

ACE4_READ_ATTRIBUTES

Operation(s) affected:

GETATTR of file system object attributes

VERIFY

NVERIFY

READDIR

Discussion:

The ability to read basic attributes (non-ACLs) of a file. On
a UNIX system, basic attributes can be thought of as the stat
level attributes. Allowing this access mask bit would mean the
entity can execute "ls -l" and stat. If a READDIR operation
requests attributes, this mask must be allowed for the READDIR
to succeed.

ACE4_WRITE_ATTRIBUTES

Operation(s) affected:

SETATTR of time_access_set, time_backup,

time_create, time_modify_set, mimetype, hidden, system

Discussion:

Permission to change the times associated with a file or
directory to an arbitrary value. Also permission to change the
mimetype, hidden and system attributes. A user having
ACE4_WRITE_DATA or ACE4_WRITE_ATTRIBUTES will be allowed to set
the times associated with a file to the current server time.

ACE4_DELETE

Operation(s) affected:

REMOVE

Discussion:

Permission to delete the file or directory. See
Section 6.2.1.3.2 for information on ACE4_DELETE and
ACE4_DELETE_CHILD interact.

ACE4_READ_ACL

Operation(s) affected:

         GETATTR of acl

         NVERIFY

         VERIFY

      Discussion:

         Permission to read the ACL.

   ACE4_WRITE_ACL

      Operation(s) affected:

         SETATTR of acl and mode

      Discussion:

         Permission to write the acl and mode attributes.

   ACE4_WRITE_OWNER

      Operation(s) affected:

         SETATTR of owner and owner_group

      Discussion:

         Permission to write the owner and owner_group attributes. On
         UNIX systems, this is the ability to execute chown() and
         chgrp().

   ACE4_SYNCHRONIZE

      Operation(s) affected:

         NONE

      Discussion:

         Permission to use the file object as a synchronization
         primitive for interprocess communication. This permission is
         not enforced or interpreted by the NFSv4.0 server on behalf of
         the client.

         Typically, the ACE4_SYNCHRONIZE permission is only meaningful
         on local file systems, i.e., file systems not accessed via
         NFSv4.0. The reason that the permission bit exists is that

some operating environments, such as Windows, use
ACE4_SYNCHRONIZE.

For example, if a client copies a file that has
ACE4_SYNCHRONIZE set from a local file system to an NFSv4.0
server, and then later copies the file from the NFSv4.0 server
to a local file system, it is likely that if ACE4_SYNCHRONIZE
was set in the original file, the client will want it set in
the second copy. The first copy will not have the permission
set unless the NFSv4.0 server has the means to set the
ACE4_SYNCHRONIZE bit. The second copy will not have the
permission set unless the NFSv4.0 server has the means to
retrieve the ACE4_SYNCHRONIZE bit.

Server implementations need not provide the granularity of control
that is implied by this list of masks. For example, POSIX-based
systems might not distinguish ACE4_APPEND_DATA (the ability to append
to a file) from ACE4_WRITE_DATA (the ability to modify existing
contents); both masks would be tied to a single "write" permission.
When such a server returns attributes to the client, it would show
both ACE4_APPEND_DATA and ACE4_WRITE_DATA if and only if the write
permission is enabled.

If a server receives a SETATTR request that it cannot accurately
implement, it should err in the direction of more restricted access,
except in the previously discussed cases of execute and read. For
example, suppose a server cannot distinguish overwriting data from
appending new data, as described in the previous paragraph. If a
client submits an ALLOW ACE where ACE4_APPEND_DATA is set but
ACE4_WRITE_DATA is not (or vice versa), the server should either turn
off ACE4_APPEND_DATA or reject the request with NFS4ERR_ATTRNOTSUPP.

6.2.1.3.2. ACE4_DELETE vs. ACE4_DELETE_CHILD

Two access mask bits govern the ability to delete a directory entry:
ACE4_DELETE on the object itself (the "target"), and
ACE4_DELETE_CHILD on the containing directory (the "parent").

Many systems also take the "sticky bit" (MODE4_SVTX) on a directory
to allow unlink only to a user that owns either the target or the
parent; on some such systems the decision also depends on whether the
target is writable.

Servers SHOULD allow unlink if either ACE4_DELETE is permitted on the
target, or ACE4_DELETE_CHILD is permitted on the parent. (Note that
this is true even if the parent or target explicitly denies one of
these permissions.)

If the ACLs in question neither explicitly ALLOW nor DENY either of
the above, and if MODE4_SVTX is not set on the parent, then the
server SHOULD allow the removal if and only if ACE4_ADD_FILE is
permitted. In the case where MODE4_SVTX is set, the server may also
require the remover to own either the parent or the target, or may
require the target to be writable.

This allows servers to support something close to traditional UNIX-
like semantics, with ACE4_ADD_FILE taking the place of the write bit.

6.2.1.4. ACE flag

The bitmask constants used for the flag field are as follows:

```
const ACE4_FILE_INHERIT_ACE           = 0x00000001;
const ACE4_DIRECTORY_INHERIT_ACE      = 0x00000002;
const ACE4_NO_PROPAGATE_INHERIT_ACE   = 0x00000004;
const ACE4_INHERIT_ONLY_ACE           = 0x00000008;
const ACE4_SUCCESSFUL_ACCESS_ACE_FLAG = 0x00000010;
const ACE4_FAILED_ACCESS_ACE_FLAG     = 0x00000020;
const ACE4_IDENTIFIER_GROUP           = 0x00000040;
```

A server need not support any of these flags. If the server supports
flags that are similar to, but not exactly the same as, these flags,
the implementation may define a mapping between the protocol-defined
flags and the implementation-defined flags.

For example, suppose a client tries to set an ACE with
ACE4_FILE_INHERIT_ACE set but not ACE4_DIRECTORY_INHERIT_ACE. If the
server does not support any form of ACL inheritance, the server
should reject the request with NFS4ERR_ATTRNOTSUPP. If the server
supports a single "inherit ACE" flag that applies to both files and
directories, the server may reject the request (i.e., requiring the
client to set both the file and directory inheritance flags). The
server may also accept the request and silently turn on the
ACE4_DIRECTORY_INHERIT_ACE flag.

6.2.1.4.1. Discussion of Flag Bits

ACE4_FILE_INHERIT_ACE
   Any non-directory file in any sub-directory will get this ACE
   inherited.

ACE4_DIRECTORY_INHERIT_ACE
   Can be placed on a directory and indicates that this ACE should be
   added to each new directory created.
   If this flag is set in an ACE in an ACL attribute to be set on a
   non-directory file system object, the operation attempting to set

      the ACL SHOULD fail with NFS4ERR_ATTRNOTSUPP.

   ACE4_INHERIT_ONLY_ACE
      Can be placed on a directory but does not apply to the directory;
      ALLOW and DENY ACEs with this bit set do not affect access to the
      directory, and AUDIT and ALARM ACEs with this bit set do not
      trigger log or alarm events. Such ACEs only take effect once they
      are applied (with this bit cleared) to newly created files and
      directories as specified by the above two flags.
      If this flag is present on an ACE, but neither
      ACE4_DIRECTORY_INHERIT_ACE nor ACE4_FILE_INHERIT_ACE is present,
      then an operation attempting to set such an attribute SHOULD fail
      with NFS4ERR_ATTRNOTSUPP.

   ACE4_NO_PROPAGATE_INHERIT_ACE
      Can be placed on a directory. This flag tells the server that
      inheritance of this ACE should stop at newly created child
      directories.

   ACE4_SUCCESSFUL_ACCESS_ACE_FLAG

   ACE4_FAILED_ACCESS_ACE_FLAG
      The ACE4_SUCCESSFUL_ACCESS_ACE_FLAG (SUCCESS) and
      ACE4_FAILED_ACCESS_ACE_FLAG (FAILED) flag bits may be set only on
      ACE4_SYSTEM_AUDIT_ACE_TYPE (AUDIT) and ACE4_SYSTEM_ALARM_ACE_TYPE
      (ALARM) ACE types. If during the processing of the file's ACL,
      the server encounters an AUDIT or ALARM ACE that matches the
      principal attempting the OPEN, the server notes that fact, and the
      presence, if any, of the SUCCESS and FAILED flags encountered in
      the AUDIT or ALARM ACE. Once the server completes the ACL
      processing, it then notes if the operation succeeded or failed.
      If the operation succeeded, and if the SUCCESS flag was set for a
      matching AUDIT or ALARM ACE, then the appropriate AUDIT or ALARM
      event occurs. If the operation failed, and if the FAILED flag was
      set for the matching AUDIT or ALARM ACE, then the appropriate
      AUDIT or ALARM event occurs. Either or both of the SUCCESS or
      FAILED can be set, but if neither is set, the AUDIT or ALARM ACE
      is not useful.

      The previously described processing applies to ACCESS operations
      even when they return NFS4_OK. For the purposes of AUDIT and
      ALARM, we consider an ACCESS operation to be a "failure" if it
      fails to return a bit that was requested and supported.

   ACE4_IDENTIFIER_GROUP
      Indicates that the "who" refers to a GROUP as defined under UNIX
      or a GROUP ACCOUNT as defined under Windows. Clients and servers
      MUST ignore the ACE4_IDENTIFIER_GROUP flag on ACEs with a who

value equal to one of the special identifiers outlined in
Section 6.2.1.5.

6.2.1.5. ACE Who

The "who" field of an ACE is an identifier that specifies the
principal or principals to whom the ACE applies. It may refer to a
user or a group, with the flag bit ACE4_IDENTIFIER_GROUP specifying
which.

There are several special identifiers which need to be understood
universally, rather than in the context of a particular DNS domain.
Some of these identifiers cannot be understood when an NFS client
accesses the server, but have meaning when a local process accesses
the file. The ability to display and modify these permissions is
permitted over NFS, even if none of the access methods on the server
understands the identifiers.

```
+---------------+----------------------------------------------------+
| Who           | Description                                        |
+---------------+----------------------------------------------------+
| OWNER         | The owner of the file                              |
| GROUP         | The group associated with the file.                |
| EVERYONE      | The world, including the owner and owning group.   |
| INTERACTIVE   | Accessed from an interactive terminal.             |
| NETWORK       | Accessed via the network.                          |
| DIALUP        | Accessed as a dialup user to the server.           |
| BATCH         | Accessed from a batch job.                          |
| ANONYMOUS     | Accessed without any authentication.               |
| AUTHENTICATED | Any authenticated user (opposite of ANONYMOUS)     |
| SERVICE       | Access from a system service.                      |
+---------------+----------------------------------------------------+
```

Table 4

To avoid conflict, these special identifiers are distinguished by an
appended "@" and should appear in the form "xxxx@" (with no domain
name after the "@"). For example: ANONYMOUS@.

The ACE4_IDENTIFIER_GROUP flag MUST be ignored on entries with these
special identifiers. When encoding entries with these special
identifiers, the ACE4_IDENTIFIER_GROUP flag SHOULD be set to zero.

6.2.1.5.1. Discussion of EVERYONE@

It is important to note that "EVERYONE@" is not equivalent to the
UNIX "other" entity. This is because, by definition, UNIX "other"
does not include the owner or owning group of a file. "EVERYONE@"

means literally everyone, including the owner or owning group.

6.2.2. Attribute 33: mode

The NFSv4.0 mode attribute is based on the UNIX mode bits. The
following bits are defined:

```
const MODE4_SUID = 0x800;  /* set user id on execution */
const MODE4_SGID = 0x400;  /* set group id on execution */
const MODE4_SVTX = 0x200;  /* save text even after use */
const MODE4_RUSR = 0x100;  /* read permission: owner */
const MODE4_WUSR = 0x080;  /* write permission: owner */
const MODE4_XUSR = 0x040;  /* execute permission: owner */
const MODE4_RGRP = 0x020;  /* read permission: group */
const MODE4_WGRP = 0x010;  /* write permission: group */
const MODE4_XGRP = 0x008;  /* execute permission: group */
const MODE4_ROTH = 0x004;  /* read permission: other */
const MODE4_WOTH = 0x002;  /* write permission: other */
const MODE4_XOTH = 0x001;  /* execute permission: other */
```

Bits MODE4_RUSR, MODE4_WUSR, and MODE4_XUSR apply to the principal
identified in the owner attribute. Bits MODE4_RGRP, MODE4_WGRP, and
MODE4_XGRP apply to principals identified in the owner_group
attribute but who are not identified in the owner attribute. Bits
MODE4_ROTH, MODE4_WOTH, MODE4_XOTH apply to any principal that does
not match that in the owner attribute, and does not have a group
matching that of the owner_group attribute.

Bits within the mode other than those specified above are not defined
by this protocol. A server MUST NOT return bits other than those
defined above in a GETATTR or READDIR operation, and it MUST return
NFS4ERR_INVAL if bits other than those defined above are set in a
SETATTR, CREATE, OPEN, VERIFY or NVERIFY operation.

6.3. Common Methods

The requirements in this section will be referred to in future
sections, especially Section 6.4.

6.3.1. Interpreting an ACL

6.3.1.1. Server Considerations

The server uses the algorithm described in Section 6.2.1 to determine
whether an ACL allows access to an object. However, the ACL may not
be the sole determiner of access. For example:

o In the case of a file system exported as read-only, the server may
   deny write permissions even though an object's ACL grants it.

o Server implementations MAY grant ACE4_WRITE_ACL and ACE4_READ_ACL
   permissions to prevent a situation from arising in which there is
   no valid way to ever modify the ACL.

o All servers will allow a user the ability to read the data of the
   file when only the execute permission is granted (i.e., If the ACL
   denies the user the ACE4_READ_DATA access and allows the user
   ACE4_EXECUTE, the server will allow the user to read the data of
   the file).

o Many servers have the notion of owner-override in which the owner
   of the object is allowed to override accesses that are denied by
   the ACL. This may be helpful, for example, to allow users
   continued access to open files on which the permissions have
   changed.

o Many servers have the notion of a "superuser" that has privileges
   beyond an ordinary user. The superuser may be able to read or
   write data or metadata in ways that would not be permitted by the
   ACL.

6.3.1.2. Client Considerations

Clients SHOULD NOT do their own access checks based on their
interpretation the ACL, but rather use the OPEN and ACCESS operations
to do access checks. This allows the client to act on the results of
having the server determine whether or not access should be granted
based on its interpretation of the ACL.

Clients must be aware of situations in which an object's ACL will
define a certain access even though the server will not enforce it.
In general, but especially in these situations, the client needs to
do its part in the enforcement of access as defined by the ACL. To
do this, the client MAY send the appropriate ACCESS operation prior
to servicing the request of the user or application in order to
determine whether the user or application should be granted the
access requested. For examples in which the ACL may define accesses
that the server doesn't enforce see Section 6.3.1.1.

6.3.2. Computing a Mode Attribute from an ACL

The following method can be used to calculate the MODE4_R*, MODE4_W*
and MODE4_X* bits of a mode attribute, based upon an ACL.

First, for each of the special identifiers OWNER@, GROUP@, and

   EVERYONE@, evaluate the ACL in order, considering only ALLOW and DENY
   ACEs for the identifier EVERYONE@ and for the identifier under
   consideration. The result of the evaluation will be an NFSv4 ACL
   mask showing exactly which bits are permitted to that identifier.

   Then translate the calculated mask for OWNER@, GROUP@, and EVERYONE@
   into mode bits for, respectively, the user, group, and other, as
   follows:

   1. Set the read bit (MODE4_RUSR, MODE4_RGRP, or MODE4_ROTH) if and
       only if ACE4_READ_DATA is set in the corresponding mask.

   2. Set the write bit (MODE4_WUSR, MODE4_WGRP, or MODE4_WOTH) if and
       only if ACE4_WRITE_DATA and ACE4_APPEND_DATA are both set in the
       corresponding mask.

   3. Set the execute bit (MODE4_XUSR, MODE4_XGRP, or MODE4_XOTH), if
       and only if ACE4_EXECUTE is set in the corresponding mask.

6.3.2.1. Discussion

   Some server implementations also add bits permitted to named users
   and groups to the group bits (MODE4_RGRP, MODE4_WGRP, and
   MODE4_XGRP).

   Implementations are discouraged from doing this, because it has been
   found to cause confusion for users who see members of a file's group
   denied access that the mode bits appear to allow. (The presence of
   DENY ACEs may also lead to such behavior, but DENY ACEs are expected
   to be more rarely used.)

   The same user confusion seen when fetching the mode also results if
   setting the mode does not effectively control permissions for the
   owner, group, and other users; this motivates some of the
   requirements that follow.

6.4. Requirements

   The server that supports both mode and ACL must take care to
   synchronize the MODE4_*USR, MODE4_*GRP, and MODE4_*OTH bits with the
   ACEs which have respective who fields of "OWNER@", "GROUP@", and
   "EVERYONE@" so that the client can see semantically equivalent access
   permissions exist whether the client asks for owner, owner_group and
   mode attributes, or for just the ACL.

   Many requirements refer to Section 6.3.2, but note that the methods
   have behaviors specified with "SHOULD". This is intentional, to
   avoid invalidating existing implementations that compute the mode

according to the withdrawn POSIX ACL draft ([P1003.1e]), rather than
by actual permissions on owner, group, and other.

6.4.1. **Setting the mode and/or ACL Attributes**

6.4.1.1. **Setting mode and not ACL**

When any of the nine low-order mode bits are changed because the mode
attribute was set, and no ACL attribute is explicitly set, the acl
attribute must be modified in accordance with the updated value of
those bits. This must happen even if the value of the low-order bits
is the same after the mode is set as before.

Note that any AUDIT or ALARM ACEs are unaffected by changes to the
mode.

In cases in which the permissions bits are subject to change, the acl
attribute MUST be modified such that the mode computed via the method
in Section 6.3.2 yields the low-order nine bits (MODE4_R*, MODE4_W*,
MODE4_X*) of the mode attribute as modified by the attribute change.
The ACL attributes SHOULD also be modified such that:

1. If MODE4_RGRP is not set, entities explicitly listed in the ACL
    other than OWNER@ and EVERYONE@ SHOULD NOT be granted
    ACE4_READ_DATA.

2. If MODE4_WGRP is not set, entities explicitly listed in the ACL
    other than OWNER@ and EVERYONE@ SHOULD NOT be granted
    ACE4_WRITE_DATA or ACE4_APPEND_DATA.

3. If MODE4_XGRP is not set, entities explicitly listed in the ACL
    other than OWNER@ and EVERYONE@ SHOULD NOT be granted
    ACE4_EXECUTE.

Access mask bits other than those listed above, appearing in ALLOW
ACEs, MAY also be disabled.

Note that ACEs with the flag ACE4_INHERIT_ONLY_ACE set do not affect
the permissions of the ACL itself, nor do ACEs of the type AUDIT and
ALARM. As such, it is desirable to leave these ACEs unmodified when
modifying the ACL attributes.

Also note that the requirement may be met by discarding the acl in
favor of an ACL that represents the mode and only the mode. This is
permitted, but it is preferable for a server to preserve as much of
the ACL as possible without violating the above requirements.
Discarding the ACL makes it effectively impossible for a file created
with a mode attribute to inherit an ACL (see Section 6.4.3).

6.4.1.2. Setting ACL and not mode

   When setting the acl and not setting the mode attribute, the
   permission bits of the mode need to be derived from the ACL. In this
   case, the ACL attribute SHOULD be set as given. The nine low-order
   bits of the mode attribute (MODE4_R*, MODE4_W*, MODE4_X*) MUST be
   modified to match the result of the method Section 6.3.2. The three
   high-order bits of the mode (MODE4_SUID, MODE4_SGID, MODE4_SVTX)
   SHOULD remain unchanged.

6.4.1.3. Setting both ACL and mode

   When setting both the mode and the acl attribute in the same
   operation, the attributes MUST be applied in this order: mode, then
   ACL. The mode-related attribute is set as given, then the ACL
   attribute is set as given, possibly changing the final mode, as
   described above in Section 6.4.1.2.

6.4.2. Retrieving the mode and/or ACL Attributes

   This section applies only to servers that support both the mode and
   ACL attributes.

   Some server implementations may have a concept of "objects without
   ACLs", meaning that all permissions are granted and denied according
   to the mode attribute, and that no ACL attribute is stored for that
   object. If an ACL attribute is requested of such a server, the
   server SHOULD return an ACL that does not conflict with the mode;
   that is to say, the ACL returned SHOULD represent the nine low-order
   bits of the mode attribute (MODE4_R*, MODE4_W*, MODE4_X*) as
   described in Section 6.3.2.

   For other server implementations, the ACL attribute is always present
   for every object. Such servers SHOULD store at least the three high-
   order bits of the mode attribute (MODE4_SUID, MODE4_SGID,
   MODE4_SVTX). The server SHOULD return a mode attribute if one is
   requested, and the low-order nine bits of the mode (MODE4_R*,
   MODE4_W*, MODE4_X*) MUST match the result of applying the method in
   Section 6.3.2 to the ACL attribute.

6.4.3. Creating New Objects

   If a server supports any ACL attributes, it may use the ACL
   attributes on the parent directory to compute an initial ACL
   attribute for a newly created object. This will be referred to as
   the inherited ACL within this section. The act of adding one or more
   ACEs to the inherited ACL that are based upon ACEs in the parent
   directory's ACL will be referred to as inheriting an ACE within this

section.

In the presence or absence of the mode and ACL attributes, the
behavior of CREATE and OPEN SHOULD be:

1. If just the mode is given in the call:

    In this case, inheritance SHOULD take place, but the mode MUST be
    applied to the inherited ACL as described in Section 6.4.1.1,
    thereby modifying the ACL.

2. If just the ACL is given in the call:

    In this case, inheritance SHOULD NOT take place, and the ACL as
    defined in the CREATE or OPEN will be set without modification,
    and the mode modified as in Section 6.4.1.2

3. If both mode and ACL are given in the call:

    In this case, inheritance SHOULD NOT take place, and both
    attributes will be set as described in Section 6.4.1.3.

4. If neither mode nor ACL are given in the call:

    In the case where an object is being created without any initial
    attributes at all, e.g., an OPEN operation with an opentype4 of
    OPEN4_CREATE and a createmode4 of EXCLUSIVE4, inheritance SHOULD
    NOT take place. Instead, the server SHOULD set permissions to
    deny all access to the newly created object. It is expected that
    the appropriate client will set the desired attributes in a
    subsequent SETATTR operation, and the server SHOULD allow that
    operation to succeed, regardless of what permissions the object
    is created with. For example, an empty ACL denies all
    permissions, but the server should allow the owner's SETATTR to
    succeed even though WRITE_ACL is implicitly denied.

    In other cases, inheritance SHOULD take place, and no
    modifications to the ACL will happen. The mode attribute, if
    supported, MUST be as computed in Section 6.3.2, with the
    MODE4_SUID, MODE4_SGID and MODE4_SVTX bits clear. If no
    inheritable ACEs exist on the parent directory, the rules for
    creating acl attributes are implementation defined.

6.4.3.1. **The Inherited ACL**

   If the object being created is not a directory, the inherited ACL
   SHOULD NOT inherit ACEs from the parent directory ACL unless the
   ACE4_FILE_INHERIT_FLAG is set.

   If the object being created is a directory, the inherited ACL should
   inherit all inheritable ACEs from the parent directory, those that
   have ACE4_FILE_INHERIT_ACE or ACE4_DIRECTORY_INHERIT_ACE flag set.
   If the inheritable ACE has ACE4_FILE_INHERIT_ACE set, but
   ACE4_DIRECTORY_INHERIT_ACE is clear, the inherited ACE on the newly
   created directory MUST have the ACE4_INHERIT_ONLY_ACE flag set to
   prevent the directory from being affected by ACEs meant for non-
   directories.

   When a new directory is created, the server MAY split any inherited
   ACE which is both inheritable and effective (in other words, which
   has neither ACE4_INHERIT_ONLY_ACE nor ACE4_NO_PROPAGATE_INHERIT_ACE
   set), into two ACEs, one with no inheritance flags, and one with
   ACE4_INHERIT_ONLY_ACE set. This makes it simpler to modify the
   effective permissions on the directory without modifying the ACE
   which is to be inherited to the new directory's children.

7. **NFS Server Name Space**

7.1. **Server Exports**

   On a UNIX server the name space describes all the files reachable by
   pathnames under the root directory or "/". On a Windows NT server
   the name space constitutes all the files on disks named by mapped
   disk letters. NFS server administrators rarely make the entire
   server's file system name space available to NFS clients. More often
   portions of the name space are made available via an "export"
   feature. In previous versions of the NFS protocol, the root
   filehandle for each export is obtained through the MOUNT protocol;
   the client sends a string that identifies the export of name space
   and the server returns the root filehandle for it. The MOUNT
   protocol supports an EXPORTS procedure that will enumerate the
   server's exports.

7.2. **Browsing Exports**

   The NFSv4 protocol provides a root filehandle that clients can use to
   obtain filehandles for these exports via a multi-component LOOKUP. A
   common user experience is to use a graphical user interface (perhaps
   a file "Open" dialog window) to find a file via progressive browsing
   through a directory tree. The client must be able to move from one

    export to another export via single-component, progressive LOOKUP
    operations.

    This style of browsing is not well supported by the NFSv2 and NFSv3
    protocols. The client expects all LOOKUP operations to remain within
    a single server file system. For example, the device attribute will
    not change. This prevents a client from taking name space paths that
    span exports.

    An automounter on the client can obtain a snapshot of the server's
    name space using the EXPORTS procedure of the MOUNT protocol. If it
    understands the server's pathname syntax, it can create an image of
    the server's name space on the client. The parts of the name space
    that are not exported by the server are filled in with a "pseudo file
    system" that allows the user to browse from one mounted file system
    to another. There is a drawback to this representation of the
    server's name space on the client: it is static. If the server
    administrator adds a new export the client will be unaware of it.

7.3. Server Pseudo Filesystem

    NFSv4 servers avoid this name space inconsistency by presenting all
    the exports within the framework of a single server name space. An
    NFSv4 client uses LOOKUP and READDIR operations to browse seamlessly
    from one export to another. Portions of the server name space that
    are not exported are bridged via a "pseudo file system" that provides
    a view of exported directories only. A pseudo file system has a
    unique fsid and behaves like a normal, read only file system.

    Based on the construction of the server's name space, it is possible
    that multiple pseudo file systems may exist. For example,

      /a   pseudo file system
      /a/b   real file system
      /a/b/c   pseudo file system
      /a/b/c/d   real file system

    Each of the pseudo file systems are considered separate entities and
    therefore will have a unique fsid.

7.4. Multiple Roots

    The DOS and Windows operating environments are sometimes described as
    having "multiple roots". Filesystems are commonly represented as
    disk letters. MacOS represents file systems as top level names.
    NFSv4 servers for these platforms can construct a pseudo file system
    above these root names so that disk letters or volume names are
    simply directory names in the pseudo root.

7.5. Filehandle Volatility

The nature of the server's pseudo file system is that it is a logical
representation of file system(s) available from the server.
Therefore, the pseudo file system is most likely constructed
dynamically when the server is first instantiated. It is expected
that the pseudo file system may not have an on disk counterpart from
which persistent filehandles could be constructed. Even though it is
preferable that the server provide persistent filehandles for the
pseudo file system, the NFS client should expect that pseudo file
system filehandles are volatile. This can be confirmed by checking
the associated "fh_expire_type" attribute for those filehandles in
question. If the filehandles are volatile, the NFS client must be
prepared to recover a filehandle value (e.g., with a multi-component
LOOKUP) when receiving an error of NFS4ERR_FHEXPIRED.

7.6. Exported Root

If the server's root file system is exported, one might conclude that
a pseudo-file system is not needed. This would be wrong. Assume the
following file systems on a server:

```
   /       disk1  (exported)
   /a      disk2  (not exported)
   /a/b    disk3  (exported)
```

Because disk2 is not exported, disk3 cannot be reached with simple
LOOKUPs. The server must bridge the gap with a pseudo-file system.

7.7. Mount Point Crossing

The server file system environment may be constructed in such a way
that one file system contains a directory which is 'covered' or
mounted upon by a second file system. For example:

```
   /a/b            (file system 1)
   /a/b/c/d        (file system 2)
```

The pseudo file system for this server may be constructed to look
like:

```
   /               (place holder/not exported)
   /a/b            (file system 1)
   /a/b/c/d        (file system 2)
```

It is the server's responsibility to present the pseudo file system
that is complete to the client. If the client sends a lookup request
for the path "/a/b/c/d", the server's response is the filehandle of

the file system "/a/b/c/d". In previous versions of the NFS
protocol, the server would respond with the filehandle of directory
"/a/b/c/d" within the file system "/a/b".

The NFS client will be able to determine if it crosses a server mount
point by a change in the value of the "fsid" attribute.

7.8. Security Policy and Name Space Presentation

The application of the server's security policy needs to be carefully
considered by the implementor. One may choose to limit the
viewability of portions of the pseudo file system based on the
server's perception of the client's ability to authenticate itself
properly. However, with the support of multiple security mechanisms
and the ability to negotiate the appropriate use of these mechanisms,
the server is unable to properly determine if a client will be able
to authenticate itself. If, based on its policies, the server
chooses to limit the contents of the pseudo file system, the server
may effectively hide file systems from a client that may otherwise
have legitimate access.

As suggested practice, the server should apply the security policy of
a shared resource in the server's namespace to the components of the
resource's ancestors. For example:

```
  /
  /a/b
  /a/b/c
```

The /a/b/c directory is a real file system and is the shared
resource. The security policy for /a/b/c is Kerberos with integrity.
The server should apply the same security policy to /, /a, and /a/b.
This allows for the extension of the protection of the server's
namespace to the ancestors of the real shared resource.

For the case of the use of multiple, disjoint security mechanisms in
the server's resources, the security for a particular object in the
server's namespace should be the union of all security mechanisms of
all direct descendants.

8. Multi-Server Namespace

NFSv4 supports attributes that allow a namespace to extend beyond the
boundaries of a single server. It is RECOMMENDED that clients and
servers support construction of such multi-server namespaces. Use of
such multi-server namespaces is OPTIONAL, however, and for many
purposes, single-server namespaces are perfectly acceptable. Use of

   multi-server namespaces can provide many advantages, however, by
   separating a file system's logical position in a namespace from the
   (possibly changing) logistical and administrative considerations that
   result in particular file systems being located on particular
   servers.

8.1. Location Attributes

   NFSv4 contains RECOMMENDED attributes that allow file systems on one
   server to be associated with one or more instances of that file
   system on other servers. These attributes specify such file system
   instances by specifying a server address target (either as a DNS name
   representing one or more IP addresses or as a literal IP address)
   together with the path of that file system within the associated
   single-server namespace.

   The fs_locations RECOMMENDED attribute allows specification of the
   file system locations where the data corresponding to a given file
   system may be found.

8.2. File System Presence or Absence

   A given location in an NFSv4 namespace (typically but not necessarily
   a multi-server namespace) can have a number of file system instance
   locations associated with it via the fs_locations attribute. There
   may also be an actual current file system at that location,
   accessible via normal namespace operations (e.g., LOOKUP). In this
   case, the file system is said to be "present" at that position in the
   namespace, and clients will typically use it, reserving use of
   additional locations specified via the location-related attributes to
   situations in which the principal location is no longer available.

   When there is no actual file system at the namespace location in
   question, the file system is said to be "absent". An absent file
   system contains no files or directories other than the root. Any
   reference to it, except to access a small set of attributes useful in
   determining alternate locations, will result in an error,
   NFS4ERR_MOVED. Note that if the server ever returns the error
   NFS4ERR_MOVED, it MUST support the fs_locations attribute.

   While the error name suggests that we have a case of a file system
   that once was present, and has only become absent later, this is only
   one possibility. A position in the namespace may be permanently
   absent with the set of file system(s) designated by the location
   attributes being the only realization. The name NFS4ERR_MOVED
   reflects an earlier, more limited conception of its function, but
   this error will be returned whenever the referenced file system is
   absent, whether it has moved or not.

Except in the case of GETATTR-type operations (to be discussed
later), when the current filehandle at the start of an operation is
within an absent file system, that operation is not performed and the
error NFS4ERR_MOVED is returned, to indicate that the file system is
absent on the current server.

Because a GETFH cannot succeed if the current filehandle is within an
absent file system, filehandles within an absent file system cannot
be transferred to the client. When a client does have filehandles
within an absent file system, it is the result of obtaining them when
the file system was present, and having the file system become absent
subsequently.

It should be noted that because the check for the current filehandle
being within an absent file system happens at the start of every
operation, operations that change the current filehandle so that it
is within an absent file system will not result in an error. This
allows such combinations as PUTFH-GETATTR and LOOKUP-GETATTR to be
used to get attribute information, particularly location attribute
information, as discussed below.

8.3. Getting Attributes for an Absent File System

When a file system is absent, most attributes are not available, but
it is necessary to allow the client access to the small set of
attributes that are available, and most particularly that which gives
information about the correct current locations for this file system,
fs_locations.

8.3.1. GETATTR Within an Absent File System

As mentioned above, an exception is made for GETATTR in that
attributes may be obtained for a filehandle within an absent file
system. This exception only applies if the attribute mask contains
at least the fs_locations attribute bit, which indicates the client
is interested in a result regarding an absent file system. If it is
not requested, GETATTR will result in an NFS4ERR_MOVED error.

When a GETATTR is done on an absent file system, the set of supported
attributes is very limited. Many attributes, including those that
are normally REQUIRED, will not be available on an absent file
system. In addition to the fs_locations attribute, the following
attributes SHOULD be available on absent file systems. In the case
of RECOMMENDED attributes, they should be available at least to the
same degree that they are available on present file systems.

fsid: This attribute should be provided so that the client can
   determine file system boundaries, including, in particular, the
   boundary between present and absent file systems. This value must
   be different from any other fsid on the current server and need
   have no particular relationship to fsids on any particular
   destination to which the client might be directed.

mounted_on_fileid: For objects at the top of an absent file system,
   this attribute needs to be available. Since the fileid is within
   the present parent file system, there should be no need to
   reference the absent file system to provide this information.

Other attributes SHOULD NOT be made available for absent file
systems, even when it is possible to provide them. The server should
not assume that more information is always better and should avoid
gratuitously providing additional information.

When a GETATTR operation includes a bit mask for the attribute
fs_locations, but where the bit mask includes attributes that are not
supported, GETATTR will not return an error, but will return the mask
of the actual attributes supported with the results.

Handling of VERIFY/NVERIFY is similar to GETATTR in that if the
attribute mask does not include fs_locations the error NFS4ERR_MOVED
will result. It differs in that any appearance in the attribute mask
of an attribute not supported for an absent file system (and note
that this will include some normally REQUIRED attributes) will also
cause an NFS4ERR_MOVED result.

8.3.2. READDIR and Absent File Systems

A READDIR performed when the current filehandle is within an absent
file system will result in an NFS4ERR_MOVED error, since, unlike the
case of GETATTR, no such exception is made for READDIR.

Attributes for an absent file system may be fetched via a READDIR for
a directory in a present file system, when that directory contains
the root directories of one or more absent file systems. In this
case, the handling is as follows:

o If the attribute set requested includes fs_locations, then
   fetching of attributes proceeds normally and no NFS4ERR_MOVED
   indication is returned, even when the rdattr_error attribute is
   requested.

o If the attribute set requested does not include fs_locations, then
   if the rdattr_error attribute is requested, each directory entry
   for the root of an absent file system will report NFS4ERR_MOVED as

the value of the rdattr_error attribute.

o If the attribute set requested does not include either of the
   attributes fs_locations or rdattr_error then the occurrence of the
   root of an absent file system within the directory will result in
   the READDIR failing with an NFS4ERR_MOVED error.

o The unavailability of an attribute because of a file system's
   absence, even one that is ordinarily REQUIRED, does not result in
   any error indication. The set of attributes returned for the root
   directory of the absent file system in that case is simply
   restricted to those actually available.

8.4. Uses of Location Information

The location-bearing attribute of fs_locations provides, together
with the possibility of absent file systems, a number of important
facilities in providing reliable, manageable, and scalable data
access.

When a file system is present, these attributes can provide
alternative locations, to be used to access the same data, in the
event of server failures, communications problems, or other
difficulties that make continued access to the current file system
impossible or otherwise impractical. Under some circumstances,
multiple alternative locations may be used simultaneously to provide
higher-performance access to the file system in question. Provision
of such alternate locations is referred to as "replication" although
there are cases in which replicated sets of data are not in fact
present, and the replicas are instead different paths to the same
data.

When a file system is present and becomes absent, clients can be
given the opportunity to have continued access to their data, at an
alternate location. In this case, a continued attempt to use the
data in the now-absent file system will result in an NFS4ERR_MOVED
error and, at that point, the successor locations (typically only one
although multiple choices are possible) can be fetched and used to
continue access. Transfer of the file system contents to the new
location is referred to as "migration", but it should be kept in mind
that there are cases in which this term can be used, like
"replication", when there is no actual data migration per se.

Where a file system was not previously present, specification of file
system location provides a means by which file systems located on one
server can be associated with a namespace defined by another server,
thus allowing a general multi-server namespace facility. A
designation of such a location, in place of an absent file system, is

called a "referral".

Because client support for location-related attributes is OPTIONAL, a server may (but is not required to) take action to hide migration and referral events from such clients, by acting as a proxy, for example.

8.4.1. File System Replication

The fs_locations attribute provides alternative locations, to be used to access data in place of or in addition to the current file system instance. On first access to a file system, the client should obtain the value of the set of alternate locations by interrogating the fs_locations attribute.

In the event that server failures, communications problems, or other difficulties make continued access to the current file system impossible or otherwise impractical, the client can use the alternate locations as a way to get continued access to its data. Multiple locations may be used simultaneously, to provide higher performance through the exploitation of multiple paths between client and target file system.

The alternate locations may be physical replicas of the (typically read-only) file system data, or they may reflect alternate paths to the same server or provide for the use of various forms of server clustering in which multiple servers provide alternate ways of accessing the same physical file system. How these different modes of file system transition are represented within the fs_locations attribute and how the client deals with file system transition issues will be discussed in detail below.

Multiple server addresses, whether they are derived from a single entry with a DNS name representing a set of IP addresses or from multiple entries each with its own server address, may correspond to the same actual server.

8.4.2. File System Migration

When a file system is present and becomes absent, clients can be given the opportunity to have continued access to their data, at an alternate location, as specified by the fs_locations attribute. Typically, a client will be accessing the file system in question, get an NFS4ERR_MOVED error, and then use the fs_locations attribute to determine the new location of the data.

Such migration can be helpful in providing load balancing or general resource reallocation. The protocol does not specify how the file system will be moved between servers. It is anticipated that a

   number of different server-to-server transfer mechanisms might be
   used with the choice left to the server implementor. The NFSv4
   protocol specifies the method used to communicate the migration event
   between client and server.

   The new location may be an alternate communication path to the same
   server or, in the case of various forms of server clustering, another
   server providing access to the same physical file system. The
   client's responsibilities in dealing with this transition depend on
   the specific nature of the new access path as well as how and whether
   data was in fact migrated. These issues will be discussed in detail
   below.

   When an alternate location is designated as the target for migration,
   it must designate the same data. Where file systems are writable, a
   change made on the original file system must be visible on all
   migration targets. Where a file system is not writable but
   represents a read-only copy (possibly periodically updated) of a
   writable file system, similar requirements apply to the propagation
   of updates. Any change visible in the original file system must
   already be effected on all migration targets, to avoid any
   possibility that a client, in effecting a transition to the migration
   target, will see any reversion in file system state.

8.4.3. Referrals

   Referrals provide a way of placing a file system in a location within
   the namespace essentially without respect to its physical location on
   a given server. This allows a single server or a set of servers to
   present a multi-server namespace that encompasses file systems
   located on multiple servers. Some likely uses of this include
   establishment of site-wide or organization-wide namespaces, or even
   knitting such together into a truly global namespace.

   Referrals occur when a client determines, upon first referencing a
   position in the current namespace, that it is part of a new file
   system and that the file system is absent. When this occurs,
   typically by receiving the error NFS4ERR_MOVED, the actual location
   or locations of the file system can be determined by fetching the
   fs_locations attribute.

   The locations-related attribute may designate a single file system
   location or multiple file system locations, to be selected based on
   the needs of the client.

   Use of multi-server namespaces is enabled by NFSv4 but is not
   required. The use of multi-server namespaces and their scope will
   depend on the applications used and system administration

   preferences.

   Multi-server namespaces can be established by a single server
   providing a large set of referrals to all of the included file
   systems. Alternatively, a single multi-server namespace may be
   administratively segmented with separate referral file systems (on
   separate servers) for each separately administered portion of the
   namespace. The top-level referral file system or any segment may use
   replicated referral file systems for higher availability.

   Generally, multi-server namespaces are for the most part uniform, in
   that the same data made available to one client at a given location
   in the namespace is made available to all clients at that location.

8.5. Location Entries and Server Identity

   As mentioned above, a single location entry may have a server address
   target in the form of a DNS name that may represent multiple IP
   addresses, while multiple location entries may have their own server
   address targets that reference the same server.

   When multiple addresses for the same server exist, the client may
   assume that for each file system in the namespace of a given server
   network address, there exist file systems at corresponding namespace
   locations for each of the other server network addresses. It may do
   this even in the absence of explicit listing in fs_locations. Such
   corresponding file system locations can be used as alternate
   locations, just as those explicitly specified via the fs_locations
   attribute.

   If a single location entry designates multiple server IP addresses,
   the client cannot assume that these addresses are multiple paths to
   the same server. In most cases, they will be, but the client MUST
   verify that before acting on that assumption. When two server
   addresses are designated by a single location entry and they
   correspond to different servers, this normally indicates some sort of
   misconfiguration, and so the client should avoid using such location
   entries when alternatives are available. When they are not, clients
   should pick one of IP addresses and use it, without using others that
   are not directed to the same server.

8.6. Additional Client-Side Considerations

   When clients make use of servers that implement referrals,
   replication, and migration, care should be taken that a user who
   mounts a given file system that includes a referral or a relocated
   file system continues to see a coherent picture of that user-side
   file system despite the fact that it contains a number of server-side

   file systems that may be on different servers.

   One important issue is upward navigation from the root of a server-
   side file system to its parent (specified as ".." in UNIX), in the
   case in which it transitions to that file system as a result of
   referral, migration, or a transition as a result of replication.
   When the client is at such a point, and it needs to ascend to the
   parent, it must go back to the parent as seen within the multi-server
   namespace rather than sending a LOOKUPP operation to the server,
   which would result in the parent within that server's single-server
   namespace. In order to do this, the client needs to remember the
   filehandles that represent such file system roots and use these
   instead of issuing a LOOKUPP operation to the current server. This
   will allow the client to present to applications a consistent
   namespace, where upward navigation and downward navigation are
   consistent.

   Another issue concerns refresh of referral locations. When referrals
   are used extensively, they may change as server configurations
   change. It is expected that clients will cache information related
   to traversing referrals so that future client-side requests are
   resolved locally without server communication. This is usually
   rooted in client-side name look up caching. Clients should
   periodically purge this data for referral points in order to detect
   changes in location information.

   A potential problem exists if a client were to allow an open owner to
   have state on multiple file systems on server, in that it is unclear
   how the sequence numbers associated with open owners are to be dealt
   with, in the event of transparent state migration. A client can
   avoid such a situation, if it ensures that any use of an open owner
   is confined to a single file system.

   A server MAY decline to migrate state associated with open owners
   that span multiple file systems. In cases in which the server
   chooses not to migrate such state, the server MUST return
   NFS4ERR_BAD_STATEID when the client uses those stateids on the new
   server.

   The server MUST return NFS4ERR_STALE_STATEID when the client uses
   those stateids on the old server, regardless of whether migration has
   occurred or not.

8.7. Effecting File System Referrals

   Referrals are effected when an absent file system is encountered, and
   one or more alternate locations are made available by the
   fs_locations attribute. The client will typically get an

NFS4ERR_MOVED error, fetch the appropriate location information, and
proceed to access the file system on a different server, even though
it retains its logical position within the original namespace.
Referrals differ from migration events in that they happen only when
the client has not previously referenced the file system in question
(so there is nothing to transition). Referrals can only come into
effect when an absent file system is encountered at its root.

The examples given in the sections below are somewhat artificial in
that an actual client will not typically do a multi-component look
up, but will have cached information regarding the upper levels of
the name hierarchy. However, these example are chosen to make the
required behavior clear and easy to put within the scope of a small
number of requests, without getting unduly into details of how
specific clients might choose to cache things.

8.7.1. Referral Example (LOOKUP)

Let us suppose that the following COMPOUND is sent in an environment
in which /this/is/the/path is absent from the target server. This
may be for a number of reasons. It may be the case that the file
system has moved, or it may be the case that the target server is
functioning mainly, or solely, to refer clients to the servers on
which various file systems are located.

o PUTROOTFH

o LOOKUP "this"

o LOOKUP "is"

o LOOKUP "the"

o LOOKUP "path"

o GETFH

o GETATTR(fsid,fileid,size,time_modify)

Under the given circumstances, the following will be the result.

o PUTROOTFH --> NFS_OK. The current fh is now the root of the
   pseudo-fs.

o LOOKUP "this" --> NFS_OK. The current fh is for /this and is
   within the pseudo-fs.

o LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is
   within the pseudo-fs.

o LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and
   is within the pseudo-fs.

o LOOKUP "path" --> NFS_OK. The current fh is for /this/is/the/path
   and is within a new, absent file system, but ... the client will
   never see the value of that fh.

o GETFH --> NFS4ERR_MOVED. Fails because current fh is in an absent
   file system at the start of the operation, and the specification
   makes no exception for GETFH.

o GETATTR(fsid,fileid,size,time_modify) Not executed because the
   failure of the GETFH stops processing of the COMPOUND.

Given the failure of the GETFH, the client has the job of determining
the root of the absent file system and where to find that file
system, i.e., the server and path relative to that server's root fh.
Note here that in this example, the client did not obtain filehandles
and attribute information (e.g., fsid) for the intermediate
directories, so that it would not be sure where the absent file
system starts. It could be the case, for example, that /this/is/the
is the root of the moved file system and that the reason that the
look up of "path" succeeded is that the file system was not absent on
that operation but was moved between the last LOOKUP and the GETFH
(since COMPOUND is not atomic). Even if we had the fsids for all of
the intermediate directories, we could have no way of knowing that
/this/is/the/path was the root of a new file system, since we don't
yet have its fsid.

In order to get the necessary information, let us re-send the chain
of LOOKUPs with GETFHs and GETATTRs to at least get the fsids so we
can be sure where the appropriate file system boundaries are. The
client could choose to get fs_locations at the same time but in most
cases the client will have a good guess as to where file system
boundaries are (because of where NFS4ERR_MOVED was, and was not,
received) making fetching of fs_locations unnecessary.

OP01: PUTROOTFH --> NFS_OK

- Current fh is root of pseudo-fs.

OP02: GETATTR(fsid) --> NFS_OK

   - Just for completeness. Normally, clients will know the fsid of
      the pseudo-fs as soon as they establish communication with a
      server.

   OP03: LOOKUP "this" --> NFS_OK

   OP04: GETATTR(fsid) --> NFS_OK

   - Get current fsid to see where file system boundaries are. The
      fsid will be that for the pseudo-fs in this example, so no
      boundary.

   OP05: GETFH --> NFS_OK

   - Current fh is for /this and is within pseudo-fs.

   OP06: LOOKUP "is" --> NFS_OK

   - Current fh is for /this/is and is within pseudo-fs.

   OP07: GETATTR(fsid) --> NFS_OK

   - Get current fsid to see where file system boundaries are. The
      fsid will be that for the pseudo-fs in this example, so no
      boundary.

   OP08: GETFH --> NFS_OK

   - Current fh is for /this/is and is within pseudo-fs.

   OP09: LOOKUP "the" --> NFS_OK

   - Current fh is for /this/is/the and is within pseudo-fs.

   OP10: GETATTR(fsid) --> NFS_OK

   - Get current fsid to see where file system boundaries are. The
      fsid will be that for the pseudo-fs in this example, so no
      boundary.

   OP11: GETFH --> NFS_OK

   - Current fh is for /this/is/the and is within pseudo-fs.

   OP12: LOOKUP "path" --> NFS_OK

- Current fh is for /this/is/the/path and is within a new, absent
  file system, but ...

- The client will never see the value of that fh.

  OP13: GETATTR(fsid, fs_locations) --> NFS_OK

- We are getting the fsid to know where the file system boundaries
  are. In this operation, the fsid will be different than that of
  the parent directory (which in turn was retrieved in OP10). Note
  that the fsid we are given will not necessarily be preserved at
  the new location. That fsid might be different, and in fact the
  fsid we have for this file system might be a valid fsid of a
  different file system on that new server.

- In this particular case, we are pretty sure anyway that what has
  moved is /this/is/the/path rather than /this/is/the since we have
  the fsid of the latter and it is that of the pseudo-fs, which
  presumably cannot move. However, in other examples, we might not
  have this kind of information to rely on (e.g., /this/is/the might
  be a non-pseudo file system separate from /this/is/the/path), so
  we need to have other reliable source information on the boundary
  of the file system that is moved. If, for example, the file
  system /this/is had moved, we would have a case of migration
  rather than referral, and once the boundaries of the migrated file
  system was clear we could fetch fs_locations.

- We are fetching fs_locations because the fact that we got an
  NFS4ERR_MOVED at this point means that it is most likely that this
  is a referral and we need the destination. Even if it is the case
  that /this/is/the is a file system that has migrated, we will
  still need the location information for that file system.

  OP14: GETFH --> NFS4ERR_MOVED

- Fails because current fh is in an absent file system at the start
  of the operation, and the specification makes no exception for
  GETFH. Note that this means the server will never send the client
  a filehandle from within an absent file system.

Given the above, the client knows where the root of the absent file
system is (/this/is/the/path) by noting where the change of fsid
occurred (between "the" and "path"). The fs_locations attribute also
gives the client the actual location of the absent file system, so
that the referral can proceed. The server gives the client the bare
minimum of information about the absent file system so that there
will be very little scope for problems of conflict between
information sent by the referring server and information of the file

   system's home. No filehandles and very few attributes are present on
   the referring server, and the client can treat those it receives as
   transient information with the function of enabling the referral.

8.7.2. Referral Example (READDIR)

   Another context in which a client may encounter referrals is when it
   does a READDIR on a directory in which some of the sub-directories
   are the roots of absent file systems.

   Suppose such a directory is read as follows:

   o PUTROOTFH

   o LOOKUP "this"

   o LOOKUP "is"

   o LOOKUP "the"

   o READDIR (fsid, size, time_modify, mounted_on_fileid)

   In this case, because rdattr_error is not requested, fs_locations is
   not requested, and some of the attributes cannot be provided, the
   result will be an NFS4ERR_MOVED error on the READDIR, with the
   detailed results as follows:

   o PUTROOTFH --> NFS_OK. The current fh is at the root of the
      pseudo-fs.

   o LOOKUP "this" --> NFS_OK. The current fh is for /this and is
      within the pseudo-fs.

   o LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is
      within the pseudo-fs.

   o LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and
      is within the pseudo-fs.

   o READDIR (fsid, size, time_modify, mounted_on_fileid) -->
      NFS4ERR_MOVED. Note that the same error would have been returned
      if /this/is/the had migrated, but it is returned because the
      directory contains the root of an absent file system.

   So now suppose that we re-send with rdattr_error:

o   PUTROOTFH

o   LOOKUP "this"

o   LOOKUP "is"

o   LOOKUP "the"

o   READDIR (rdattr_error, fsid, size, time_modify, mounted_on_fileid)

The results will be:

o   PUTROOTFH --> NFS_OK. The current fh is at the root of the
    pseudo-fs.

o   LOOKUP "this" --> NFS_OK. The current fh is for /this and is
    within the pseudo-fs.

o   LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is
    within the pseudo-fs.

o   LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and
    is within the pseudo-fs.

o   READDIR (rdattr_error, fsid, size, time_modify, mounted_on_fileid)
    --> NFS_OK. The attributes for directory entry with the component
    named "path" will only contain rdattr_error with the value
    NFS4ERR_MOVED, together with an fsid value and a value for
    mounted_on_fileid.

So suppose we do another READDIR to get fs_locations (although we
could have used a GETATTR directly, as in Section 8.7.1).

o   PUTROOTFH

o   LOOKUP "this"

o   LOOKUP "is"

o   LOOKUP "the"

o   READDIR (rdattr_error, fs_locations, mounted_on_fileid, fsid,
    size, time_modify)

The results would be:

o PUTROOTFH --> NFS_OK. The current fh is at the root of the
   pseudo-fs.

o LOOKUP "this" --> NFS_OK. The current fh is for /this and is
   within the pseudo-fs.

o LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is
   within the pseudo-fs.

o LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and
   is within the pseudo-fs.

o READDIR (rdattr_error, fs_locations, mounted_on_fileid, fsid,
   size, time_modify) --> NFS_OK. The attributes will be as shown
   below.

The attributes for the directory entry with the component named
"path" will only contain:

o rdattr_error (value: NFS_OK)

o fs_locations

o mounted_on_fileid (value: unique fileid within referring file
   system)

o fsid (value: unique value within referring server)

The attributes for entry "path" will not contain size or time_modify
because these attributes are not available within an absent file
system.

8.8. The Attribute fs_locations

The fs_locations attribute is structured in the following way:

```
struct fs_location4 {
        utf8str_cis             server<>;
        pathname4               rootpath;
};


struct fs_locations4 {
        pathname4       fs_root;
        fs_location4    locations<>;
};
```

The fs_location4 data type is used to represent the location of a
file system by providing a server name and the path to the root of
the file system within that server's namespace. When a set of
servers have corresponding file systems at the same path within their
namespaces, an array of server names may be provided. An entry in
the server array is a UTF-8 string and represents one of a
traditional DNS host name, IPv4 address, IPv6 address, or an zero-
length string. A zero-length string SHOULD be used to indicate the
current address being used for the RPC call. It is not a requirement
that all servers that share the same rootpath be listed in one
fs_location4 instance. The array of server names is provided for
convenience. Servers that share the same rootpath may also be listed
in separate fs_location4 entries in the fs_locations attribute.

The fs_locations4 data type and fs_locations attribute contain an
array of such locations. Since the namespace of each server may be
constructed differently, the "fs_root" field is provided. The path
represented by fs_root represents the location of the file system in
the current server's namespace, i.e., that of the server from which
the fs_locations attribute was obtained. The fs_root path is meant
to aid the client by clearly referencing the root of the file system
whose locations are being reported, no matter what object within the
current file system the current filehandle designates. The fs_root
is simply the pathname the client used to reach the object on the
current server (i.e., the object to which the fs_locations attribute
applies).

When the fs_locations attribute is interrogated and there are no
alternate file system locations, the server SHOULD return a zero-
length array of fs_location4 structures, together with a valid
fs_root.

As an example, suppose there is a replicated file system located at
two servers (servA and servB). At servA, the file system is located
at path /a/b/c. At, servB the file system is located at path /x/y/z.
If the client were to obtain the fs_locations value for the directory
at /a/b/c/d, it might not necessarily know that the file system's
root is located in servA's namespace at /a/b/c. When the client
switches to servB, it will need to determine that the directory it
first referenced at servA is now represented by the path /x/y/z/d on
servB. To facilitate this, the fs_locations attribute provided by
servA would have an fs_root value of /a/b/c and two entries in
fs_locations. One entry in fs_locations will be for itself (servA)
and the other will be for servB with a path of /x/y/z. With this
information, the client is able to substitute /x/y/z for the /a/b/c
at the beginning of its access path and construct /x/y/z/d to use for
the new server.

Note that: there is no requirement that the number of components in
each rootpath be the same; there is no relation between the number of
components in rootpath or fs_root, and none of the components in each
rootpath and fs_root have to be the same. In the above example, we
could have had a third element in the locations array, with server
equal to "servC", and rootpath equal to "/I/II", and a fourth element
in locations with server equal to "servD" and rootpath equal to
"/aleph/beth/gimel/daleth/he".

The relationship between fs_root to a rootpath is that the client
replaces the pathname indicated in fs_root for the current server for
the substitute indicated in rootpath for the new server.

For an example of a referred or migrated file system, suppose there
is a file system located at serv1. At serv1, the file system is
located at /az/buky/vedi/glagoli. The client finds that object at
glagoli has migrated (or is a referral). The client gets the
fs_locations attribute, which contains an fs_root of /az/buky/vedi/
glagoli, and one element in the locations array, with server equal to
serv2, and rootpath equal to /izhitsa/fita. The client replaces /az/
buky/vedi/glagoli with /izhitsa/fita, and uses the latter pathname on
serv2.

Thus, the server MUST return an fs_root that is equal to the path the
client used to reach the object to which the fs_locations attribute
applies. Otherwise, the client cannot determine the new path to use
on the new server.

8.8.1. Inferring Transition Modes

When fs_locations is used, information about the specific locations
should be assumed based on the following rules.

The following rules are general and apply irrespective of the
context.

o All listed file system instances should be considered as of the
   same handle class if and only if the current fh_expire_type
   attribute does not include the FH4_VOL_MIGRATION bit. Note that
   in the case of referral, filehandle issues do not apply since
   there can be no filehandles known within the current file system
   nor is there any access to the fh_expire_type attribute on the
   referring (absent) file system.

o All listed file system instances should be considered as of the
   same fileid class if and only if the fh_expire_type attribute
   indicates persistent filehandles and does not include the
   FH4_VOL_MIGRATION bit. Note that in the case of referral, fileid

issues do not apply since there can be no fileids known within the referring (absent) file system nor is there any access to the fh_expire_type attribute.

o All file system instances servers should be considered as of different change classes.

o All file system instances servers should be considered as of different readdir classes.

For other class assignments, handling of file system transitions depends on the reasons for the transition:

o When the transition is due to migration, that is, the client was directed to a new file system after receiving an NFS4ERR_MOVED error, the target should be treated as being of the same write-verifier class as the source.

o When the transition is due to failover to another replica, that is, the client selected another replica without receiving and NFS4ERR_MOVED error, the target should be treated as being of a different write-verifier class from the source.

The specific choices reflect typical implementation patterns for failover and controlled migration, respectively.

See Section 17 for a discussion on the recommendations for the security flavor to be used by any GETATTR operation that requests the "fs_locations" attribute.

9. File Locking and Share Reservations

Integrating locking into the NFS protocol necessarily causes it to be stateful. With the inclusion of share reservations the protocol becomes substantially more dependent on state than the traditional combination of NFS and NLM (Network Lock Manager) [xnfs]. There are three components to making this state manageable:

o clear division between client and server

o ability to reliably detect inconsistency in state between client and server

o simple and robust recovery mechanisms

In this model, the server owns the state information. The client requests changes in locks and the server responds with the changes

made. Non-client-initiated changes in locking state are infrequent.
The client receives prompt notification of such changes and can
adjust its view of the locking state to reflect the server's changes.

Individual pieces of state created by the server and passed to the
client at its request are represented by 128-bit stateids. These
stateids may represent a particular open file, a set of byte-range
locks held by a particular owner, or a recallable delegation of
privileges to access a file in particular ways or at a particular
location.

In all cases, there is a transition from the most general information
that represents a client as a whole to the eventual lightweight
stateid used for most client and server locking interactions. The
details of this transition will vary with the type of object but it
always starts with a client ID.

To support Win32 share reservations it is necessary to atomically
OPEN or CREATE files and apply the appropriate locks in the same
operation. Having a separate share/unshare operation would not allow
correct implementation of the Win32 OpenFile API. In order to
correctly implement share semantics, the previous NFS protocol
mechanisms used when a file is opened or created (LOOKUP, CREATE,
ACCESS) need to be replaced. The NFSv4 protocol has an OPEN
operation that subsumes the NFSv3 methodology of LOOKUP, CREATE, and
ACCESS. However, because many operations require a filehandle, the
traditional LOOKUP is preserved to map a file name to filehandle
without establishing state on the server. The policy of granting
access or modifying files is managed by the server based on the
client's state. These mechanisms can implement policy ranging from
advisory only locking to full mandatory locking.

9.1. Opens and Byte-Range Locks

It is assumed that manipulating a byte-range lock is rare when
compared to READ and WRITE operations. It is also assumed that
server restarts and network partitions are relatively rare.
Therefore it is important that the READ and WRITE operations have a
lightweight mechanism to indicate if they possess a held lock. A
byte-range lock request contains the heavyweight information required
to establish a lock and uniquely define the owner of the lock.

The following sections describe the transition from the heavy weight
information to the eventual stateid used for most client and server
locking and lease interactions.

9.1.1. **Client ID**

   For each LOCK request, the client must identify itself to the server.
   This is done in such a way as to allow for correct lock
   identification and crash recovery. A sequence of a SETCLIENTID
   operation followed by a SETCLIENTID_CONFIRM operation is required to
   establish the identification onto the server. Establishment of
   identification by a new incarnation of the client also has the effect
   of immediately breaking any leased state that a previous incarnation
   of the client might have had on the server, as opposed to forcing the
   new client incarnation to wait for the leases to expire. Breaking
   the lease state amounts to the server removing all lock, share
   reservation, and, where the server is not supporting the
   CLAIM_DELEGATE_PREV claim type, all delegation state associated with
   same client with the same identity. For discussion of delegation
   state recovery, see Section 10.2.1.

   Owners of opens and owners of byte-range locks are separate entities
   and remain separate even if the same opaque arrays are used to
   designate owners of each. The protocol distinguishes between open-
   owners (represented by open_owner4 structures) and lock-owners
   (represented by lock_owner4 structures).

   Both sorts of owners consist of a clientid and an opaque owner
   string. For each client, the set of distinct owner values used with
   that client constitutes the set of owners of that type, for the given
   client.

   Each open is associated with a specific open-owner while each byte-
   range lock is associated with a lock-owner and an open-owner, the
   latter being the open-owner associated with the open file under which
   the LOCK operation was done.

   Client identification is encapsulated in the following structure:

```
struct nfs_client_id4 {
        verifier4       verifier;
        opaque          id<NFS4_OPAQUE_LIMIT>;
};
```

   The first field, verifier is a client incarnation verifier that is
   used to detect client reboots. Only if the verifier is different
   from that which the server has previously recorded for the client (as
   identified by the second field of the structure, id) does the server
   start the process of canceling the client's leased state.

   The second field, id is a variable length string that uniquely
   defines the client.

There are several considerations for how the client generates the id
string:

o The string should be unique so that multiple clients do not
   present the same string. The consequences of two clients
   presenting the same string range from one client getting an error
   to one client having its leased state abruptly and unexpectedly
   canceled.

o The string should be selected so the subsequent incarnations
   (e.g., reboots) of the same client cause the client to present the
   same string. The implementor is cautioned against an approach
   that requires the string to be recorded in a local file because
   this precludes the use of the implementation in an environment
   where there is no local disk and all file access is from an NFSv4
   server.

o The string should be different for each server network address
   that the client accesses, rather than common to all server network
   addresses. The reason is that it may not be possible for the
   client to tell if the same server is listening on multiple network
   addresses. If the client issues SETCLIENTID with the same id
   string to each network address of such a server, the server will
   think it is the same client, and each successive SETCLIENTID will
   cause the server to begin the process of removing the client's
   previous leased state.

o The algorithm for generating the string should not assume that the
   client's network address won't change. This includes changes
   between client incarnations and even changes while the client is
   stilling running in its current incarnation. This means that if
   the client includes just the client's and server's network address
   in the id string, there is a real risk, after the client gives up
   the network address, that another client, using a similar
   algorithm for generating the id string, will generate a
   conflicting id string.

Given the above considerations, an example of a well generated id
string is one that includes:

o The server's network address.

o The client's network address.

o For a user level NFSv4 client, it should contain additional
   information to distinguish the client from other user level
   clients running on the same host, such as an universally unique
   identifier (UUID).

   o Additional information that tends to be unique, such as one or
      more of:

      * The client machine's serial number (for privacy reasons, it is
         best to perform some one way function on the serial number).

      * A MAC address.

      * The timestamp of when the NFSv4 software was first installed on
         the client (though this is subject to the previously mentioned
         caution about using information that is stored in a file,
         because the file might only be accessible over NFSv4).

      * A true random number. However since this number ought to be
         the same between client incarnations, this shares the same
         problem as that of the using the timestamp of the software
         installation.

   As a security measure, the server MUST NOT cancel a client's leased
   state if the principal that established the state for a given id
   string is not the same as the principal issuing the SETCLIENTID.

   Note that SETCLIENTID and SETCLIENTID_CONFIRM has a secondary purpose
   of establishing the information the server needs to make callbacks to
   the client for purpose of supporting delegations. It is permitted to
   change this information via SETCLIENTID and SETCLIENTID_CONFIRM
   within the same incarnation of the client without removing the
   client's leased state.

   Once a SETCLIENTID and SETCLIENTID_CONFIRM sequence has successfully
   completed, the client uses the shorthand client identifier, of type
   clientid4, instead of the longer and less compact nfs_client_id4
   structure. This shorthand client identifier (a client ID) is
   assigned by the server and should be chosen so that it will not
   conflict with a client ID previously assigned by the server. This
   applies across server restarts or reboots. When a client ID is
   presented to a server and that client ID is not recognized, as would
   happen after a server reboot, the server will reject the request with
   the error NFS4ERR_STALE_CLIENTID. When this happens, the client must
   obtain a new client ID by use of the SETCLIENTID operation and then
   proceed to any other necessary recovery for the server reboot case
   (See Section 9.6.2).

   The client must also employ the SETCLIENTID operation when it
   receives a NFS4ERR_STALE_STATEID error using a stateid derived from
   its current client ID, since this also indicates a server reboot
   which has invalidated the existing client ID (see Section 9.6.2 for
   details).

See the detailed descriptions of SETCLIENTID and SETCLIENTID_CONFIRM
for a complete specification of the operations.

9.1.2. **Server Release of Client ID**

If the server determines that the client holds no associated state
for its client ID, the server may choose to release the client ID.
The server may make this choice for an inactive client so that
resources are not consumed by those intermittently active clients.
If the client contacts the server after this release, the server must
ensure the client receives the appropriate error so that it will use
the SETCLIENTID/SETCLIENTID_CONFIRM sequence to establish a new
identity. It should be clear that the server must be very hesitant
to release a client ID since the resulting work on the client to
recover from such an event will be the same burden as if the server
had failed and restarted. Typically a server would not release a
client ID unless there had been no activity from that client for many
minutes.

Note that if the id string in a SETCLIENTID request is properly
constructed, and if the client takes care to use the same principal
for each successive use of SETCLIENTID, then, barring an active
denial of service attack, NFS4ERR_CLID_INUSE should never be
returned.

However, client bugs, server bugs, or perhaps a deliberate change of
the principal owner of the id string (such as the case of a client
that changes security flavors, and under the new flavor, there is no
mapping to the previous owner) will in rare cases result in
NFS4ERR_CLID_INUSE.

In that event, when the server gets a SETCLIENTID for a client ID
that currently has no state, or it has state, but the lease has
expired, rather than returning NFS4ERR_CLID_INUSE, the server MUST
allow the SETCLIENTID, and confirm the new client ID if followed by
the appropriate SETCLIENTID_CONFIRM.

9.1.3. **Stateid Definition**

When the server grants a lock of any type (including opens, byte-
range locks, and delegations), it responds with a unique stateid that
represents a set of locks (often a single lock) for the same file, of
the same type, and sharing the same ownership characteristics. Thus,
opens of the same file by different open-owners each have an
identifying stateid. Similarly, each set of byte-range locks on a
file owned by a specific lock-owner has its own identifying stateid.
Delegations also have associated stateids by which they may be
referenced. The stateid is used as a shorthand reference to a lock

or set of locks, and given a stateid, the server can determine the
associated state-owner or state-owners (in the case of an open-owner/
lock-owner pair) and the associated filehandle. When stateids are
used, the current filehandle must be the one associated with that
stateid.

All stateids associated with a given client ID are associated with a
common lease that represents the claim of those stateids and the
objects they represent to be maintained by the server. See
Section 9.5 for a discussion of the lease.

Each stateid must be unique to the server. Many operations take a
stateid as an argument but not a clientid, so the server must be able
to infer the client from the stateid.

9.1.3.1. Stateid Types

With the exception of special stateids (see Section 9.1.3.3), each
stateid represents locking objects of one of a set of types defined
by the NFSv4 protocol. Note that in all these cases, where we speak
of guarantee, it is understood there are situations such as a client
restart, or lock revocation, that allow the guarantee to be voided.

o Stateids may represent opens of files.

   Each stateid in this case represents the OPEN state for a given
   client ID/open-owner/filehandle triple. Such stateids are subject
   to change (with consequent incrementing of the stateid's seqid) in
   response to OPENs that result in upgrade and OPEN_DOWNGRADE
   operations.

o Stateids may represent sets of byte-range locks.

   All locks held on a particular file by a particular owner and all
   gotten under the aegis of a particular open file are associated
   with a single stateid with the seqid being incremented whenever
   LOCK and LOCKU operations affect that set of locks.

o Stateids may represent file delegations, which are recallable
   guarantees by the server to the client, that other clients will
   not reference, or will not modify a particular file, until the
   delegation is returned.

   A stateid represents a single delegation held by a client for a
   particular filehandle.

9.1.3.2. Stateid Structure

Stateids are divided into two fields, a 96-bit "other" field
identifying the specific set of locks and a 32-bit "seqid" sequence
value. Except in the case of special stateids (see Section 9.1.3.3),
a particular value of the "other" field denotes a set of locks of the
same type (for example, byte-range locks, opens, or delegations), for
a specific file or directory, and sharing the same ownership
characteristics. The seqid designates a specific instance of such a
set of locks, and is incremented to indicate changes in such a set of
locks, either by the addition or deletion of locks from the set, a
change in the byte-range they apply to, or an upgrade or downgrade in
the type of one or more locks.

When such a set of locks is first created, the server SHOULD return a
stateid with seqid value of one. On subsequent operations that
modify the set of locks, the server is required to increment the
"seqid" field by one whenever it returns a stateid for the same
state-owner/file/type combination and there is some change in the set
of locks actually designated. In this case, the server will return a
stateid with an "other" field the same as previously used for that
state-owner/file/type combination, with an incremented "seqid" field.
This pattern continues until the seqid is incremented past
NFS4_UINT32_MAX, and one (not zero) SHOULD be the next seqid value.
The purpose of the incrementing of the seqid is to allow the server
to communicate to the client the order in which operations that
modified locking state associated with a stateid have been processed.

In making comparisons between seqids, both by the client in
determining the order of operations and by the server in determining
whether the NFS4ERR_OLD_STATEID is to be returned, the possibility of
the seqid being swapped around past the NFS4_UINT32_MAX value needs
to be taken into account.

9.1.3.3. Special Stateids

Stateid values whose "other" field is either all zeros or all ones
are reserved. They may not be assigned by the server but have
special meanings defined by the protocol. The particular meaning
depends on whether the "other" field is all zeros or all ones and the
specific value of the "seqid" field.

The following combinations of "other" and "seqid" are defined in
NFSv4:

o When "other" and "seqid" are both zero, the stateid is treated as
   a special anonymous stateid, which can be used in READ, WRITE, and
   SETATTR requests to indicate the absence of any open state

associated with the request. When an anonymous stateid value is
used, and an existing open denies the form of access requested,
then access will be denied to the request.

o When "other" and "seqid" are both all ones, the stateid is a
   special READ bypass stateid. When this value is used in WRITE or
   SETATTR, it is treated like the anonymous value. When used in
   READ, the server MAY grant access, even if access would normally
   be denied to READ requests.

If a stateid value is used which has all zero or all ones in the
"other" field, but does not match one of the cases above, the server
MUST return the error NFS4ERR_BAD_STATEID.

Special stateids, unlike other stateids, are not associated with
individual client IDs or filehandles and can be used with all valid
client IDs and filehandles.

9.1.3.4. Stateid Lifetime and Validation

Stateids must remain valid until either a client restart or a server
restart or until the client returns all of the locks associated with
the stateid by means of an operation such as CLOSE or DELEGRETURN.
If the locks are lost due to revocation as long as the client ID is
valid, the stateid remains a valid designation of that revoked state.
Stateids associated with byte-range locks are an exception. They
remain valid even if a LOCKU frees all remaining locks, so long as
the open file with which they are associated remains open.

It should be noted that there are situations in which the client's
locks become invalid, without the client requesting they be returned.
These include lease expiration and a number of forms of lock
revocation within the lease period. It is important to note that in
these situations, the stateid remains valid and the client can use it
to determine the disposition of the associated lost locks.

An "other" value must never be reused for a different purpose (i.e.
different filehandle, owner, or type of locks) within the context of
a single client ID. A server may retain the "other" value for the
same purpose beyond the point where it may otherwise be freed but if
it does so, it must maintain "seqid" continuity with previous values.

One mechanism that may be used to satisfy the requirement that the
server recognize invalid and out-of-date stateids is for the server
to divide the "other" field of the stateid into two fields.

o An index into a table of locking-state structures.

   o A generation number which is incremented on each allocation of a
      table entry for a particular use.

   And then store in each table entry,

   o The client ID with which the stateid is associated.

   o The current generation number for the (at most one) valid stateid
      sharing this index value.

   o The filehandle of the file on which the locks are taken.

   o An indication of the type of stateid (open, byte-range lock, file
      delegation).

   o The last "seqid" value returned corresponding to the current
      "other" value.

   o An indication of the current status of the locks associated with
      this stateid. In particular, whether these have been revoked and
      if so, for what reason.

   With this information, an incoming stateid can be validated and the
   appropriate error returned when necessary. Special and non-special
   stateids are handled separately. (See Section 9.1.3.3 for a
   discussion of special stateids.)

   When a stateid is being tested, and the "other" field is all zeros or
   all ones, a check that the "other" and "seqid" fields match a defined
   combination for a special stateid is done and the results determined
   as follows:

   o If the "other" and "seqid" fields do not match a defined
      combination associated with a special stateid, the error
      NFS4ERR_BAD_STATEID is returned.

   o If the combination is valid in general but is not appropriate to
      the context in which the stateid is used (e.g., an all-zero
      stateid is used when an open stateid is required in a LOCK
      operation), the error NFS4ERR_BAD_STATEID is also returned.

   o Otherwise, the check is completed and the special stateid is
      accepted as valid.

   When a stateid is being tested, and the "other" field is neither all
   zeros or all ones, the following procedure could be used to validate
   an incoming stateid and return an appropriate error, when necessary,
   assuming that the "other" field would be divided into a table index

and an entry generation.

o If the table index field is outside the range of the associated
   table, return NFS4ERR_BAD_STATEID.

o If the selected table entry is of a different generation than that
   specified in the incoming stateid, return NFS4ERR_BAD_STATEID.

o If the selected table entry does not match the current filehandle,
   return NFS4ERR_BAD_STATEID.

o If the stateid represents revoked state or state lost as a result
   of lease expiration, then return NFS4ERR_EXPIRED,
   NFS4ERR_BAD_STATEID, or NFS4ERR_ADMIN_REVOKED, as appropriate.

o If the stateid type is not valid for the context in which the
   stateid appears, return NFS4ERR_BAD_STATEID. Note that a stateid
   may be valid in general, but be invalid for a particular
   operation, as, for example, when a stateid which doesn't represent
   byte-range locks is passed to the non-from_open case of LOCK or to
   LOCKU, or when a stateid which does not represent an open is
   passed to CLOSE or OPEN_DOWNGRADE. In such cases, the server MUST
   return NFS4ERR_BAD_STATEID.

o If the "seqid" field is not zero, and it is greater than the
   current sequence value corresponding the current "other" field,
   return NFS4ERR_BAD_STATEID.

o If the "seqid" field is less than the current sequence value
   corresponding the current "other" field, return
   NFS4ERR_OLD_STATEID.

o Otherwise, the stateid is valid and the table entry should contain
   any additional information about the type of stateid and
   information associated with that particular type of stateid, such
   as the associated set of locks, such as open-owner and lock-owner
   information, as well as information on the specific locks, such as
   open modes and byte ranges.

9.1.3.5. Stateid Use for I/O Operations

Clients performing Input/Output (I/O) operations need to select an
appropriate stateid based on the locks (including opens and
delegations) held by the client and the various types of state-owners
sending the I/O requests. SETATTR operations that change the file
size are treated like I/O operations in this regard.

The following rules, applied in order of decreasing priority, govern

the selection of the appropriate stateid. In following these rules,
the client will only consider locks of which it has actually received
notification by an appropriate operation response or callback.

o If the client holds a delegation for the file in question, the
   delegation stateid SHOULD be used.

o Otherwise, if the entity corresponding to the lock-owner (e.g., a
   process) sending the I/O has a byte-range lock stateid for the
   associated open file, then the byte-range lock stateid for that
   lock-owner and open file SHOULD be used.

o If there is no byte-range lock stateid, then the OPEN stateid for
   the current open-owner, and that OPEN stateid for the open file in
   question SHOULD be used.

o Finally, if none of the above apply, then a special stateid SHOULD
   be used.

Ignoring these rules may result in situations in which the server
does not have information necessary to properly process the request.
For example, when mandatory byte-range locks are in effect, if the
stateid does not indicate the proper lock-owner, via a lock stateid,
a request might be avoidably rejected.

The server however should not try to enforce these ordering rules and
should use whatever information is available to properly process I/O
requests. In particular, when a client has a delegation for a given
file, it SHOULD take note of this fact in processing a request, even
if it is sent with a special stateid.

9.1.3.6. Stateid Use for SETATTR Operations

In the case of SETATTR operations, a stateid is present. In cases
other than those that set the file size, the client may send either a
special stateid or, when a delegation is held for the file in
question, a delegation stateid. While the server SHOULD validate the
stateid and may use the stateid to optimize the determination as to
whether a delegation is held, it SHOULD note the presence of a
delegation even when a special stateid is sent, and MUST accept a
valid delegation stateid when sent.

9.1.4. lock-owner

When requesting a lock, the client must present to the server the
client ID and an identifier for the owner of the requested lock.
These two fields are referred to as the lock-owner and the definition
of those fields are:

o A client ID returned by the server as part of the client's use of
   the SETCLIENTID operation.

o A variable length opaque array used to uniquely define the owner
   of a lock managed by the client.

   This may be a thread id, process id, or other unique value.

When the server grants the lock, it responds with a unique stateid.
The stateid is used as a shorthand reference to the lock-owner, since
the server will be maintaining the correspondence between them.

9.1.5. Use of the Stateid and Locking

All READ, WRITE and SETATTR operations contain a stateid. For the
purposes of this section, SETATTR operations which change the size
attribute of a file are treated as if they are writing the area
between the old and new size (i.e., the range truncated or added to
the file by means of the SETATTR), even where SETATTR is not
explicitly mentioned in the text. The stateid passed to one of these
operations must be one that represents an OPEN (e.g., via the open-
owner), a set of byte-range locks, or a delegation, or it may be a
special stateid representing anonymous access or the special bypass
stateid.

If the state-owner performs a READ or WRITE in a situation in which
it has established a lock or share reservation on the server (any
OPEN constitutes a share reservation) the stateid (previously
returned by the server) must be used to indicate what locks,
including both byte-range locks and share reservations, are held by
the state-owner. If no state is established by the client, either
byte-range lock or share reservation, a stateid of all bits 0 is
used. Regardless whether a stateid of all bits 0, or a stateid
returned by the server is used, if there is a conflicting share
reservation or mandatory byte-range lock held on the file, the server
MUST refuse to service the READ or WRITE operation.

Share reservations are established by OPEN operations and by their
nature are mandatory in that when the OPEN denies READ or WRITE
operations, that denial results in such operations being rejected
with error NFS4ERR_LOCKED. Byte-range locks may be implemented by
the server as either mandatory or advisory, or the choice of
mandatory or advisory behavior may be determined by the server on the
basis of the file being accessed (for example, some UNIX-based
servers support a "mandatory lock bit" on the mode attribute such
that if set, byte-range locks are required on the file before I/O is
possible). When byte-range locks are advisory, they only prevent the
granting of conflicting lock requests and have no effect on READs or

WRITEs. Mandatory byte-range locks, however, prevent conflicting I/O
operations. When they are attempted, they are rejected with
NFS4ERR_LOCKED. When the client gets NFS4ERR_LOCKED on a file it
knows it has the proper share reservation for, it will need to issue
a LOCK request on the region of the file that includes the region the
I/O was to be performed on, with an appropriate locktype (i.e.,
READ*_LT for a READ operation, WRITE*_LT for a WRITE operation).

With NFSv3, there was no notion of a stateid so there was no way to
tell if the application process of the client sending the READ or
WRITE operation had also acquired the appropriate byte-range lock on
the file. Thus there was no way to implement mandatory locking.
With the stateid construct, this barrier has been removed.

Note that for UNIX environments that support mandatory file locking,
the distinction between advisory and mandatory locking is subtle. In
fact, advisory and mandatory byte-range locks are exactly the same in
so far as the APIs and requirements on implementation. If the
mandatory lock attribute is set on the file, the server checks to see
if the lock-owner has an appropriate shared (read) or exclusive
(write) byte-range lock on the region it wishes to read or write to.
If there is no appropriate lock, the server checks if there is a
conflicting lock (which can be done by attempting to acquire the
conflicting lock on the behalf of the lock-owner, and if successful,
release the lock after the READ or WRITE is done), and if there is,
the server returns NFS4ERR_LOCKED.

For Windows environments, there are no advisory byte-range locks, so
the server always checks for byte-range locks during I/O requests.

Thus, the NFSv4 LOCK operation does not need to distinguish between
advisory and mandatory byte-range locks. It is the NFS version 4
server's processing of the READ and WRITE operations that introduces
the distinction.

Every stateid other than the special stateid values noted in this
section, whether returned by an OPEN-type operation (i.e., OPEN,
OPEN_DOWNGRADE), or by a LOCK-type operation (i.e., LOCK or LOCKU),
defines an access mode for the file (i.e., READ, WRITE, or READ-
WRITE) as established by the original OPEN which began the stateid
sequence, and as modified by subsequent OPENs and OPEN_DOWNGRADEs
within that stateid sequence. When a READ, WRITE, or SETATTR which
specifies the size attribute, is done, the operation is subject to
checking against the access mode to verify that the operation is
appropriate given the OPEN with which the operation is associated.

In the case of WRITE-type operations (i.e., WRITEs and SETATTRs which
set size), the server must verify that the access mode allows writing

and return an NFS4ERR_OPENMODE error if it does not. In the case, of
READ, the server may perform the corresponding check on the access
mode, or it may choose to allow READ on opens for WRITE only, to
accommodate clients whose write implementation may unavoidably do
reads (e.g., due to buffer cache constraints). However, even if
READs are allowed in these circumstances, the server MUST still check
for locks that conflict with the READ (e.g., another open specifying
denial of READs). Note that a server which does enforce the access
mode check on READs need not explicitly check for conflicting share
reservations since the existence of OPEN for read access guarantees
that no conflicting share reservation can exist.

A stateid of all bits 1 (one) MAY allow READ operations to bypass
locking checks at the server. However, WRITE operations with a
stateid with bits all 1 (one) MUST NOT bypass locking checks and are
treated exactly the same as if a stateid of all bits 0 were used.

A lock may not be granted while a READ or WRITE operation using one
of the special stateids is being performed and the range of the lock
request conflicts with the range of the READ or WRITE operation. For
the purposes of this paragraph, a conflict occurs when a shared lock
is requested and a WRITE operation is being performed, or an
exclusive lock is requested and either a READ or a WRITE operation is
being performed. A SETATTR that sets size is treated similarly to a
WRITE as discussed above.

9.1.6. Sequencing of Lock Requests

Locking is different than most NFS operations as it requires "at-
most-one" semantics that are not provided by ONC RPC. ONC RPC over a
reliable transport is not sufficient because a sequence of locking
requests may span multiple TCP connections. In the face of
retransmission or reordering, lock or unlock requests must have a
well defined and consistent behavior. To accomplish this, each lock
request contains a sequence number that is a consecutively increasing
integer. Different state-owners have different sequences. The
server maintains the last sequence number (L) received and the
response that was returned. The server SHOULD assign a seqid value
of one for the first request issued for any given state-owner.

Note that for requests that contain a sequence number, for each
state-owner, there should be no more than one outstanding request.

If a request (r) with a previous sequence number (r < L) is received,
it is rejected with the return of error NFS4ERR_BAD_SEQID. Given a
properly-functioning client, the response to (r) must have been
received before the last request (L) was sent. If a duplicate of
last request (r == L) is received, the stored response is returned.

If a request beyond the next sequence (r == L + 2) is received, it is
rejected with the return of error NFS4ERR_BAD_SEQID. Sequence
history is reinitialized whenever the SETCLIENTID/SETCLIENTID_CONFIRM
sequence changes the client verifier.

Since the sequence number is represented with an unsigned 32-bit
integer, the arithmetic involved with the sequence number is mod
2^32. Note that when the seqid wraps, it SHOULD bypass zero and use
one as the next seqid value. For an example of modulo arithmetic
involving sequence numbers see [RFC0793].

It is critical the server maintain the last response sent to the
client to provide a more reliable cache of duplicate non-idempotent
requests than that of the traditional cache described in [Chet]. The
traditional duplicate request cache uses a least recently used
algorithm for removing unneeded requests. However, the last lock
request and response on a given state-owner must be cached as long as
the lock state exists on the server.

The client MUST monotonically increment the sequence number for the
CLOSE, LOCK, LOCKU, OPEN, OPEN_CONFIRM, and OPEN_DOWNGRADE
operations. This is true even in the event that the previous
operation that used the sequence number received an error. The only
exception to this rule is if the previous operation received one of
the following errors: NFS4ERR_STALE_CLIENTID, NFS4ERR_STALE_STATEID,
NFS4ERR_BAD_STATEID, NFS4ERR_BAD_SEQID, NFS4ERR_BADXDR,
NFS4ERR_RESOURCE, NFS4ERR_NOFILEHANDLE, or NFS4ERR_MOVED.

9.1.7. Recovery from Replayed Requests

As described above, the sequence number is per state-owner. As long
as the server maintains the last sequence number received and follows
the methods described above, there are no risks of a Byzantine router
re-sending old requests. The server need only maintain the (state-
owner, sequence number) state as long as there are open files or
closed files with locks outstanding.

LOCK, LOCKU, OPEN, OPEN_DOWNGRADE, and CLOSE each contain a sequence
number and therefore the risk of the replay of these operations
resulting in undesired effects is non-existent while the server
maintains the state-owner state.

9.1.8. Interactions of multiple sequence values

Some Operations may have multiple sources of data for request
sequence checking and retransmission determination. Some Operations
have multiple sequence values associated with multiple types of
state-owners. In addition, such Operations may also have a stateid

   with its own seqid value, that will be checked for validity.

   As noted above, there may be multiple sequence values to check. The
   following rules should be followed by the server in processing these
   multiple sequence values within a single operation.

   o When a sequence value associated with a state-owner is unavailable
      for checking because the state-owner is unknown to the server, it
      takes no part in the comparison.

   o When any of the state-owner sequence values are invalid,
      NFS4ERR_BAD_SEQID is returned. When a stateid sequence is
      checked, NFS4ERR_BAD_STATEID, or NFS4ERR_OLD_STATEID is returned
      as appropriate, but NFS4ERR_BAD_SEQID has priority.

   o When any one of the sequence values matches a previous request,
      for a state-owner, it is treated as a retransmission and not re-
      executed. When the type of the operation does not match that
      originally used, NFS4ERR_BAD_SEQID is returned. When the server
      can determine that the request differs from the original it may
      return NFS4ERR_BAD_SEQID.

   o When multiple of the sequence values match previous operations,
      but the operations are not the same, NFS4ERR_BAD_SEQID is
      returned.

   o When there are no available sequence values available for
      comparison and the operation is an OPEN, the server indicates to
      the client that an OPEN_CONFIRM is required, unless it can
      conclusively determine that confirmation is not required (e.g., by
      knowing that no open-owner state has ever been released for the
      current clientid).

9.1.9. Releasing state-owner State

   When a particular state-owner no longer holds open or file locking
   state at the server, the server may choose to release the sequence
   number state associated with the state-owner. The server may make
   this choice based on lease expiration, for the reclamation of server
   memory, or other implementation specific details. Note that when
   this is done, a retransmitted request, normally identified by a
   matching state-owner sequence may not be correctly recognized, so
   that the client will not receive the original response that it would
   have if the state-owner state was not released.

   If the server were able to be sure that a given state-owner would
   never again be used by a client, such an issue could not arise. Even
   when the state-owner state is released and the client subsequently

uses that state-owner, retransmitted requests will be detected as
invalid and the request not executed, although the client may have a
recovery path that is more complicated than simply getting the
original response back transparently.

In any event, the server is able to safely release state-owner state
(in the sense that retransmitted requests will not be erroneously
acted upon) when the state-owner no currently being utilized by the
client (i.e., there are no open files associated with an open-owner
and no lock stateids associated with a lock-owner). The server may
choose to hold the state-owner state in order to simplify the
recovery path, in the case in which retransmissions of currently
active requests are received. However, the period it chooses to hold
this state is implementation specific.

In the case that a LOCK, LOCKU, OPEN_DOWNGRADE, or CLOSE is
retransmitted after the server has previously released the state-
owner state, the server will find that the state-owner has no files
open and an error will be returned to the client. If the state-owner
does have a file open, the stateid will not match and again an error
is returned to the client.

9.1.10. Use of Open Confirmation

In the case that an OPEN is retransmitted and the open-owner is being
used for the first time or the open-owner state has been previously
released by the server, the use of the OPEN_CONFIRM operation will
prevent incorrect behavior. When the server observes the use of the
open-owner for the first time, it will direct the client to perform
the OPEN_CONFIRM for the corresponding OPEN. This sequence
establishes the use of a open-owner and associated sequence number.
Since the OPEN_CONFIRM sequence connects a new open-owner on the
server with an existing open-owner on a client, the sequence number
may have any value. The OPEN_CONFIRM step assures the server that
the value received is the correct one. (see Section 15.20 for further
details.)

There are a number of situations in which the requirement to confirm
an OPEN would pose difficulties for the client and server, in that
they would be prevented from acting in a timely fashion on
information received, because that information would be provisional,
subject to deletion upon non-confirmation. Fortunately, these are
situations in which the server can avoid the need for confirmation
when responding to open requests. The two constraints are:

o The server must not bestow a delegation for any open which would
   require confirmation.

o The server MUST NOT require confirmation on a reclaim-type open
   (i.e., one specifying claim type CLAIM_PREVIOUS or
   CLAIM_DELEGATE_PREV).

These constraints are related in that reclaim-type opens are the only
ones in which the server may be required to send a delegation. For
CLAIM_NULL, sending the delegation is optional while for
CLAIM_DELEGATE_CUR, no delegation is sent.

Delegations being sent with an open requiring confirmation are
troublesome because recovering from non-confirmation adds undue
complexity to the protocol while requiring confirmation on reclaim-
type opens poses difficulties in that the inability to resolve the
status of the reclaim until lease expiration may make it difficult to
have timely determination of the set of locks being reclaimed (since
the grace period may expire).

Requiring open confirmation on reclaim-type opens is avoidable
because of the nature of the environments in which such opens are
done. For CLAIM_PREVIOUS opens, this is immediately after server
reboot, so there should be no time for open-owners to be created,
found to be unused, and recycled. For CLAIM_DELEGATE_PREV opens, we
are dealing with either a client reboot situation or a network
partition resulting in deletion of lease state (and returning
NFS4ERR_EXPIRED). A server which supports delegations can be sure
that no open-owners for that client have been recycled since client
initialization or deletion of lease state and thus can ensure that
confirmation will not be required.

9.2. Lock Ranges

The protocol allows a lock owner to request a lock with a byte range
and then either upgrade or unlock a sub-range of the initial lock.
It is expected that this will be an uncommon type of request. In any
case, servers or server file systems may not be able to support sub-
range lock semantics. In the event that a server receives a locking
request that represents a sub-range of current locking state for the
lock owner, the server is allowed to return the error
NFS4ERR_LOCK_RANGE to signify that it does not support sub-range lock
operations. Therefore, the client should be prepared to receive this
error and, if appropriate, report the error to the requesting
application.

The client is discouraged from combining multiple independent locking
ranges that happen to be adjacent into a single request since the
server may not support sub-range requests and for reasons related to
the recovery of file locking state in the event of server failure.
As discussed in the Section 9.6.2 below, the server may employ

certain optimizations during recovery that work effectively only when
the client's behavior during lock recovery is similar to the client's
locking behavior prior to server failure.

9.3. Upgrading and Downgrading Locks

If a client has a write lock on a record, it can request an atomic
downgrade of the lock to a read lock via the LOCK request, by setting
the type to READ_LT. If the server supports atomic downgrade, the
request will succeed. If not, it will return NFS4ERR_LOCK_NOTSUPP.
The client should be prepared to receive this error, and if
appropriate, report the error to the requesting application.

If a client has a read lock on a record, it can request an atomic
upgrade of the lock to a write lock via the LOCK request by setting
the type to WRITE_LT or WRITEW_LT. If the server does not support
atomic upgrade, it will return NFS4ERR_LOCK_NOTSUPP. If the upgrade
can be achieved without an existing conflict, the request will
succeed. Otherwise, the server will return either NFS4ERR_DENIED or
NFS4ERR_DEADLOCK. The error NFS4ERR_DEADLOCK is returned if the
client issued the LOCK request with the type set to WRITEW_LT and the
server has detected a deadlock. The client should be prepared to
receive such errors and if appropriate, report the error to the
requesting application.

9.4. Blocking Locks

Some clients require the support of blocking locks. The NFS version
4 protocol must not rely on a callback mechanism and therefore is
unable to notify a client when a previously denied lock has been
granted. Clients have no choice but to continually poll for the
lock. This presents a fairness problem. Two new lock types are
added, READW and WRITEW, and are used to indicate to the server that
the client is requesting a blocking lock. The server should maintain
an ordered list of pending blocking locks. When the conflicting lock
is released, the server may wait the lease period for the first
waiting client to re-request the lock. After the lease period
expires the next waiting client request is allowed the lock. Clients
are required to poll at an interval sufficiently small that it is
likely to acquire the lock in a timely manner. The server is not
required to maintain a list of pending blocked locks as it is not
used to provide correct operation but only to increase fairness.
Because of the unordered nature of crash recovery, storing of lock
state to stable storage would be required to guarantee ordered
granting of blocking locks.

Servers may also note the lock types and delay returning denial of
the request to allow extra time for a conflicting lock to be

released, allowing a successful return. In this way, clients can
avoid the burden of needlessly frequent polling for blocking locks.
The server should take care in the length of delay in the event the
client retransmits the request.

If a server receives a blocking lock request, denies it, and then
later receives a nonblocking request for the same lock, which is also
denied, then it should remove the lock in question from its list of
pending blocking locks. Clients should use such a nonblocking
request to indicate to the server that this is the last time they
intend to poll for the lock, as may happen when the process
requesting the lock is interrupted. This is a courtesy to the
server, to prevent it from unnecessarily waiting a lease period
before granting other lock requests. However, clients are not
required to perform this courtesy, and servers must not depend on
them doing so. Also, clients must be prepared for the possibility
that this final locking request will be accepted.

9.5. Lease Renewal

The purpose of a lease is to allow a server to remove stale locks
that are held by a client that has crashed or is otherwise
unreachable. It is not a mechanism for cache consistency and lease
renewals may not be denied if the lease interval has not expired.

The client can implicitly provide a positive indication that it is
still active and that the associated state held at the server, for
the client, is still valid. Any operation made with a valid clientid
(DELEGPURGE, LOCK, LOCKT, OPEN, RELEASE_LOCKOWNER, or RENEW) or a
valid stateid (CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, OPEN_CONFIRM,
OPEN_DOWNGRADE, READ, SETATTR, or WRITE) informs the server to renew
all of the leases for that client (i.e., all those sharing a given
client ID). In the latter case, the stateid must not be one of the
special stateids consisting of all bits 0 or all bits 1.

Note that if the client had restarted or rebooted, the client would
not be making these requests without issuing the SETCLIENTID/
SETCLIENTID_CONFIRM sequence. The use of the SETCLIENTID/
SETCLIENTID_CONFIRM sequence (one that changes the client verifier)
notifies the server to drop the locking state associated with the
client. SETCLIENTID/SETCLIENTID_CONFIRM never renews a lease.

If the server has rebooted, the stateids (NFS4ERR_STALE_STATEID
error) or the client ID (NFS4ERR_STALE_CLIENTID error) will not be
valid hence preventing spurious renewals.

This approach allows for low overhead lease renewal which scales
well. In the typical case no extra RPC calls are required for lease

renewal and in the worst case one RPC is required every lease period
(i.e., a RENEW operation). The number of locks held by the client is
not a factor since all state for the client is involved with the
lease renewal action.

Since all operations that create a new lease also renew existing
leases, the server must maintain a common lease expiration time for
all valid leases for a given client. This lease time can then be
easily updated upon implicit lease renewal actions.

9.6. Crash Recovery

The important requirement in crash recovery is that both the client
and the server know when the other has failed. Additionally, it is
required that a client sees a consistent view of data across server
restarts or reboots. All READ and WRITE operations that may have
been queued within the client or network buffers must wait until the
client has successfully recovered the locks protecting the READ and
WRITE operations.

9.6.1. Client Failure and Recovery

In the event that a client fails, the server may recover the client's
locks when the associated leases have expired. Conflicting locks
from another client may only be granted after this lease expiration.
If the client is able to restart or reinitialize within the lease
period the client may be forced to wait the remainder of the lease
period before obtaining new locks.

To minimize client delay upon restart, open and lock requests are
associated with an instance of the client by a client supplied
verifier. This verifier is part of the initial SETCLIENTID call made
by the client. The server returns a client ID as a result of the
SETCLIENTID operation. The client then confirms the use of the
client ID with SETCLIENTID_CONFIRM. The client ID in combination
with an opaque owner field is then used by the client to identify the
open owner for OPEN. This chain of associations is then used to
identify all locks for a particular client.

Since the verifier will be changed by the client upon each
initialization, the server can compare a new verifier to the verifier
associated with currently held locks and determine that they do not
match. This signifies the client's new instantiation and subsequent
loss of locking state. As a result, the server is free to release
all locks held which are associated with the old client ID which was
derived from the old verifier.

Note that the verifier must have the same uniqueness properties of

the verifier for the COMMIT operation.

9.6.2. Server Failure and Recovery

If the server loses locking state (usually as a result of a restart
or reboot), it must allow clients time to discover this fact and re-
establish the lost locking state. The client must be able to re-
establish the locking state without having the server deny valid
requests because the server has granted conflicting access to another
client. Likewise, if there is the possibility that clients have not
yet re-established their locking state for a file, the server must
disallow READ and WRITE operations for that file. The duration of
this recovery period is equal to the duration of the lease period.

A client can determine that server failure (and thus loss of locking
state) has occurred, when it receives one of two errors. The
NFS4ERR_STALE_STATEID error indicates a stateid invalidated by a
reboot or restart. The NFS4ERR_STALE_CLIENTID error indicates a
client ID invalidated by reboot or restart. When either of these are
received, the client must establish a new client ID (see
Section 9.1.1) and re-establish the locking state as discussed below.

The period of special handling of locking and READs and WRITEs, equal
in duration to the lease period, is referred to as the "grace
period". During the grace period, clients recover locks and the
associated state by reclaim-type locking requests (i.e., LOCK
requests with reclaim set to true and OPEN operations with a claim
type of either CLAIM_PREVIOUS or CLAIM_DELEGATE_PREV). During the
grace period, the server must reject READ and WRITE operations and
non-reclaim locking requests (i.e., other LOCK and OPEN operations)
with an error of NFS4ERR_GRACE.

If the server can reliably determine that granting a non-reclaim
request will not conflict with reclamation of locks by other clients,
the NFS4ERR_GRACE error does not have to be returned and the non-
reclaim client request can be serviced. For the server to be able to
service READ and WRITE operations during the grace period, it must
again be able to guarantee that no possible conflict could arise
between an impending reclaim locking request and the READ or WRITE
operation. If the server is unable to offer that guarantee, the
NFS4ERR_GRACE error must be returned to the client.

For a server to provide simple, valid handling during the grace
period, the easiest method is to simply reject all non-reclaim
locking requests and READ and WRITE operations by returning the
NFS4ERR_GRACE error. However, a server may keep information about
granted locks in stable storage. With this information, the server
could determine if a regular lock or READ or WRITE operation can be

safely processed.

For example, if a count of locks on a given file is available in
stable storage, the server can track reclaimed locks for the file and
when all reclaims have been processed, non-reclaim locking requests
may be processed. This way the server can ensure that non-reclaim
locking requests will not conflict with potential reclaim requests.
With respect to I/O requests, if the server is able to determine that
there are no outstanding reclaim requests for a file by information
from stable storage or another similar mechanism, the processing of
I/O requests could proceed normally for the file.

To reiterate, for a server that allows non-reclaim lock and I/O
requests to be processed during the grace period, it MUST determine
that no lock subsequently reclaimed will be rejected and that no lock
subsequently reclaimed would have prevented any I/O operation
processed during the grace period.

Clients should be prepared for the return of NFS4ERR_GRACE errors for
non-reclaim lock and I/O requests. In this case the client should
employ a retry mechanism for the request. A delay (on the order of
several seconds) between retries should be used to avoid overwhelming
the server. Further discussion of the general issue is included in
[Floyd]. The client must account for the server that is able to
perform I/O and non-reclaim locking requests within the grace period
as well as those that cannot do so.

A reclaim-type locking request outside the server's grace period can
only succeed if the server can guarantee that no conflicting lock or
I/O request has been granted since reboot or restart.

A server may, upon restart, establish a new value for the lease
period. Therefore, clients should, once a new client ID is
established, refetch the lease_time attribute and use it as the basis
for lease renewal for the lease associated with that server.
However, the server must establish, for this restart event, a grace
period at least as long as the lease period for the previous server
instantiation. This allows the client state obtained during the
previous server instance to be reliably re-established.

9.6.3. Network Partitions and Recovery

If the duration of a network partition is greater than the lease
period provided by the server, the server will have not received a
lease renewal from the client. If this occurs, the server may cancel
the lease and free all locks held for the client. As a result, all
stateids held by the client will become invalid or stale. Once the
client is able to reach the server after such a network partition,

all I/O submitted by the client with the now invalid stateids will
fail with the server returning the error NFS4ERR_EXPIRED. Once this
error is received, the client will suitably notify the application
that held the lock.

9.6.3.1. Courtesy Locks

As a courtesy to the client or as an optimization, the server may
continue to hold locks, including delegations, on behalf of a client
for which recent communication has extended beyond the lease period,
delaying the cancellation of the lease. If the server receives a
lock or I/O request that conflicts with one of these courtesy locks
or if it runs out of resources, the server MAY cause lease
cancellation to occur at that time and henceforth return
NFS4ERR_EXPIRED when any of the stateids associated with the freed
locks is used. If lease cancellation has not occurred and the server
receives a lock or I/O request that conflicts with one of the
courtesy locks, the requirements are as follows:

o In the case of a courtesy lock which is not a delegation, it MUST
   free the courtesy lock and grant the new request.

o In the case of lock or IO request which conflicts with a
   delegation which is being held as courtesy lock, the server MAY
   delay resolution of request but MUST NOT reject the request and
   MUST free the delegation and grant the new request eventually.

o In the case of a requests for a delegation which conflicts with a
   delegation which is being held as courtesy lock, the server MAY
   grant the new request or not as it chooses, but if it grants the
   conflicting request, the delegation haled as courtesy lock MUST be
   freed.

If the server does not reboot or cancel the lease before the network
partition is healed, when the original client tries to access a
courtesy lock which was freed, the server SHOULD send back a
NFS4ERR_BAD_STATEID to the client. If the client tries to access a
courtesy lock which was not freed, then the server SHOULD mark all of
the courtesy locks as implicitly being renewed.

9.6.3.2. Lease Cancellation

As a result of lease expiration, leases may be cancelled, either
immediately upon expiration or subsequently, depending on the
occurrence of a conflicting lock or extension of the period of
partition beyond what the server will tolerate.

When a lease is cancelled, all locking state associated with it is

freed and use of any the associated stateids will result in
NFS4ERR_EXPIRED being returned. Similarly, use of the associated
clientid will result in NFS4ERR_EXPIRED being returned.

The client should recover from this situation by using SETCLIENTID
followed by SETCLIENTID_CONFIRM, in order to establish a new
clientid. Once a lock is obtained using this clientid, a lease will
be established.

9.6.3.3. Client's Reaction to a Freed Lock

There is no way for a client to predetermine how a given server is
going to behave during a network partition. When the partition
heals, either the client still has all of its locks, it has some of
its locks, or it has none of them. The client will be able to
examine the various error return values to determine its response.

NFS4ERR_EXPIRED:

   All locks have been freed as a result of a lease cancellation
   which occurred during the partition. The client should use a
   SETCLIENTID to recover.

NFS4ERR_ADMIN_REVOKED:

   The current lock has been revoked before, during, or after the
   partition. The client SHOULD handle this error as it normally
   would.

NFS4ERR_BAD_STATEID:

   The current lock has been revoked/released during the partition
   and the server did not reboot. Other locks MAY still be renewed.
   The client need not do a SETCLIENTID and instead SHOULD probe via
   a RENEW call.

NFS4ERR_RECLAIM_BAD:

   The current lock has been revoked during the partition and the
   server rebooted. The server might have no information on the
   other locks. They may still be renewable.

NFS4ERR_NO_GRACE:

   The client's locks have been revoked during the partition and the
   server rebooted. None of the client's locks will be renewable.

NFS4ERR_OLD_STATEID:

> The server has not rebooted. The client SHOULD handle this error
> as it normally would.

9.6.3.4. Edge Conditions

When a network partition is combined with a server reboot, then both
the server and client have responsibilities to ensure that the client
does not reclaim a lock which it should no longer be able to access.
Briefly those are:

o Client's responsibility: A client MUST NOT attempt to reclaim any
   locks which it did not hold at the end of its most recent
   successfully established client lease.

o Server's responsibility: A server MUST NOT allow a client to
   reclaim a lock unless it knows that it could not have since
   granted a conflicting lock. However, in deciding whether a
   conflicting lock could have been granted, it is permitted to
   assume its clients are responsible, as above.

A server may consider a client's lease "successfully established"
once it has received an open operation from that client.

The above are directed to CLAIM_PREVIOUS reclaims and not to
CLAIM_DELEGATE_PREV reclaims, which generally do not involve a server
reboot. However, when a server persistently stores delegation
information to support CLAIM_DELEGATE_PREV across a period in which
both client and server are down at the same time, similar strictures
apply.

The next sections give examples showing what can go wrong if these
responsibilities are neglected, and provides examples of server
implementation strategies that could meet a server's
responsibilities.

9.6.3.4.1. First Server Edge Condition

The first edge condition has the following scenario:

1. Client A acquires a lock.

2. Client A and server experience mutual network partition, such
    that client A is unable to renew its lease.

3. Client A's lease expires, so server releases lock.

4. Client B acquires a lock that would have conflicted with that of
    Client A.

5. Client B releases the lock

6. Server reboots

7. Network partition between client A and server heals.

8. Client A issues a RENEW operation, and gets back a
    NFS4ERR_STALE_CLIENTID.

9. Client A reclaims its lock within the server's grace period.

Thus, at the final step, the server has erroneously granted client
A's lock reclaim. If client B modified the object the lock was
protecting, client A will experience object corruption.

9.6.3.4.2. Second Server Edge Condition

The second known edge condition follows:

1. Client A acquires a lock.

2. Server reboots.

3. Client A and server experience mutual network partition, such
    that client A is unable to reclaim its lock within the grace
    period.

4. Server's reclaim grace period ends. Client A has no locks
    recorded on server.

5. Client B acquires a lock that would have conflicted with that of
    Client A.

6. Client B releases the lock.

7. Server reboots a second time.

8. Network partition between client A and server heals.

9. Client A issues a RENEW operation, and gets back a
    NFS4ERR_STALE_CLIENTID.

10. Client A reclaims its lock within the server's grace period.

As with the first edge condition, the final step of the scenario of

the second edge condition has the server erroneously granting client
A's lock reclaim.

9.6.3.4.3. Handling Server Edge Conditions

In both of the above examples, the client attempts reclaim of a lock
that it held at the end of its most recent successfully established
lease; thus, it has fulfilled its responsibility.

The server, however, has failed, by granting a reclaim, despite
having granted a conflicting lock since the reclaimed lock was last
held.

Solving these edge conditions requires that the server either assume
after it reboots that edge condition occurs, and thus return
NFS4ERR_NO_GRACE for all reclaim attempts, or that the server record
some information in stable storage. The amount of information the
server records in stable storage is in inverse proportion to how
harsh the server wants to be whenever the edge conditions occur. The
server that is completely tolerant of all edge conditions will record
in stable storage every lock that is acquired, removing the lock
record from stable storage only when the lock is unlocked by the
client and the lock's owner advances the sequence number such that
the lock release is not the last stateful event for the owner's
sequence. For the two aforementioned edge conditions, the harshest a
server can be, and still support a grace period for reclaims,
requires that the server record in stable storage information some
minimal information. For example, a server implementation could, for
each client, save in stable storage a record containing:

o the client's id string

o a boolean that indicates if the client's lease expired or if there
   was administrative intervention (see Section 9.8) to revoke a
   byte-range lock, share reservation, or delegation

o a timestamp that is updated the first time after a server boot or
   reboot the client acquires byte-range locking, share reservation,
   or delegation state on the server. The timestamp need not be
   updated on subsequent lock requests until the server reboots.

The server implementation would also record in the stable storage the
timestamps from the two most recent server reboots.

Assuming the above record keeping, for the first edge condition,
after the server reboots, the record that client A's lease expired
means that another client could have acquired a conflicting record
lock, share reservation, or delegation. Hence the server must reject

a reclaim from client A with the error NFS4ERR_NO_GRACE or
NFS4ERR_RECLAIM_BAD.

For the second edge condition, after the server reboots for a second
time, the record that the client had an unexpired record lock, share
reservation, or delegation established before the server's previous
incarnation means that the server must reject a reclaim from client A
with the error NFS4ERR_NO_GRACE or NFS4ERR_RECLAIM_BAD.

Regardless of the level and approach to record keeping, the server
MUST implement one of the following strategies (which apply to
reclaims of share reservations, byte-range locks, and delegations):

1. Reject all reclaims with NFS4ERR_NO_GRACE. This is super harsh,
    but necessary if the server does not want to record lock state in
    stable storage.

2. Record sufficient state in stable storage to meet its
    responsibilities. In doubt, the server should err on the side of
    being harsh.

    In the event that, after a server reboot, the server determines
    that there is unrecoverable damage or corruption to the the
    stable storage, then for all clients and/or locks affected, the
    server MUST return NFS4ERR_NO_GRACE.

9.6.3.4.4. Client Edge Condition

A third edge condition effects the client and not the server. If the
server reboots in the middle of the client reclaiming some locks and
then a network partition is established, the client might be in the
situation of having reclaimed some, but not all locks. In that case,
a conservative client would assume that the non-reclaimed locks were
revoked.

The third known edge condition follows:

1.   Client A acquires a lock 1.

2.   Client A acquires a lock 2.

3.   Server reboots.

4.   Client A issues a RENEW operation, and gets back a
     NFS4ERR_STALE_CLIENTID.

5.   Client A reclaims its lock 1 within the server's grace period.

6.   Client A and server experience mutual network partition, such
     that client A is unable to reclaim its remaining locks within
     the grace period.

7.   Server's reclaim grace period ends.

8.   Client B acquires a lock that would have conflicted with Client
     A's lock 2.

9.   Client B releases the lock.

10. Server reboots a second time.

11. Network partition between client A and server heals.

12. Client A issues a RENEW operation, and gets back a
     NFS4ERR_STALE_CLIENTID.

13. Client A reclaims both lock 1 and lock 2 within the server's
     grace period.

At the last step, the client reclaims lock 2 as if it had held that
lock continuously, when in fact a conflicting lock was granted to
client B.

This occurs because the client failed its responsibility, by
attempting to reclaim lock 2 even though it had not held that lock at
the end of the lease that was established by the SETCLIENTID after
the first server reboot. (The client did hold lock 2 on a previous
lease. But it is only the most recent lease that matters.)

A server could avoid this situation by rejecting the reclaim of lock
2. However, to do so accurately it would have to ensure that
additional information about individual locks held survives reboot.
Server implementations are not required to do that, so the client
must not assume that the server will.

Instead, a client MUST reclaim only those locks which it successfully
acquired from the previous server instance, omitting any that it
failed to reclaim before a new reboot. Thus, in the last step above,
client A should reclaim only lock 1.

9.6.3.4.5. Client's Handling of Reclaim Errors

A mandate for the client's handling of the NFS4ERR_NO_GRACE and
NFS4ERR_RECLAIM_BAD errors is outside the scope of this
specification, since the strategies for such handling are very
dependent on the client's operating environment. However, one

potential approach is described below.

When the client's reclaim fails, it could examine the change
attribute of the objects the client is trying to reclaim state for,
and use that to determine whether to re-establish the state via
normal OPEN or LOCK requests. This is acceptable provided the
client's operating environment allows it. In other words, the client
implementor is advised to document for his users the behavior. The
client could also inform the application that its byte-range lock or
share reservations (whether they were delegated or not) have been
lost, such as via a UNIX signal, a GUI pop-up window, etc. See
Section 10.5, for a discussion of what the client should do for
dealing with unreclaimed delegations on client state.

For further discussion of revocation of locks see Section 9.8.

9.7. Recovery from a Lock Request Timeout or Abort

In the event a lock request times out, a client may decide to not
retry the request. The client may also abort the request when the
process for which it was issued is terminated (e.g., in UNIX due to a
signal). It is possible though that the server received the request
and acted upon it. This would change the state on the server without
the client being aware of the change. It is paramount that the
client re-synchronize state with server before it attempts any other
operation that takes a seqid and/or a stateid with the same state-
owner. This is straightforward to do without a special re-
synchronize operation.

Since the server maintains the last lock request and response
received on the state-owner, for each state-owner, the client should
cache the last lock request it sent such that the lock request did
not receive a response. From this, the next time the client does a
lock operation for the state-owner, it can send the cached request,
if there is one, and if the request was one that established state
(e.g., a LOCK or OPEN operation), the server will return the cached
result or if never saw the request, perform it. The client can
follow up with a request to remove the state (e.g., a LOCKU or CLOSE
operation). With this approach, the sequencing and stateid
information on the client and server for the given state-owner will
re-synchronize and in turn the lock state will re-synchronize.

9.8. Server Revocation of Locks

At any point, the server can revoke locks held by a client and the
client must be prepared for this event. When the client detects that
its locks have been or may have been revoked, the client is
responsible for validating the state information between itself and

the server. Validating locking state for the client means that it
must verify or reclaim state for each lock currently held.

The first instance of lock revocation is upon server reboot or re-
initialization. In this instance the client will receive an error
(NFS4ERR_STALE_STATEID or NFS4ERR_STALE_CLIENTID) and the client will
proceed with normal crash recovery as described in the previous
section.

The second lock revocation event is the inability to renew the lease
before expiration. While this is considered a rare or unusual event,
the client must be prepared to recover. Both the server and client
will be able to detect the failure to renew the lease and are capable
of recovering without data corruption. For the server, it tracks the
last renewal event serviced for the client and knows when the lease
will expire. Similarly, the client must track operations which will
renew the lease period. Using the time that each such request was
sent and the time that the corresponding reply was received, the
client should bound the time that the corresponding renewal could
have occurred on the server and thus determine if it is possible that
a lease period expiration could have occurred.

The third lock revocation event can occur as a result of
administrative intervention within the lease period. While this is
considered a rare event, it is possible that the server's
administrator has decided to release or revoke a particular lock held
by the client. As a result of revocation, the client will receive an
error of NFS4ERR_ADMIN_REVOKED. In this instance the client may
assume that only the state-owner's locks have been lost. The client
notifies the lock holder appropriately. The client MUST NOT assume
the lease period has been renewed as a result of a failed operation.

When the client determines the lease period may have expired, the
client must mark all locks held for the associated lease as
"unvalidated". This means the client has been unable to re-establish
or confirm the appropriate lock state with the server. As described
in Section 9.6, there are scenarios in which the server may grant
conflicting locks after the lease period has expired for a client.
When it is possible that the lease period has expired, the client
must validate each lock currently held to ensure that a conflicting
lock has not been granted. The client may accomplish this task by
issuing an I/O request, either a pending I/O or a zero-length read,
specifying the stateid associated with the lock in question. If the
response to the request is success, the client has validated all of
the locks governed by that stateid and re-established the appropriate
state between itself and the server.

If the I/O request is not successful, then one or more of the locks

associated with the stateid was revoked by the server and the client
must notify the owner.

9.9. Share Reservations

A share reservation is a mechanism to control access to a file. It
is a separate and independent mechanism from byte-range locking.
When a client opens a file, it issues an OPEN operation to the server
specifying the type of access required (READ, WRITE, or BOTH) and the
type of access to deny others (OPEN4_SHARE_DENY_NONE,
OPEN4_SHARE_DENY_READ, OPEN4_SHARE_DENY_WRITE, or
OPEN4_SHARE_DENY_BOTH). If the OPEN fails the client will fail the
application's open request.

Pseudo-code definition of the semantics:

```
  if (request.access == 0)
          return (NFS4ERR_INVAL)
  else if ((request.access & file_state.deny)) ||
      (request.deny & file_state.access))
          return (NFS4ERR_DENIED)
```

This checking of share reservations on OPEN is done with no exception
for an existing OPEN for the same open-owner.

The constants used for the OPEN and OPEN_DOWNGRADE operations for the
access and deny fields are as follows:

```
const OPEN4_SHARE_ACCESS_READ   = 0x00000001;
const OPEN4_SHARE_ACCESS_WRITE  = 0x00000002;
const OPEN4_SHARE_ACCESS_BOTH   = 0x00000003;

const OPEN4_SHARE_DENY_NONE     = 0x00000000;
const OPEN4_SHARE_DENY_READ     = 0x00000001;
const OPEN4_SHARE_DENY_WRITE    = 0x00000002;
const OPEN4_SHARE_DENY_BOTH     = 0x00000003;
```

9.10. OPEN/CLOSE Operations

To provide correct share semantics, a client MUST use the OPEN
operation to obtain the initial filehandle and indicate the desired
access and what access, if any, to deny. Even if the client intends
to use a stateid of all 0's or all 1's, it must still obtain the
filehandle for the regular file with the OPEN operation so the
appropriate share semantics can be applied. Clients that do not have
a deny mode built into their programming interfaces for opening a
file should request a deny mode of OPEN4_SHARE_DENY_NONE.

The OPEN operation with the CREATE flag, also subsumes the CREATE
operation for regular files as used in previous versions of the NFS
protocol. This allows a create with a share to be done atomically.

The CLOSE operation removes all share reservations held by the open-
owner on that file. If byte-range locks are held, the client SHOULD
release all locks before issuing a CLOSE. The server MAY free all
outstanding locks on CLOSE but some servers may not support the CLOSE
of a file that still has byte-range locks held. The server MUST
return failure, NFS4ERR_LOCKS_HELD, if any locks would exist after
the CLOSE.

The LOOKUP operation will return a filehandle without establishing
any lock state on the server. Without a valid stateid, the server
will assume the client has the least access. For example, if one
client opened a file with OPEN4_SHARE_DENY_BOTH and another client
accesses the file via a filehandle obtained through LOOKUP, the
second client could only read the file using the special read bypass
stateid. The second client could not WRITE the file at all because
it would not have a valid stateid from OPEN and the special anonymous
stateid would not be allowed access.

9.10.1. Close and Retention of State Information

Since a CLOSE operation requests deallocation of a stateid, dealing
with retransmission of the CLOSE, may pose special difficulties,
since the state information, which normally would be used to
determine the state of the open file being designated, might be
deallocated, resulting in an NFS4ERR_BAD_STATEID error.

Servers may deal with this problem in a number of ways. To provide
the greatest degree assurance that the protocol is being used
properly, a server should, rather than deallocate the stateid, mark
it as close-pending, and retain the stateid with this status, until
later deallocation. In this way, a retransmitted CLOSE can be
recognized since the stateid points to state information with this
distinctive status, so that it can be handled without error.

When adopting this strategy, a server should retain the state
information until the earliest of:

o Another validly sequenced request for the same open-owner, that is
   not a retransmission.

o The time that an open-owner is freed by the server due to period
   with no activity.

o All locks for the client are freed as a result of a SETCLIENTID.

Servers may avoid this complexity, at the cost of less complete
protocol error checking, by simply responding NFS4_OK in the event of
a CLOSE for a deallocated stateid, on the assumption that this case
must be caused by a retransmitted close. When adopting this
approach, it is desirable to at least log an error when returning a
no-error indication in this situation. If the server maintains a
reply-cache mechanism, it can verify the CLOSE is indeed a
retransmission and avoid error logging in most cases.

9.11. Open Upgrade and Downgrade

When an OPEN is done for a file and the open-owner for which the open
is being done already has the file open, the result is to upgrade the
open file status maintained on the server to include the access and
deny bits specified by the new OPEN as well as those for the existing
OPEN. The result is that there is one open file, as far as the
protocol is concerned, and it includes the union of the access and
deny bits for all of the OPEN requests completed. Only a single
CLOSE will be done to reset the effects of both OPENs. Note that the
client, when issuing the OPEN, may not know that the same file is in
fact being opened. The above only applies if both OPENs result in
the OPENed object being designated by the same filehandle.

When the server chooses to export multiple filehandles corresponding
to the same file object and returns different filehandles on two
different OPENs of the same file object, the server MUST NOT "OR"
together the access and deny bits and coalesce the two open files.
Instead the server must maintain separate OPENs with separate
stateids and will require separate CLOSEs to free them.

When multiple open files on the client are merged into a single open
file object on the server, the close of one of the open files (on the
client) may necessitate change of the access and deny status of the
open file on the server. This is because the union of the access and
deny bits for the remaining opens may be smaller (i.e., a proper
subset) than previously. The OPEN_DOWNGRADE operation is used to
make the necessary change and the client should use it to update the
server so that share reservation requests by other clients are
handled properly. The stateid returned has the same "other" field as
that passed to the server. The "seqid" value in the returned stateid
MUST be incremented, even in situations in which there is no change
to the access and deny bits for the file.

9.12. Short and Long Leases

   When determining the time period for the server lease, the usual
   lease tradeoffs apply. Short leases are good for fast server
   recovery at a cost of increased RENEW or READ (with zero length)
   requests. Longer leases are certainly kinder and gentler to servers
   trying to handle very large numbers of clients. The number of RENEW
   requests drop in proportion to the lease time. The disadvantages of
   long leases are slower recovery after server failure (the server must
   wait for the leases to expire and the grace period to elapse before
   granting new lock requests) and increased file contention (if client
   fails to transmit an unlock request then server must wait for lease
   expiration before granting new locks).

   Long leases are usable if the server is able to store lease state in
   non-volatile memory. Upon recovery, the server can reconstruct the
   lease state from its non-volatile memory and continue operation with
   its clients and therefore long leases would not be an issue.

9.13. Clocks, Propagation Delay, and Calculating Lease Expiration

   To avoid the need for synchronized clocks, lease times are granted by
   the server as a time delta. However, there is a requirement that the
   client and server clocks do not drift excessively over the duration
   of the lock. There is also the issue of propagation delay across the
   network which could easily be several hundred milliseconds as well as
   the possibility that requests will be lost and need to be
   retransmitted.

   To take propagation delay into account, the client should subtract it
   from lease times (e.g., if the client estimates the one-way
   propagation delay as 200 msec, then it can assume that the lease is
   already 200 msec old when it gets it). In addition, it will take
   another 200 msec to get a response back to the server. So the client
   must send a lock renewal or write data back to the server 400 msec
   before the lease would expire.

   The server's lease period configuration should take into account the
   network distance of the clients that will be accessing the server's
   resources. It is expected that the lease period will take into
   account the network propagation delays and other network delay
   factors for the client population. Since the protocol does not allow
   for an automatic method to determine an appropriate lease period, the
   server's administrator may have to tune the lease period.

9.14. Migration, Replication and State

   When responsibility for handling a given file system is transferred
   to a new server (migration) or the client chooses to use an alternate
   server (e.g., in response to server unresponsiveness) in the context
   of file system replication, the appropriate handling of state shared
   between the client and server (i.e., locks, leases, stateids, and
   client IDs) is as described below. The handling differs between
   migration and replication. For related discussion of file server
   state and recover of such see the sections under Section 9.6.

   If a server replica or a server immigrating a file system agrees to,
   or is expected to, accept opaque values from the client that
   originated from another server, then servers SHOULD encode the
   "opaque" values in network byte order. This way, servers acting as
   replicas or immigrating file systems will be able to parse values
   like stateids, directory cookies, filehandles, etc. even if their
   native byte order is different from other servers cooperating in the
   replication and migration of the file system.

9.14.1. Migration and State

   In the case of migration, the servers involved in the migration of a
   file system SHOULD transfer all server state from the original to the
   new server. This must be done in a way that is transparent to the
   client. This state transfer will ease the client's transition when a
   file system migration occurs. If the servers are successful in
   transferring all state, the client will continue to use stateids
   assigned by the original server. Therefore the new server must
   recognize these stateids as valid. This holds true for the client ID
   as well. Since responsibility for an entire file system is
   transferred with a migration event, there is no possibility that
   conflicts will arise on the new server as a result of the transfer of
   locks.

   As part of the transfer of information between servers, leases would
   be transferred as well. The leases being transferred to the new
   server will typically have a different expiration time from those for
   the same client, previously on the old server. To maintain the
   property that all leases on a given server for a given client expire
   at the same time, the server should advance the expiration time to
   the later of the leases being transferred or the leases already
   present. This allows the client to maintain lease renewal of both
   classes without special effort.

   The servers may choose not to transfer the state information upon
   migration. However, this choice is discouraged. In this case, when
   the client presents state information from the original server (e.g.,

in a RENEW op or a READ op of zero length), the client must be
prepared to receive either NFS4ERR_STALE_CLIENTID or
NFS4ERR_STALE_STATEID from the new server. The client should then
recover its state information as it normally would in response to a
server failure. The new server must take care to allow for the
recovery of state information as it would in the event of server
restart.

A client SHOULD re-establish new callback information with the new
server as soon as possible, according to sequences described in
Section 15.35 and Section 15.36. This ensures that server operations
are not blocked by the inability to recall delegations.

9.14.2. Replication and State

Since client switch-over in the case of replication is not under
server control, the handling of state is different. In this case,
leases, stateids and client IDs do not have validity across a
transition from one server to another. The client must re-establish
its locks on the new server. This can be compared to the re-
establishment of locks by means of reclaim-type requests after a
server reboot. The difference is that the server has no provision to
distinguish requests reclaiming locks from those obtaining new locks
or to defer the latter. Thus, a client re-establishing a lock on the
new server (by means of a LOCK or OPEN request), may have the
requests denied due to a conflicting lock. Since replication is
intended for read-only use of file systems, such denial of locks
should not pose large difficulties in practice. When an attempt to
re-establish a lock on a new server is denied, the client should
treat the situation as if his original lock had been revoked.

9.14.3. Notification of Migrated Lease

In the case of lease renewal, the client may not be submitting
requests for a file system that has been migrated to another server.
This can occur because of the implicit lease renewal mechanism. The
client renews leases for all file systems when submitting a request
to any one file system at the server.

In order for the client to schedule renewal of leases that may have
been relocated to the new server, the client must find out about
lease relocation before those leases expire. To accomplish this, all
operations which implicitly renew leases for a client (such as OPEN,
CLOSE, READ, WRITE, RENEW, LOCK, and others), will return the error
NFS4ERR_LEASE_MOVED if responsibility for any of the leases to be
renewed has been transferred to a new server. This condition will
continue until the client receives an NFS4ERR_MOVED error and the
server receives the subsequent GETATTR(fs_locations) for an access to

each file system for which a lease has been moved to a new server.
By convention, the compound including the GETATTR(fs_locations)
SHOULD append a RENEW operation to permit the server to identify the
client doing the access.

Upon receiving the NFS4ERR_LEASE_MOVED error, a client that supports
file system migration MUST probe all file systems from that server on
which it holds open state. Once the client has successfully probed
all those file systems which are migrated, the server MUST resume
normal handling of stateful requests from that client.

In order to support legacy clients that do not handle the
NFS4ERR_LEASE_MOVED error correctly, the server SHOULD time out after
a wait of at least two lease periods, at which time it will resume
normal handling of stateful requests from all clients. If a client
attempts to access the migrated files, the server MUST reply
NFS4ERR_MOVED.

When the client receives an NFS4ERR_MOVED error, the client can
follow the normal process to obtain the new server information
(through the fs_locations attribute) and perform renewal of those
leases on the new server. If the server has not had state
transferred to it transparently, the client will receive either
NFS4ERR_STALE_CLIENTID or NFS4ERR_STALE_STATEID from the new server,
as described above. The client can then recover state information as
it does in the event of server failure.

9.14.4. Migration and the Lease_time Attribute

In order that the client may appropriately manage its leases in the
case of migration, the destination server must establish proper
values for the lease_time attribute.

When state is transferred transparently, that state should include
the correct value of the lease_time attribute. The lease_time
attribute on the destination server must never be less than that on
the source since this would result in premature expiration of leases
granted by the source server. Upon migration in which state is
transferred transparently, the client is under no obligation to re-
fetch the lease_time attribute and may continue to use the value
previously fetched (on the source server).

If state has not been transferred transparently (i.e., the client
sees a real or simulated server reboot), the client should fetch the
value of lease_time on the new (i.e., destination) server, and use it
for subsequent locking requests. However the server must respect a
grace period at least as long as the lease_time on the source server,
in order to ensure that clients have ample time to reclaim their

locks before potentially conflicting non-reclaimed locks are granted.
The means by which the new server obtains the value of lease_time on
the old server is left to the server implementations. It is not
specified by the NFS version 4 protocol.

10. Client-Side Caching

Client-side caching of data, of file attributes, and of file names is
essential to providing good performance with the NFS protocol.
Providing distributed cache coherence is a difficult problem and
previous versions of the NFS protocol have not attempted it.
Instead, several NFS client implementation techniques have been used
to reduce the problems that a lack of coherence poses for users.
These techniques have not been clearly defined by earlier protocol
specifications and it is often unclear what is valid or invalid
client behavior.

The NFSv4 protocol uses many techniques similar to those that have
been used in previous protocol versions. The NFSv4 protocol does not
provide distributed cache coherence. However, it defines a more
limited set of caching guarantees to allow locks and share
reservations to be used without destructive interference from client
side caching.

In addition, the NFSv4 protocol introduces a delegation mechanism
which allows many decisions normally made by the server to be made
locally by clients. This mechanism provides efficient support of the
common cases where sharing is infrequent or where sharing is read-
only.

10.1. Performance Challenges for Client-Side Caching

Caching techniques used in previous versions of the NFS protocol have
been successful in providing good performance. However, several
scalability challenges can arise when those techniques are used with
very large numbers of clients. This is particularly true when
clients are geographically distributed which classically increases
the latency for cache re-validation requests.

The previous versions of the NFS protocol repeat their file data
cache validation requests at the time the file is opened. This
behavior can have serious performance drawbacks. A common case is
one in which a file is only accessed by a single client. Therefore,
sharing is infrequent.

In this case, repeated reference to the server to find that no
conflicts exist is expensive. A better option with regards to

performance is to allow a client that repeatedly opens a file to do
so without reference to the server. This is done until potentially
conflicting operations from another client actually occur.

A similar situation arises in connection with file locking. Sending
file lock and unlock requests to the server as well as the read and
write requests necessary to make data caching consistent with the
locking semantics (see Section 10.3.2) can severely limit
performance. When locking is used to provide protection against
infrequent conflicts, a large penalty is incurred. This penalty may
discourage the use of file locking by applications.

The NFSv4 protocol provides more aggressive caching strategies with
the following design goals:

o Compatibility with a large range of server semantics.

o Provide the same caching benefits as previous versions of the NFS
   protocol when unable to provide the more aggressive model.

o Requirements for aggressive caching are organized so that a large
   portion of the benefit can be obtained even when not all of the
   requirements can be met.

The appropriate requirements for the server are discussed in later
sections in which specific forms of caching are covered (see
Section 10.4).

10.2. Delegation and Callbacks

Recallable delegation of server responsibilities for a file to a
client improves performance by avoiding repeated requests to the
server in the absence of inter-client conflict. With the use of a
"callback" RPC from server to client, a server recalls delegated
responsibilities when another client engages in sharing of a
delegated file.

A delegation is passed from the server to the client, specifying the
object of the delegation and the type of delegation. There are
different types of delegations but each type contains a stateid to be
used to represent the delegation when performing operations that
depend on the delegation. This stateid is similar to those
associated with locks and share reservations but differs in that the
stateid for a delegation is associated with a client ID and may be
used on behalf of all the open-owners for the given client. A
delegation is made to the client as a whole and not to any specific
process or thread of control within it.

Because callback RPCs may not work in all environments (due to firewalls, for example), correct protocol operation does not depend on them. Preliminary testing of callback functionality by means of a CB_NULL procedure determines whether callbacks can be supported. The CB_NULL procedure checks the continuity of the callback path. A server makes a preliminary assessment of callback availability to a given client and avoids delegating responsibilities until it has determined that callbacks are supported. Because the granting of a delegation is always conditional upon the absence of conflicting access, clients must not assume that a delegation will be granted and they must always be prepared for OPENs to be processed without any delegations being granted.

Once granted, a delegation behaves in most ways like a lock. There is an associated lease that is subject to renewal together with all of the other leases held by that client.

Unlike locks, an operation by a second client to a delegated file will cause the server to recall a delegation through a callback.

On recall, the client holding the delegation must flush modified state (such as modified data) to the server and return the delegation. The conflicting request will not be acted on until the recall is complete. The recall is considered complete when the client returns the delegation or the server times out its wait for the delegation to be returned and revokes the delegation as a result of the timeout. In the interim, the server will either delay responding to conflicting requests or respond to them with NFS4ERR_DELAY. Following the resolution of the recall, the server has the information necessary to grant or deny the second client's request.

At the time the client receives a delegation recall, it may have substantial state that needs to be flushed to the server. Therefore, the server should allow sufficient time for the delegation to be returned since it may involve numerous RPCs to the server. If the server is able to determine that the client is diligently flushing state to the server as a result of the recall, the server MAY extend the usual time allowed for a recall. However, the time allowed for recall completion SHOULD NOT be unbounded.

An example of this is when responsibility to mediate opens on a given file is delegated to a client (see Section 10.4). The server will not know what opens are in effect on the client. Without this knowledge the server will be unable to determine if the access and deny state for the file allows any particular open until the delegation for the file has been returned.

A client failure or a network partition can result in failure to
respond to a recall callback. In this case, the server will revoke
the delegation which in turn will render useless any modified state
still on the client.

Clients need to be aware that server implementors may enforce
practical limitations on the number of delegations issued. Further,
as there is no way to determine which delegations to revoke, the
server is allowed to revoke any. If the server is implemented to
revoke another delegation held by that client, then the client may be
able to determine that a limit has been reached because each new
delegation request results in a revoke. The client could then
determine which delegations it may not need and preemptively release
them.

10.2.1. Delegation Recovery

There are three situations that delegation recovery must deal with:

o Client reboot or restart

o Server reboot or restart (see Section 9.6.3.1)

o Network partition (full or callback-only)

In the event the client reboots or restarts, the confirmation of a
SETCLIENTID done with an nfs_client_id4 with a new verifier4 value
will result in the release of byte-range locks and share
reservations. Delegations, however, may be treated a bit
differently.

There will be situations in which delegations will need to be
reestablished after a client reboots or restarts. The reason for
this is the client may have file data stored locally and this data
was associated with the previously held delegations. The client will
need to reestablish the appropriate file state on the server.

To allow for this type of client recovery, the server MAY allow
delegations to be retained after other sort of locks are released.
This implies that requests from other clients that conflict with
these delegations will need to wait. Because the normal recall
process may require significant time for the client to flush changed
state to the server, other clients need to be prepared for delays
that occur because of a conflicting delegation. In order to give
clients a chance to get through the reboot process during which
leases will not be renewed, the server MAY extend the period for
delegation recovery beyond the typical lease expiration period. For
open delegations, such delegations that are not released are

reclaimed using OPEN with a claim type of CLAIM_DELEGATE_PREV. (See
Section 10.5 and Section 15.18 for discussion of open delegation and
the details of OPEN respectively).

A server MAY support a claim type of CLAIM_DELEGATE_PREV, but if it
does, it MUST NOT remove delegations upon SETCLIENTID_CONFIRM and
instead MUST make them available for client reclaim using
CLAIM_DELEGATE_PREV. The server MUST NOT remove the delegations
until either the client does a DELEGPURGE, or one lease period has
elapsed from the time the later of the SETCLIENTID_CONFIRM or the
last successful CLAIM_DELEGATE_PREV reclaim.

Note that the requirement stated above is not meant to imply that
when the client is no longer obliged, as required above, to retain
delegation information, that it should necessarily dispose of it.
Some specific cases are:

o When the period is terminated by the occurrence of DELEGPURGE,
   deletion of unreclaimed delegations is appropriate and desirable.

o When the period is terminated by a lease period elapsing without a
   successful CLAIM_DELEGATE_PREV reclaim, and that situation appears
   to be the result of a network partition (i.e., lease expiration
   has occurred), a server's lease expiration approach, possibly
   including the use of courtesy locks would normally provide for the
   retention of unreclaimed delegations. Even in the event that
   lease cancellation occurs, such delegation should be reclaimed
   using CLAIM_DELEGATE_PREV as part of network partition recovery.

o When the period of non-communicating is followed by a client
   reboot, unreclaimed delegations, should also be reclaimable by use
   of CLAIM_DELEGATE_PREV as part of client reboot recovery.

o When the period is terminated by a lease period elapsing without a
   successful CLAIM_DELEGATE_PREV reclaim, and lease renewal is
   occurring, the server may well conclude that unreclaimed
   delegations have been abandoned, and consider the situation as one
   in which an implied DELEGPURGE should be assumed.

A server that supports a claim type of CLAIM_DELEGATE_PREV MUST
support the DELEGPURGE operation, and similarly a server that
supports DELEGPURGE MUST support CLAIM_DELEGATE_PREV. A server which
does not support CLAIM_DELEGATE_PREV MUST return NFS4ERR_NOTSUPP if
the client attempts to use that feature or performs a DELEGPURGE
operation.

Support for a claim type of CLAIM_DELEGATE_PREV, is often referred to
as providing for "client-persistent delegations" in that they allow

use of client persistent storage on the client to store data written
by the client, even across a client restart. It should be noted
that, with the optional exception noted below, this feature requires
persistent storage to be used on the client and does not add to
persistent storage requirements on the server.

One good way to think about client-persistent delegations is that for
the most part, they function like "courtesy locks", with special
semantic adjustments to allow them to be retained across a client
restart, which cause all other sorts of locks to be freed. Such
locks are generally not retained across a server restart. The one
exception is the case of simultaneous failure of the client and
server and is discussed below.

When the server indicates support of CLAIM_DELEGATE_PREV (implicitly)
by returning NFS_OK to DELEGPURGE, a client with a write delegation,
can use write-back caching for data to be written to the server,
deferring the write-back, until such time as the delegation is
recalled, possibly after intervening client restarts. Similarly,
when the server indicates support of CLAIM_DELEGATE_PREV, a client
with a read delegation and an open-for-write subordinate to that
delegation, may be sure of the integrity of its persistently cached
copy of the file after a client restart without specific verification
of the change attribute.

When the server reboots or restarts, delegations are reclaimed (using
the OPEN operation with CLAIM_PREVIOUS) in a similar fashion to byte-
range locks and share reservations. However, there is a slight
semantic difference. In the normal case, if the server decides that
a delegation should not be granted, it performs the requested action
(e.g., OPEN) without granting any delegation. For reclaim, the
server grants the delegation but a special designation is applied so
that the client treats the delegation as having been granted but
recalled by the server. Because of this, the client has the duty to
write all modified state to the server and then return the
delegation. This process of handling delegation reclaim reconciles
three principles of the NFSv4 protocol:

o Upon reclaim, a client claiming resources assigned to it by an
   earlier server instance must be granted those resources.

o The server has unquestionable authority to determine whether
   delegations are to be granted and, once granted, whether they are
   to be continued.

o The use of callbacks is not to be depended upon until the client
   has proven its ability to receive them.

When a client has more than a single open associated with a
delegation, state for those additional opens can be established using
OPEN operations of type CLAIM_DELEGATE_CUR. When these are used to
establish opens associated with reclaimed delegations, the server
MUST allow them when made within the grace period.

Situations in which there is a series of client and server restarts
where there is no restart of both at the same time, are dealt with
via a combination of CLAIM_DELEGATE_PREV and CLAIM_PREVIOUS reclaim
cycles. Persistent storage is needed only on the client. For each
server failure, a CLAIM_PREVIOUS reclaim cycle is done, while for
each client restart, a CLAIM_DELEGATE_PREV reclaim cycle is done.

To deal with the possibility of simultaneous failure of client and
server (e.g., a data center power outage), the server MAY
persistently store delegation information so that it can respond to a
CLAIM_DELEGATE_PREV reclaim request which it receives from a
restarting client. This is the one case in which persistent
delegation state can be retained across a server restart. A server
is not required to store this information, but if it does do so, it
should do so for write delegations and for read delegations, during
the pendency of which (across multiple client and/or server
instances), some open-for-write was done as part of delegation. When
the space to persistently record such information is limited, the
server should recall delegations in this class in preference to
keeping them active without persistent storage recording.

When a network partition occurs, delegations are subject to freeing
by the server when the lease renewal period expires. This is similar
to the behavior for locks and share reservations, and, as for locks
and share reservations it may be modified by support for "courtesy
locks" in which locks are not freed in the absence of a conflicting
lock request. Whereas, for locks and share reservations, freeing of
locks will occur immediately upon the appearance of a conflicting
request, for delegations, the server may institute period during
which conflicting requests are held off. Eventually the occurrence
of a conflicting request from another client will cause revocation of
the delegation.

A loss of the callback path (e.g., by later network configuration
change) will have a similar effect in that it can also result in
revocation of a delegation A recall request will fail and revocation
of the delegation will result.

A client normally finds out about revocation of a delegation when it
uses a stateid associated with a delegation and receives one of the
errors NFS4ERR_EXPIRED, NFS4ERR_BAD_STATEID, or NFS4ERR_ADMIN_REVOKED
(NFS4ERR_EXPIRED indicates that all lock state associated with the

client has been lost). It also may find out about delegation
revocation after a client reboot when it attempts to reclaim a
delegation and receives NFS4ERR_EXPIRED. Note that in the case of a
revoked OPEN_DELEGATE_WRITE delegation, there are issues because data
may have been modified by the client whose delegation is revoked and
separately by other clients. See Section 10.5.1 for a discussion of
such issues. Note also that when delegations are revoked,
information about the revoked delegation will be written by the
server to stable storage (as described in Section 9.6). This is done
to deal with the case in which a server reboots after revoking a
delegation but before the client holding the revoked delegation is
notified about the revocation.

Note that when there is a loss of a delegation, due to a network
partition in which all locks associated with the lease are lost, the
client will also receive the error NFS4ERR_EXPIRED. This case can be
distinguished from other situations in which delegations are revoked
by seeing that the associated clientid becomes invalid so that
NFS4ERR_STALE_CLIENTID is returned when it is used.

When NFS4ERR_EXPIRED is returned, the server MAY retain information
about the delegations held by the client, deleting those that are
invalidated by a conflicting request. Retaining such information
will allow the client to recover all non-invalidated delegations
using the claim type CLAIM_DELEGATE_PREV, once the
SETCLIENTID_CONFIRM is done to recover. Attempted recovery of a
delegation that the client has no record of, typically because they
were invalidated by conflicting requests, will get the error
NFS4ERR_BAD_RECLAIM. Once a reclaim is attempted for all delegations
that the client held, it SHOULD do a DELEGPURGE to allow any
remaining server delegation information to be freed.

10.3. Data Caching

When applications share access to a set of files, they need to be
implemented so as to take account of the possibility of conflicting
access by another application. This is true whether the applications
in question execute on different clients or reside on the same
client.

Share reservations and byte-range locks are the facilities the NFS
version 4 protocol provides to allow applications to coordinate
access by providing mutual exclusion facilities. The NFSv4
protocol's data caching must be implemented such that it does not
invalidate the assumptions that those using these facilities depend
upon.

10.3.1. **Data Caching and OPENs**

   In order to avoid invalidating the sharing assumptions that
   applications rely on, NFSv4 clients should not provide cached data to
   applications or modify it on behalf of an application when it would
   not be valid to obtain or modify that same data via a READ or WRITE
   operation.

   Furthermore, in the absence of open delegation (see Section 10.4) two
   additional rules apply. Note that these rules are obeyed in practice
   by many NFSv2 and NFSv3 clients.

   o First, cached data present on a client must be revalidated after
      doing an OPEN. Revalidating means that the client fetches the
      change attribute from the server, compares it with the cached
      change attribute, and if different, declares the cached data (as
      well as the cached attributes) as invalid. This is to ensure that
      the data for the OPENed file is still correctly reflected in the
      client's cache. This validation must be done at least when the
      client's OPEN operation includes DENY=WRITE or BOTH thus
      terminating a period in which other clients may have had the
      opportunity to open the file with WRITE access. Clients may
      choose to do the revalidation more often (i.e., at OPENs
      specifying DENY=NONE) to parallel the NFSv3 protocol's practice
      for the benefit of users assuming this degree of cache
      revalidation. Since the change attribute is updated for data and
      metadata modifications, some client implementors may be tempted to
      use the time_modify attribute and not the change attribute to
      validate cached data, so that metadata changes do not spuriously
      invalidate clean data. The implementor is cautioned in this
      approach. The change attribute is guaranteed to change for each
      update to the file, whereas time_modify is guaranteed to change
      only at the granularity of the time_delta attribute. Use by the
      client's data cache validation logic of time_modify and not the
      change attribute runs the risk of the client incorrectly marking
      stale data as valid.

   o Second, modified data must be flushed to the server before closing
      a file OPENed for write. This is complementary to the first rule.
      If the data is not flushed at CLOSE, the revalidation done after
      the client OPENs a file is unable to achieve its purpose. The
      other aspect to flushing the data before close is that the data
      must be committed to stable storage, at the server, before the
      CLOSE operation is requested by the client. In the case of a
      server reboot or restart and a CLOSEd file, it may not be possible
      to retransmit the data to be written to the file. Hence, this
      requirement.

10.3.2. **Data Caching and File Locking**

   For those applications that choose to use file locking instead of
   share reservations to exclude inconsistent file access, there is an
   analogous set of constraints that apply to client side data caching.
   These rules are effective only if the file locking is used in a way
   that matches in an equivalent way the actual READ and WRITE
   operations executed. This is as opposed to file locking that is
   based on pure convention. For example, it is possible to manipulate
   a two-megabyte file by dividing the file into two one-megabyte
   regions and protecting access to the two regions by file locks on
   bytes zero and one. A lock for write on byte zero of the file would
   represent the right to do READ and WRITE operations on the first
   region. A lock for write on byte one of the file would represent the
   right to do READ and WRITE operations on the second region. As long
   as all applications manipulating the file obey this convention, they
   will work on a local file system. However, they may not work with
   the NFSv4 protocol unless clients refrain from data caching.

   The rules for data caching in the file locking environment are:

   o First, when a client obtains a file lock for a particular region,
      the data cache corresponding to that region (if any cached data
      exists) must be revalidated. If the change attribute indicates
      that the file may have been updated since the cached data was
      obtained, the client must flush or invalidate the cached data for
      the newly locked region. A client might choose to invalidate all
      of non-modified cached data that it has for the file but the only
      requirement for correct operation is to invalidate all of the data
      in the newly locked region.

   o Second, before releasing a write lock for a region, all modified
      data for that region must be flushed to the server. The modified
      data must also be written to stable storage.

   Note that flushing data to the server and the invalidation of cached
   data must reflect the actual byte ranges locked or unlocked.
   Rounding these up or down to reflect client cache block boundaries
   will cause problems if not carefully done. For example, writing a
   modified block when only half of that block is within an area being
   unlocked may cause invalid modification to the region outside the
   unlocked area. This, in turn, may be part of a region locked by
   another client. Clients can avoid this situation by synchronously
   performing portions of write operations that overlap that portion
   (initial or final) that is not a full block. Similarly, invalidating
   a locked area which is not an integral number of full buffer blocks
   would require the client to read one or two partial blocks from the
   server if the revalidation procedure shows that the data which the

client possesses may not be valid.

The data that is written to the server as a prerequisite to the
unlocking of a region must be written, at the server, to stable
storage. The client may accomplish this either with synchronous
writes or by following asynchronous writes with a COMMIT operation.
This is required because retransmission of the modified data after a
server reboot might conflict with a lock held by another client.

A client implementation may choose to accommodate applications which
use byte-range locking in non-standard ways (e.g., using a byte-range
lock as a global semaphore) by flushing to the server more data upon
a LOCKU than is covered by the locked range. This may include
modified data within files other than the one for which the unlocks
are being done. In such cases, the client must not interfere with
applications whose READs and WRITEs are being done only within the
bounds of record locks which the application holds. For example, an
application locks a single byte of a file and proceeds to write that
single byte. A client that chose to handle a LOCKU by flushing all
modified data to the server could validly write that single byte in
response to an unrelated unlock. However, it would not be valid to
write the entire block in which that single written byte was located
since it includes an area that is not locked and might be locked by
another client. Client implementations can avoid this problem by
dividing files with modified data into those for which all
modifications are done to areas covered by an appropriate byte-range
lock and those for which there are modifications not covered by a
byte-range lock. Any writes done for the former class of files must
not include areas not locked and thus not modified on the client.

[10.3.3]. Data Caching and Mandatory File Locking

Client side data caching needs to respect mandatory file locking when
it is in effect. The presence of mandatory file locking for a given
file is indicated when the client gets back NFS4ERR_LOCKED from a
READ or WRITE on a file it has an appropriate share reservation for.
When mandatory locking is in effect for a file, the client must check
for an appropriate file lock for data being read or written. If a
lock exists for the range being read or written, the client may
satisfy the request using the client's validated cache. If an
appropriate file lock is not held for the range of the read or write,
the read or write request must not be satisfied by the client's cache
and the request must be sent to the server for processing. When a
read or write request partially overlaps a locked region, the request
should be subdivided into multiple pieces with each region (locked or
not) treated appropriately.

10.3.4. Data Caching and File Identity

When clients cache data, the file data needs to be organized
according to the file system object to which the data belongs. For
NFSv3 clients, the typical practice has been to assume for the
purpose of caching that distinct filehandles represent distinct file
system objects. The client then has the choice to organize and
maintain the data cache on this basis.

In the NFSv4 protocol, there is now the possibility to have
significant deviations from a "one filehandle per object" model
because a filehandle may be constructed on the basis of the object's
pathname. Therefore, clients need a reliable method to determine if
two filehandles designate the same file system object. If clients
were simply to assume that all distinct filehandles denote distinct
objects and proceed to do data caching on this basis, caching
inconsistencies would arise between the distinct client side objects
which mapped to the same server side object.

By providing a method to differentiate filehandles, the NFSv4
protocol alleviates a potential functional regression in comparison
with the NFSv3 protocol. Without this method, caching
inconsistencies within the same client could occur and this has not
been present in previous versions of the NFS protocol. Note that it
is possible to have such inconsistencies with applications executing
on multiple clients but that is not the issue being addressed here.

For the purposes of data caching, the following steps allow an NFSv4
client to determine whether two distinct filehandles denote the same
server side object:

o If GETATTR directed to two filehandles returns different values of
   the fsid attribute, then the filehandles represent distinct
   objects.

o If GETATTR for any file with an fsid that matches the fsid of the
   two filehandles in question returns a unique_handles attribute
   with a value of TRUE, then the two objects are distinct.

o If GETATTR directed to the two filehandles does not return the
   fileid attribute for both of the handles, then it cannot be
   determined whether the two objects are the same. Therefore,
   operations which depend on that knowledge (e.g., client side data
   caching) cannot be done reliably. Note that if GETATTR does not
   return the fileid attribute for both filehandles, it will return
   it for neither of the filehandles, since the fsid for both
   filehandles is the same.

   o If GETATTR directed to the two filehandles returns different
      values for the fileid attribute, then they are distinct objects.

   o Otherwise they are the same object.

10.4. Open Delegation

   When a file is being OPENed, the server may delegate further handling
   of opens and closes for that file to the opening client. Any such
   delegation is recallable, since the circumstances that allowed for
   the delegation are subject to change. In particular, the server may
   receive a conflicting OPEN from another client, the server must
   recall the delegation before deciding whether the OPEN from the other
   client may be granted. Making a delegation is up to the server and
   clients should not assume that any particular OPEN either will or
   will not result in an open delegation. The following is a typical
   set of conditions that servers might use in deciding whether OPEN
   should be delegated:

   o The client must be able to respond to the server's callback
      requests. The server will use the CB_NULL procedure for a test of
      callback ability.

   o The client must have responded properly to previous recalls.

   o There must be no current open conflicting with the requested
      delegation.

   o There should be no current delegation that conflicts with the
      delegation being requested.

   o The probability of future conflicting open requests should be low
      based on the recent history of the file.

   o The existence of any server-specific semantics of OPEN/CLOSE that
      would make the required handling incompatible with the prescribed
      handling that the delegated client would apply (see below).

   There are two types of open delegations, OPEN_DELEGATE_READ and
   OPEN_DELEGATE_WRITE. A OPEN_DELEGATE_READ delegation allows a client
   to handle, on its own, requests to open a file for reading that do
   not deny read access to others. It MUST, however, continue to send
   all requests to open a file for writing to the server. Multiple
   OPEN_DELEGATE_READ delegations may be outstanding simultaneously and
   do not conflict. A OPEN_DELEGATE_WRITE delegation allows the client
   to handle, on its own, all opens. Only one OPEN_DELEGATE_WRITE
   delegation may exist for a given file at a given time and it is
   inconsistent with any OPEN_DELEGATE_READ delegations.

When a single client holds a OPEN_DELEGATE_READ delegation, it is
assured that no other client may modify the contents or attributes of
the file. If more than one client holds an OPEN_DELEGATE_READ
delegation, then the contents and attributes of that file are not
allowed to change. When a client has an OPEN_DELEGATE_WRITE
delegation, it may modify the file data since no other client will be
accessing the file's data. The client holding a OPEN_DELEGATE_WRITE
delegation may only affect file attributes which are intimately
connected with the file data: size, time_modify, change.

When a client has an open delegation, it does not send OPENs or
CLOSEs to the server but updates the appropriate status internally.
For a OPEN_DELEGATE_READ delegation, opens that cannot be handled
locally (opens for write or that deny read access) must be sent to
the server.

When an open delegation is made, the response to the OPEN contains an
open delegation structure which specifies the following:

o   the type of delegation (read or write)

o   space limitation information to control flushing of data on close
    (OPEN_DELEGATE_WRITE delegation only, see Section 10.4.1)

o   an nfsace4 specifying read and write permissions

o   a stateid to represent the delegation for READ and WRITE

The delegation stateid is separate and distinct from the stateid for
the OPEN proper. The standard stateid, unlike the delegation
stateid, is associated with a particular open-owner and will continue
to be valid after the delegation is recalled and the file remains
open.

When a request internal to the client is made to open a file and open
delegation is in effect, it will be accepted or rejected solely on
the basis of the following conditions. Any requirement for other
checks to be made by the delegate should result in open delegation
being denied so that the checks can be made by the server itself.

o   The access and deny bits for the request and the file as described
    in Section 9.9.

o   The read and write permissions as determined below.

The nfsace4 passed with delegation can be used to avoid frequent
ACCESS calls. The permission check should be as follows:

o If the nfsace4 indicates that the open may be done, then it should
   be granted without reference to the server.

o If the nfsace4 indicates that the open may not be done, then an
   ACCESS request must be sent to the server to obtain the definitive
   answer.

The server may return an nfsace4 that is more restrictive than the
actual ACL of the file. This includes an nfsace4 that specifies
denial of all access. Note that some common practices such as
mapping the traditional user "root" to the user "nobody" may make it
incorrect to return the actual ACL of the file in the delegation
response.

The use of delegation together with various other forms of caching
creates the possibility that no server authentication will ever be
performed for a given user since all of the user's requests might be
satisfied locally. Where the client is depending on the server for
authentication, the client should be sure authentication occurs for
each user by use of the ACCESS operation. This should be the case
even if an ACCESS operation would not be required otherwise. As
mentioned before, the server may enforce frequent authentication by
returning an nfsace4 denying all access with every open delegation.

10.4.1. **Open Delegation and Data Caching**

   OPEN delegation allows much of the message overhead associated with
   the opening and closing files to be eliminated. An open when an open
   delegation is in effect does not require that a validation message be
   sent to the server unless there exists a potential for conflict with
   the requested share mode. The continued endurance of the
   "OPEN_DELEGATE_READ delegation" provides a guarantee that no OPEN for
   write and thus no write has occurred that did not originate from this
   client. Similarly, when closing a file opened for write and if
   OPEN_DELEGATE_WRITE delegation is in effect, the data written does
   not have to be flushed to the server until the open delegation is
   recalled. The continued endurance of the open delegation provides a
   guarantee that no open and thus no read or write has been done by
   another client.

   For the purposes of open delegation, READs and WRITEs done without an
   OPEN are treated as the functional equivalents of a corresponding
   type of OPEN. This refers to the READs and WRITEs that use the
   special stateids consisting of all zero bits or all one bits.
   Therefore, READs or WRITEs with a special stateid done by another
   client will force the server to recall a OPEN_DELEGATE_WRITE
   delegation. A WRITE with a special stateid done by another client
   will force a recall of OPEN_DELEGATE_READ delegations.

With delegations, a client is able to avoid writing data to the
server when the CLOSE of a file is serviced. The file close system
call is the usual point at which the client is notified of a lack of
stable storage for the modified file data generated by the
application. At the close, file data is written to the server and
through normal accounting the server is able to determine if the
available file system space for the data has been exceeded (i.e.,
server returns NFS4ERR_NOSPC or NFS4ERR_DQUOT). This accounting
includes quotas. The introduction of delegations requires that a
alternative method be in place for the same type of communication to
occur between client and server.

In the delegation response, the server provides either the limit of
the size of the file or the number of modified blocks and associated
block size. The server must ensure that the client will be able to
flush data to the server of a size equal to that provided in the
original delegation. The server must make this assurance for all
outstanding delegations. Therefore, the server must be careful in
its management of available space for new or modified data taking
into account available file system space and any applicable quotas.
The server can recall delegations as a result of managing the
available file system space. The client should abide by the server's
state space limits for delegations. If the client exceeds the stated
limits for the delegation, the server's behavior is undefined.

Based on server conditions, quotas or available file system space,
the server may grant OPEN_DELEGATE_WRITE delegations with very
restrictive space limitations. The limitations may be defined in a
way that will always force modified data to be flushed to the server
on close.

With respect to authentication, flushing modified data to the server
after a CLOSE has occurred may be problematic. For example, the user
of the application may have logged off the client and unexpired
authentication credentials may not be present. In this case, the
client may need to take special care to ensure that local unexpired
credentials will in fact be available. This may be accomplished by
tracking the expiration time of credentials and flushing data well in
advance of their expiration or by making private copies of
credentials to assure their availability when needed.

10.4.2. Open Delegation and File Locks

When a client holds a OPEN_DELEGATE_WRITE delegation, lock operations
may be performed locally. This includes those required for mandatory
file locking. This can be done since the delegation implies that
there can be no conflicting locks. Similarly, all of the
revalidations that would normally be associated with obtaining locks

and the flushing of data associated with the releasing of locks need
not be done.

When a client holds a OPEN_DELEGATE_READ delegation, lock operations
are not performed locally. All lock operations, including those
requesting non-exclusive locks, are sent to the server for
resolution.

10.4.3. Handling of CB_GETATTR

The server needs to employ special handling for a GETATTR where the
target is a file that has a OPEN_DELEGATE_WRITE delegation in effect.
The reason for this is that the client holding the
OPEN_DELEGATE_WRITE delegation may have modified the data and the
server needs to reflect this change to the second client that
submitted the GETATTR. Therefore, the client holding the
OPEN_DELEGATE_WRITE delegation needs to be interrogated. The server
will use the CB_GETATTR operation. The only attributes that the
server can reliably query via CB_GETATTR are size and change.

Since CB_GETATTR is being used to satisfy another client's GETATTR
request, the server only needs to know if the client holding the
delegation has a modified version of the file. If the client's copy
of the delegated file is not modified (data or size), the server can
satisfy the second client's GETATTR request from the attributes
stored locally at the server. If the file is modified, the server
only needs to know about this modified state. If the server
determines that the file is currently modified, it will respond to
the second client's GETATTR as if the file had been modified locally
at the server.

Since the form of the change attribute is determined by the server
and is opaque to the client, the client and server need to agree on a
method of communicating the modified state of the file. For the size
attribute, the client will report its current view of the file size.
For the change attribute, the handling is more involved.

For the client, the following steps will be taken when receiving a
OPEN_DELEGATE_WRITE delegation:

o The value of the change attribute will be obtained from the server
   and cached. Let this value be represented by c.

o The client will create a value greater than c that will be used
   for communicating modified data is held at the client. Let this
   value be represented by d.

o When the client is queried via CB_GETATTR for the change
   attribute, it checks to see if it holds modified data. If the
   file is modified, the value d is returned for the change attribute
   value. If this file is not currently modified, the client returns
   the value c for the change attribute.

For simplicity of implementation, the client MAY for each CB_GETATTR
return the same value d. This is true even if, between successive
CB_GETATTR operations, the client again modifies in the file's data
or metadata in its cache. The client can return the same value
because the only requirement is that the client be able to indicate
to the server that the client holds modified data. Therefore, the
value of d may always be c + 1.

While the change attribute is opaque to the client in the sense that
it has no idea what units of time, if any, the server is counting
change with, it is not opaque in that the client has to treat it as
an unsigned integer, and the server has to be able to see the results
of the client's changes to that integer. Therefore, the server MUST
encode the change attribute in network order when sending it to the
client. The client MUST decode it from network order to its native
order when receiving it and the client MUST encode it network order
when sending it to the server. For this reason, the change attribute
is defined as an unsigned integer rather than an opaque array of
bytes.

For the server, the following steps will be taken when providing a
OPEN_DELEGATE_WRITE delegation:

o Upon providing a OPEN_DELEGATE_WRITE delegation, the server will
   cache a copy of the change attribute in the data structure it uses
   to record the delegation. Let this value be represented by sc.

o When a second client sends a GETATTR operation on the same file to
   the server, the server obtains the change attribute from the first
   client. Let this value be cc.

o If the value cc is equal to sc, the file is not modified and the
   server returns the current values for change, time_metadata, and
   time_modify (for example) to the second client.

o If the value cc is NOT equal to sc, the file is currently modified
   at the first client and most likely will be modified at the server
   at a future time. The server then uses its current time to
   construct attribute values for time_metadata and time_modify. A
   new value of sc, which we will call nsc, is computed by the
   server, such that nsc >= sc + 1. The server then returns the
   constructed time_metadata, time_modify, and nsc values to the

      requester. The server replaces sc in the delegation record with
      nsc. To prevent the possibility of time_modify, time_metadata,
      and change from appearing to go backward (which would happen if
      the client holding the delegation fails to write its modified data
      to the server before the delegation is revoked or returned), the
      server SHOULD update the file's metadata record with the
      constructed attribute values. For reasons of reasonable
      performance, committing the constructed attribute values to stable
      storage is OPTIONAL.

   As discussed earlier in this section, the client MAY return the same
   cc value on subsequent CB_GETATTR calls, even if the file was
   modified in the client's cache yet again between successive
   CB_GETATTR calls. Therefore, the server must assume that the file
   has been modified yet again, and MUST take care to ensure that the
   new nsc it constructs and returns is greater than the previous nsc it
   returned. An example implementation's delegation record would
   satisfy this mandate by including a boolean field (let us call it
   "modified") that is set to FALSE when the delegation is granted, and
   an sc value set at the time of grant to the change attribute value.
   The modified field would be set to TRUE the first time cc != sc, and
   would stay TRUE until the delegation is returned or revoked. The
   processing for constructing nsc, time_modify, and time_metadata would
   use this pseudo code:

       if (!modified) {
           do CB_GETATTR for change and size;

           if (cc != sc)
               modified = TRUE;
       } else {
           do CB_GETATTR for size;
       }

       if (modified) {
           sc = sc + 1;
           time_modify = time_metadata = current_time;
           update sc, time_modify, time_metadata into file's metadata;
       }

   This would return to the client (that sent GETATTR) the attributes it
   requested, but make sure size comes from what CB_GETATTR returned.
   The server would not update the file's metadata with the client's
   modified size.

   In the case that the file attribute size is different than the
   server's current value, the server treats this as a modification
   regardless of the value of the change attribute retrieved via

CB_GETATTR and responds to the second client as in the last step.

This methodology resolves issues of clock differences between client
and server and other scenarios where the use of CB_GETATTR break
down.

It should be noted that the server is under no obligation to use
CB_GETATTR and therefore the server MAY simply recall the delegation
to avoid its use.

10.4.4. Recall of Open Delegation

The following events necessitate recall of an open delegation:

o Potentially conflicting OPEN request (or READ/WRITE done with
   "special" stateid)

o SETATTR issued by another client

o REMOVE request for the file

o RENAME request for the file as either source or target of the
      RENAME

Whether a RENAME of a directory in the path leading to the file
results in recall of an open delegation depends on the semantics of
the server file system. If that file system denies such RENAMEs when
a file is open, the recall must be performed to determine whether the
file in question is, in fact, open.

In addition to the situations above, the server may choose to recall
open delegations at any time if resource constraints make it
advisable to do so. Clients should always be prepared for the
possibility of recall.

When a client receives a recall for an open delegation, it needs to
update state on the server before returning the delegation. These
same updates must be done whenever a client chooses to return a
delegation voluntarily. The following items of state need to be
dealt with:

o If the file associated with the delegation is no longer open and
   no previous CLOSE operation has been sent to the server, a CLOSE
   operation must be sent to the server.

o If a file has other open references at the client, then OPEN
   operations must be sent to the server. The appropriate stateids
   will be provided by the server for subsequent use by the client

   since the delegation stateid will not longer be valid. These OPEN
   requests are done with the claim type of CLAIM_DELEGATE_CUR. This
   will allow the presentation of the delegation stateid so that the
   client can establish the appropriate rights to perform the OPEN.
   (see Section 15.18 for details.)

o If there are granted file locks, the corresponding LOCK operations
   need to be performed. This applies to the OPEN_DELEGATE_WRITE
   delegation case only.

o For a OPEN_DELEGATE_WRITE delegation, if at the time of recall the
   file is not open for write, all modified data for the file must be
   flushed to the server. If the delegation had not existed, the
   client would have done this data flush before the CLOSE operation.

o For a OPEN_DELEGATE_WRITE delegation when a file is still open at
   the time of recall, any modified data for the file needs to be
   flushed to the server.

o With the OPEN_DELEGATE_WRITE delegation in place, it is possible
   that the file was truncated during the duration of the delegation.
   For example, the truncation could have occurred as a result of an
   OPEN UNCHECKED4 with a size attribute value of zero. Therefore,
   if a truncation of the file has occurred and this operation has
   not been propagated to the server, the truncation must occur
   before any modified data is written to the server.

In the case of OPEN_DELEGATE_WRITE delegation, file locking imposes
some additional requirements. To precisely maintain the associated
invariant, it is required to flush any modified data in any region
for which a write lock was released while the OPEN_DELEGATE_WRITE
delegation was in effect. However, because the OPEN_DELEGATE_WRITE
delegation implies no other locking by other clients, a simpler
implementation is to flush all modified data for the file (as
described just above) if any write lock has been released while the
OPEN_DELEGATE_WRITE delegation was in effect.

An implementation need not wait until delegation recall (or deciding
to voluntarily return a delegation) to perform any of the above
actions, if implementation considerations (e.g., resource
availability constraints) make that desirable. Generally, however,
the fact that the actual open state of the file may continue to
change makes it not worthwhile to send information about opens and
closes to the server, except as part of delegation return. Only in
the case of closing the open that resulted in obtaining the
delegation would clients be likely to do this early, since, in that
case, the close once done will not be undone. Regardless of the
client's choices on scheduling these actions, all must be performed

before the delegation is returned, including (when applicable) the
close that corresponds to the open that resulted in the delegation.
These actions can be performed either in previous requests or in
previous operations in the same COMPOUND request.

10.4.5. OPEN Delegation Race with CB_RECALL

The server informs the client of recall via a CB_RECALL. A race case
which may develop is when the delegation is immediately recalled
before the COMPOUND which established the delegation is returned to
the client. As the CB_RECALL provides both a stateid and a
filehandle for which the client has no mapping, it cannot honor the
recall attempt. At this point, the client has two choices, either do
not respond or respond with NFS4ERR_BADHANDLE. If it does not
respond, then it runs the risk of the server deciding to not grant it
further delegations.

If instead it does reply with NFS4ERR_BADHANDLE, then both the client
and the server might be able to detect that a race condition is
occurring. The client can keep a list of pending delegations. When
it receives a CB_RECALL for an unknown delegation, it can cache the
stateid and filehandle on a list of pending recalls. When it is
provided with a delegation, it would only use it if it was not on the
pending recall list. Upon the next CB_RECALL, it could immediately
return the delegation.

In turn, the server can keep track of when it issues a delegation and
assume that if a client responds to the CB_RECALL with a
NFS4ERR_BADHANDLE, then the client has yet to receive the delegation.
The server SHOULD give the client a reasonable time both to get this
delegation and to return it before revoking the delegation. Unlike a
failed callback path, the server should periodically probe the client
with CB_RECALL to see if it has received the delegation and is ready
to return it.

When the server finally determines that enough time has lapsed, it
SHOULD revoke the delegation and it SHOULD NOT revoke the lease.
During this extended recall process, the server SHOULD be renewing
the client lease. The intent here is that the client not pay too
onerous a burden for a condition caused by the server.

10.4.6. Clients that Fail to Honor Delegation Recalls

A client may fail to respond to a recall for various reasons, such as
a failure of the callback path from server to the client. The client
may be unaware of a failure in the callback path. This lack of
awareness could result in the client finding out long after the
failure that its delegation has been revoked, and another client has

modified the data for which the client had a delegation. This is
especially a problem for the client that held a OPEN_DELEGATE_WRITE
delegation.

The server also has a dilemma in that the client that fails to
respond to the recall might also be sending other NFS requests,
including those that renew the lease before the lease expires.
Without returning an error for those lease renewing operations, the
server leads the client to believe that the delegation it has is in
force.

This difficulty is solved by the following rules:

o When the callback path is down, the server MUST NOT revoke the
   delegation if one of the following occurs:

   * The client has issued a RENEW operation and the server has
      returned an NFS4ERR_CB_PATH_DOWN error. The server MUST renew
      the lease for any byte-range locks and share reservations the
      client has that the server has known about (as opposed to those
      locks and share reservations the client has established but not
      yet sent to the server, due to the delegation). The server
      SHOULD give the client a reasonable time to return its
      delegations to the server before revoking the client's
      delegations.

   * The client has not issued a RENEW operation for some period of
      time after the server attempted to recall the delegation. This
      period of time MUST NOT be less than the value of the
      lease_time attribute.

o When the client holds a delegation, it cannot rely on operations,
   except for RENEW, that take a stateid, to renew delegation leases
   across callback path failures. The client that wants to keep
   delegations in force across callback path failures must use RENEW
   to do so.

10.4.7. Delegation Revocation

At the point a delegation is revoked, if there are associated opens
on the client, the applications holding these opens need to be
notified. This notification usually occurs by returning errors for
READ/WRITE operations or when a close is attempted for the open file.

If no opens exist for the file at the point the delegation is
revoked, then notification of the revocation is unnecessary.
However, if there is modified data present at the client for the
file, the user of the application should be notified. Unfortunately,

it may not be possible to notify the user since active applications
may not be present at the client. See Section 10.5.1 for additional
details.

10.5. Data Caching and Revocation

When locks and delegations are revoked, the assumptions upon which
successful caching depend are no longer guaranteed. For any locks or
share reservations that have been revoked, the corresponding owner
needs to be notified. This notification includes applications with a
file open that has a corresponding delegation which has been revoked.
Cached data associated with the revocation must be removed from the
client. In the case of modified data existing in the client's cache,
that data must be removed from the client without it being written to
the server. As mentioned, the assumptions made by the client are no
longer valid at the point when a lock or delegation has been revoked.
For example, another client may have been granted a conflicting lock
after the revocation of the lock at the first client. Therefore, the
data within the lock range may have been modified by the other
client. Obviously, the first client is unable to guarantee to the
application what has occurred to the file in the case of revocation.

Notification to a lock owner will in many cases consist of simply
returning an error on the next and all subsequent READs/WRITEs to the
open file or on the close. Where the methods available to a client
make such notification impossible because errors for certain
operations may not be returned, more drastic action such as signals
or process termination may be appropriate. The justification for
this is that an invariant for which an application depends on may be
violated. Depending on how errors are typically treated for the
client operating environment, further levels of notification
including logging, console messages, and GUI pop-ups may be
appropriate.

10.5.1. Revocation Recovery for Write Open Delegation

Revocation recovery for a OPEN_DELEGATE_WRITE delegation poses the
special issue of modified data in the client cache while the file is
not open. In this situation, any client which does not flush
modified data to the server on each close must ensure that the user
receives appropriate notification of the failure as a result of the
revocation. Since such situations may require human action to
correct problems, notification schemes in which the appropriate user
or administrator is notified may be necessary. Logging and console
messages are typical examples.

If there is modified data on the client, it must not be flushed
normally to the server. A client may attempt to provide a copy of

the file data as modified during the delegation under a different
name in the file system name space to ease recovery. Note that when
the client can determine that the file has not been modified by any
other client, or when the client has a complete cached copy of the
file in question, such a saved copy of the client's view of the file
may be of particular value for recovery. In other cases, recovery
using a copy of the file based partially on the client's cached data
and partially on the server copy as modified by other clients, will
be anything but straightforward, so clients may avoid saving file
contents in these situations or mark the results specially to warn
users of possible problems.

Saving of such modified data in delegation revocation situations may
be limited to files of a certain size or might be used only when
sufficient disk space is available within the target file system.
Such saving may also be restricted to situations when the client has
sufficient buffering resources to keep the cached copy available
until it is properly stored to the target file system.

10.6. Attribute Caching

The attributes discussed in this section do not include named
attributes. Individual named attributes are analogous to files and
caching of the data for these needs to be handled just as data
caching is for regular files. Similarly, LOOKUP results from an
OPENATTR directory are to be cached on the same basis as any other
pathnames and similarly for directory contents.

Clients may cache file attributes obtained from the server and use
them to avoid subsequent GETATTR requests. Such caching is write
through in that modification to file attributes is always done by
means of requests to the server and should not be done locally and
cached. The exception to this are modifications to attributes that
are intimately connected with data caching. Therefore, extending a
file by writing data to the local data cache is reflected immediately
in the size as seen on the client without this change being
immediately reflected on the server. Normally such changes are not
propagated directly to the server but when the modified data is
flushed to the server, analogous attribute changes are made on the
server. When open delegation is in effect, the modified attributes
may be returned to the server in the response to a CB_GETATTR call.

The result of local caching of attributes is that the attribute
caches maintained on individual clients will not be coherent.
Changes made in one order on the server may be seen in a different
order on one client and in a third order on a different client.

The typical file system application programming interfaces do not

provide means to atomically modify or interrogate attributes for
multiple files at the same time. The following rules provide an
environment where the potential incoherency mentioned above can be
reasonably managed. These rules are derived from the practice of
previous NFS protocols.

o All attributes for a given file (per-fsid attributes excepted) are
   cached as a unit at the client so that no non-serializability can
   arise within the context of a single file.

o An upper time boundary is maintained on how long a client cache
   entry can be kept without being refreshed from the server.

o When operations are performed that modify attributes at the
   server, the updated attribute set is requested as part of the
   containing RPC. This includes directory operations that update
   attributes indirectly. This is accomplished by following the
   modifying operation with a GETATTR operation and then using the
   results of the GETATTR to update the client's cached attributes.

Note that if the full set of attributes to be cached is requested by
READDIR, the results can be cached by the client on the same basis as
attributes obtained via GETATTR.

A client may validate its cached version of attributes for a file by
fetching just both the change and time_access attributes and assuming
that if the change attribute has the same value as it did when the
attributes were cached, then no attributes other than time_access
have changed. The reason why time_access is also fetched is because
many servers operate in environments where the operation that updates
change does not update time_access. For example, POSIX file
semantics do not update access time when a file is modified by the
write system call. Therefore, the client that wants a current
time_access value should fetch it with change during the attribute
cache validation processing and update its cached time_access.

The client may maintain a cache of modified attributes for those
attributes intimately connected with data of modified regular files
(size, time_modify, and change). Other than those three attributes,
the client MUST NOT maintain a cache of modified attributes.
Instead, attribute changes are immediately sent to the server.

In some operating environments, the equivalent to time_access is
expected to be implicitly updated by each read of the content of the
file object. If an NFS client is caching the content of a file
object, whether it is a regular file, directory, or symbolic link,
the client SHOULD NOT update the time_access attribute (via SETATTR
or a small READ or READDIR request) on the server with each read that

   is satisfied from cache. The reason is that this can defeat the
   performance benefits of caching content, especially since an explicit
   SETATTR of time_access may alter the change attribute on the server.
   If the change attribute changes, clients that are caching the content
   will think the content has changed, and will re-read unmodified data
   from the server. Nor is the client encouraged to maintain a modified
   version of time_access in its cache, since this would mean that the
   client will either eventually have to write the access time to the
   server with bad performance effects, or it would never update the
   server's time_access, thereby resulting in a situation where an
   application that caches access time between a close and open of the
   same file observes the access time oscillating between the past and
   present. The time_access attribute always means the time of last
   access to a file by a read that was satisfied by the server. This
   way clients will tend to see only time_access changes that go forward
   in time.

10.7. **Data and Metadata Caching and Memory Mapped Files**

   Some operating environments include the capability for an application
   to map a file's content into the application's address space. Each
   time the application accesses a memory location that corresponds to a
   block that has not been loaded into the address space, a page fault
   occurs and the file is read (or if the block does not exist in the
   file, the block is allocated and then instantiated in the
   application's address space).

   As long as each memory mapped access to the file requires a page
   fault, the relevant attributes of the file that are used to detect
   access and modification (time_access, time_metadata, time_modify, and
   change) will be updated. However, in many operating environments,
   when page faults are not required these attributes will not be
   updated on reads or updates to the file via memory access (regardless
   of whether the file is a local file or is being access remotely). A
   client or server MAY fail to update attributes of a file that is
   being accessed via memory mapped I/O. This has several implications:

   o If there is an application on the server that has memory mapped a
      file that a client is also accessing, the client may not be able
      to get a consistent value of the change attribute to determine
      whether its cache is stale or not. A server that knows that the
      file is memory mapped could always pessimistically return updated
      values for change so as to force the application to always get the
      most up to date data and metadata for the file. However, due to
      the negative performance implications of this, such behavior is
      OPTIONAL.

o If the memory mapped file is not being modified on the server, and
   instead is just being read by an application via the memory mapped
   interface, the client will not see an updated time_access
   attribute. However, in many operating environments, neither will
   any process running on the server. Thus NFS clients are at no
   disadvantage with respect to local processes.

o If there is another client that is memory mapping the file, and if
   that client is holding a OPEN_DELEGATE_WRITE delegation, the same
   set of issues as discussed in the previous two bullet items apply.
   So, when a server does a CB_GETATTR to a file that the client has
   modified in its cache, the response from CB_GETATTR will not
   necessarily be accurate. As discussed earlier, the client's
   obligation is to report that the file has been modified since the
   delegation was granted, not whether it has been modified again
   between successive CB_GETATTR calls, and the server MUST assume
   that any file the client has modified in cache has been modified
   again between successive CB_GETATTR calls. Depending on the
   nature of the client's memory management system, this weak
   obligation may not be possible. A client MAY return stale
   information in CB_GETATTR whenever the file is memory mapped.

o The mixture of memory mapping and file locking on the same file is
   problematic. Consider the following scenario, where the page size
   on each client is 8192 bytes.

   * Client A memory maps first page (8192 bytes) of file X

   * Client B memory maps first page (8192 bytes) of file X

   * Client A write locks first 4096 bytes

   * Client B write locks second 4096 bytes

   * Client A, via a STORE instruction modifies part of its locked
      region.

   * Simultaneous to client A, client B issues a STORE on part of
      its locked region.

Here the challenge is for each client to resynchronize to get a
correct view of the first page. In many operating environments, the
virtual memory management systems on each client only know a page is
modified, not that a subset of the page corresponding to the
respective lock regions has been modified. So it is not possible for
each client to do the right thing, which is to only write to the
server that portion of the page that is locked. For example, if
client A simply writes out the page, and then client B writes out the

page, client A's data is lost.

Moreover, if mandatory locking is enabled on the file, then we have a
different problem. When clients A and B issue the STORE
instructions, the resulting page faults require a byte-range lock on
the entire page. Each client then tries to extend their locked range
to the entire page, which results in a deadlock.

Communicating the NFS4ERR_DEADLOCK error to a STORE instruction is
difficult at best.

If a client is locking the entire memory mapped file, there is no
problem with advisory or mandatory byte-range locking, at least until
the client unlocks a region in the middle of the file.

Given the above issues the following are permitted:

o Clients and servers MAY deny memory mapping a file they know there
   are byte-range locks for.

o Clients and servers MAY deny a byte-range lock on a file they know
   is memory mapped.

o A client MAY deny memory mapping a file that it knows requires
   mandatory locking for I/O. If mandatory locking is enabled after
   the file is opened and mapped, the client MAY deny the application
   further access to its mapped file.

10.8. Name Caching

The results of LOOKUP and READDIR operations may be cached to avoid
the cost of subsequent LOOKUP operations. Just as in the case of
attribute caching, inconsistencies may arise among the various client
caches. To mitigate the effects of these inconsistencies and given
the context of typical file system APIs, an upper time boundary is
maintained on how long a client name cache entry can be kept without
verifying that the entry has not been made invalid by a directory
change operation performed by another client.

When a client is not making changes to a directory for which there
exist name cache entries, the client needs to periodically fetch
attributes for that directory to ensure that it is not being
modified. After determining that no modification has occurred, the
expiration time for the associated name cache entries may be updated
to be the current time plus the name cache staleness bound.

When a client is making changes to a given directory, it needs to
determine whether there have been changes made to the directory by

other clients. It does this by using the change attribute as
reported before and after the directory operation in the associated
change_info4 value returned for the operation. The server is able to
communicate to the client whether the change_info4 data is provided
atomically with respect to the directory operation. If the change
values are provided atomically, the client is then able to compare
the pre-operation change value with the change value in the client's
name cache. If the comparison indicates that the directory was
updated by another client, the name cache associated with the
modified directory is purged from the client. If the comparison
indicates no modification, the name cache can be updated on the
client to reflect the directory operation and the associated timeout
extended. The post-operation change value needs to be saved as the
basis for future change_info4 comparisons.

As demonstrated by the scenario above, name caching requires that the
client revalidate name cache data by inspecting the change attribute
of a directory at the point when the name cache item was cached.
This requires that the server update the change attribute for
directories when the contents of the corresponding directory is
modified. For a client to use the change_info4 information
appropriately and correctly, the server must report the pre and post
operation change attribute values atomically. When the server is
unable to report the before and after values atomically with respect
to the directory operation, the server must indicate that fact in the
change_info4 return value. When the information is not atomically
reported, the client should not assume that other clients have not
changed the directory.

10.9. Directory Caching

The results of READDIR operations may be used to avoid subsequent
READDIR operations. Just as in the cases of attribute and name
caching, inconsistencies may arise among the various client caches.
To mitigate the effects of these inconsistencies, and given the
context of typical file system APIs, the following rules should be
followed:

o Cached READDIR information for a directory which is not obtained
   in a single READDIR operation must always be a consistent snapshot
   of directory contents. This is determined by using a GETATTR
   before the first READDIR and after the last of READDIR that
   contributes to the cache.

o An upper time boundary is maintained to indicate the length of
   time a directory cache entry is considered valid before the client
   must revalidate the cached information.

The revalidation technique parallels that discussed in the case of
name caching. When the client is not changing the directory in
question, checking the change attribute of the directory with GETATTR
is adequate. The lifetime of the cache entry can be extended at
these checkpoints. When a client is modifying the directory, the
client needs to use the change_info4 data to determine whether there
are other clients modifying the directory. If it is determined that
no other client modifications are occurring, the client may update
its directory cache to reflect its own changes.

As demonstrated previously, directory caching requires that the
client revalidate directory cache data by inspecting the change
attribute of a directory at the point when the directory was cached.
This requires that the server update the change attribute for
directories when the contents of the corresponding directory is
modified. For a client to use the change_info4 information
appropriately and correctly, the server must report the pre and post
operation change attribute values atomically. When the server is
unable to report the before and after values atomically with respect
to the directory operation, the server must indicate that fact in the
change_info4 return value. When the information is not atomically
reported, the client should not assume that other clients have not
changed the directory.

11. Minor Versioning

To address the requirement of an NFS protocol that can evolve as the
need arises, the NFSv4 protocol contains the rules and framework to
allow for future minor changes or versioning.

The base assumption with respect to minor versioning is that any
future accepted minor version must follow the IETF process and be
documented in a standards track RFC. Therefore, each minor version
number will correspond to an RFC. Minor version 0 of the NFS version
4 protocol is represented by this RFC. The COMPOUND and CB_COMPOUND
procedures support the encoding of the minor version being requested
by the client.

The following items represent the basic rules for the development of
minor versions. Note that a future minor version may decide to
modify or add to the following rules as part of the minor version
definition.

1.   Procedures are not added or deleted

     To maintain the general RPC model, NFSv4 minor versions will not
     add to or delete procedures from the NFS program.

2.   Minor versions may add operations to the COMPOUND and
     CB_COMPOUND procedures.

     The addition of operations to the COMPOUND and CB_COMPOUND
     procedures does not affect the RPC model.

     1. Minor versions may append attributes to the bitmap4 that
         represents sets of attributes and to the fattr4 that
         represents sets of attribute values.

         This allows for the expansion of the attribute model to
         allow for future growth or adaptation.

     2. Minor version X must append any new attributes after the
         last documented attribute.

         Since attribute results are specified as an opaque array of
         per-attribute XDR encoded results, the complexity of adding
         new attributes in the midst of the current definitions would
         be too burdensome.

3.   Minor versions must not modify the structure of an existing
     operation's arguments or results.

     Again, the complexity of handling multiple structure definitions
     for a single operation is too burdensome. New operations should
     be added instead of modifying existing structures for a minor
     version.

     This rule does not preclude the following adaptations in a minor
     version.

     * adding bits to flag fields, such as new attributes to
        GETATTR's bitmap4 data type, and providing corresponding
        variants of opaque arrays, such as a notify4 used together
        with such bitmaps

     * adding bits to existing attributes like ACLs that have flag
        words

     * extending enumerated types (including NFS4ERR_*) with new
        values

4.   Minor versions must not modify the structure of existing
     attributes.

5.   Minor versions must not delete operations.

This prevents the potential reuse of a particular operation
"slot" in a future minor version.

6. Minor versions must not delete attributes.

7. Minor versions must not delete flag bits or enumeration values.

8. Minor versions may declare an operation MUST NOT be implemented.

    Specifying that an operation MUST NOT be implemented is
    equivalent to obsoleting an operation. For the client, it means
    that the operation MUST NOT be sent to the server. For the
    server, an NFS error can be returned as opposed to "dropping"
    the request as an XDR decode error. This approach allows for
    the obsolescence of an operation while maintaining its structure
    so that a future minor version can reintroduce the operation.

    1. Minor versions may declare that an attribute MUST NOT be
        implemented.

    2. Minor versions may declare that a flag bit or enumeration
        value MUST NOT be implemented.

9. Minor versions may downgrade features from REQUIRED to
    RECOMMENDED, or RECOMMENDED to OPTIONAL.

10. Minor versions may upgrade features from OPTIONAL to RECOMMENDED
    or RECOMMENDED to REQUIRED.

11. A client and server that support minor version X SHOULD support
    minor versions 0 through X-1 as well.

12. Except for infrastructural changes, no new features may be
    introduced as REQUIRED in a minor version.

    This rule allows for the introduction of new functionality and
    forces the use of implementation experience before designating a
    feature as REQUIRED. On the other hand, some classes of
    features are infrastructural and have broad effects. Allowing
    infrastructural features to be RECOMMENDED or OPTIONAL
    complicates implementation of the minor version.

13. A client MUST NOT attempt to use a stateid, filehandle, or
    similar returned object from the COMPOUND procedure with minor
    version X for another COMPOUND procedure with minor version Y,
    where X != Y.

12. Internationalization

12.1. Introduction

   This section describes NFSv4.0 internationalization as implemented by
   existing clients and servers. As a result, the keywords "MUST",
   "SHOULD", and "MAY", even though they retain their normal meanings,
   reflect patterns of existing implementation:

   o Behavior implemented by all existing clients or servers is
      described using "MUST", since new implementations need to follow
      existing ones to be assured of interoperability.

   o Behavior implemented by no existing clients or servers is
      described using "MUST NOT", if such behavior poses
      interoperability problems.

   o Behavior implemented by most existing clients or servers, where
      that behavior is more desirable than any alternative is described
      using "SHOULD", since new implementations need to follow that
      existing practice unless there are strong reasons to do otherwise.
      This also holds for "SHOULD NOT".

   o Behavior implemented by some not all existing clients or servers,
      is described using "MAY", indicating that new implementations have
      a choice as to whether they will behave in that way.

   o Behavior implemented by all existing clients or servers, so far as
      is known, but where there remains some uncertainty as to details
      is described using "should". Such cases primary concern details
      of error returns. New implementations should follow existing
      practice even though such situations generally do not affect
      interoperability.

12.2. String Encoding

   Strings that potentially contain non-ASCII Characters are represented
   in NFSv4 using the UTF-8 encoding of Unicode. See [RFC2279] for
   precise encoding and decoding rules.

   Some details of the protocol treatment depend on the type of string:

   o For strings that are component names, any non-ASCII characters
      MUST be represented using the UTF-8 encoding of Unicode.

   o For strings whose form is defined by other internet standards,
      non-ASCII characters MUST be represented using the UTF-8 encoding
      of Unicode. In addition other sorts of restrictions defined by

those standards need to be addressed. See section 11.4 for
details.

o For other sorts of strings, any non-ASCII characters SHOULD be
   represented using the UTF-8 encoding of Unicode.

12.3. Normalization

The client and server operating environments may differ in their
policies and operational methods with respect to character
normalization (See [Unicode1] for a discussion of normalization
forms). This difference may also exist between applications on the
same client. This adds to the difficulty of providing a single
normalization policy for the protocol that allows for maximal
interoperability. This issue is similar to the character case issues
where the server may or may not support case insensitive file name
matching and may or may not preserve the character case when storing
file names. The protocol does not mandate a particular behavior but
allows for a range of useful behaviors.

The NFS version 4 protocol does not mandate the use of a particular
normalization form at this time. A subsequent minor version of the
NFSv4 protocol might specify a particular normalization form.
Therefore, the server and client can expect that they may receive
unnormalized characters within protocol requests and responses. If
the operating environment requires normalization, then the
implementation must normalize the various UTF-8 encoded strings
within the protocol before presenting the information to an
application (at the client) or local file system (at the server).

Server implementations MAY normalize file names to conform to a
particular normalization form before using the resulting string when
looking up or creating a file. Servers MAY also perform
normalization-insensitive string comparisons without modifying name
to match a particular normalization form. Servers MUST NOT reject a
file name because it doesn't a conform to a particular normalization
form.

12.4. Types with Processing Defined by Other Internet Areas

There are two types of strings which NFSv4 deals with whose
processing is defined by other Internet standards, and where issues
related to different handling choices by server operating systems or
server file systems do not apply.

These are as follows:

   o Server names as they appear in the fs_locations attribute. Note
      that for most purposes, such server names will only be sent by the
      server to the client. The exception is use of the fs_locations
      attribute in a VERIFY or NVERIFY operation.

   o Principal suffixes which are used to denote sets of users and
      groups, and are in the form of domain names.

   The general rules for handling all of these domain-related strings
   are similar and independent of role the of the sender or receiver as
   client or server although the consequences of failure to obey these
   rules may be different for client or server. The server can report
   errors when it is sent invalid strings, whereas the client will
   simply ignore invalid string or use a default value in their place.

   The string sent SHOULD be in the form of a U-label although it MAY be
   in the form of an A-label or a UTF-8 string that would not map to
   itself when canonicalized by applying ToUnicode(ToASCII(...)). The
   receiver needs to be able to accept domain and server names in any of
   the formats allowed. The server MUST reject, using the error
   NFS4ERR_INVAL, a string which is not valid UTF-8 or which begins with
   "xn--" and violates the rules for a valid A-label.

   When a domain string is part of id@domain or group@domain, the server
   SHOULD map domain strings which are A-labels or are UTF-8 domain
   names which are not U-labels, to the corresponding U-label, using
   ToUnicode(domain) or ToUnicode(ToASCII(domain)). As a result, the
   domain name returned within a userid on a GETATTR may not match that
   sent when the userid is set using SETATTR, although when this
   happens, the domain will be in the form of a U-label. When the
   server does not map domain strings which are not U-labels into a
   U-label, which it MAY do, it MUST NOT modify the domain and the
   domain returned on a GETATTR of the userid MUST be the same as that
   used when setting the userid by the SETATTR.

   The server MAY implement VERIFY and NVERIFY without translating
   internal state to a string form, so that, for example, a user
   principal which represents a specific numeric user id, will match a
   different principal string which represents the same numeric user id.

12.5. UTF-8 Related Errors

   Where the client sends an invalid UTF-8 string, the server SHOULD
   return an NFS4ERR_INVAL error. This includes cases in which
   inappropriate prefixes are detected and where the count includes
   trailing bytes that do not constitute a full UCS character.

   Where the client supplied string is not rejected with NFS4ERR_INVAL

but contains characters that are not supported by the server as a
value for that string (e.g. names containing characters that have
more than two octets on a file system that supports Unicode
characters only), the server should return an NFS4ERR_BADCHAR error.

Where a UTF-8 string is used as a file name, and the file system,
while supporting all of the characters within the name, does not
allow that particular name to be used, the error should return the
error NFS4ERR_BADNAME. This includes situations in which the server
file system imposes a normalization constraint on name strings, but
will also include such situations as file system prohibitions of "."
and ".." as file names for certain operations, and other such
constraints.

13. Error Values

NFS error numbers are assigned to failed operations within a Compound
(COMPOUND or CB_COMPOUND) request. A Compound request contains a
number of NFS operations that have their results encoded in sequence
in a Compound reply. The results of successful operations will
consist of an NFS4_OK status followed by the encoded results of the
operation. If an NFS operation fails, an error status will be
entered in the reply and the Compound request will be terminated.

13.1. Error Definitions

                  Protocol Error Definitions

Error	Number	Description
NFS4_OK	0	Section 13.1.3.1
NFS4ERR_ACCESS	13	Section 13.1.6.1
NFS4ERR_ATTRNOTSUPP	10032	Section 13.1.11.1
NFS4ERR_ADMIN_REVOKED	10047	Section 13.1.5.1
NFS4ERR_BADCHAR	10040	Section 13.1.7.1
NFS4ERR_BADHANDLE	10001	Section 13.1.2.1
NFS4ERR_BADNAME	10041	Section 13.1.7.2
NFS4ERR_BADOWNER	10039	Section 13.1.11.2
NFS4ERR_BADTYPE	10007	Section 13.1.4.1
NFS4ERR_BADXDR	10036	Section 13.1.1.1
NFS4ERR_BAD_COOKIE	10003	Section 13.1.1.2
NFS4ERR_BAD_RANGE	10042	Section 13.1.8.1
NFS4ERR_BAD_SEQID	10026	Section 13.1.8.2
NFS4ERR_BAD_STATEID	10025	Section 13.1.5.2
NFS4ERR_CLID_INUSE	10017	Section 13.1.10.1
NFS4ERR_DEADLOCK	10045	Section 13.1.8.3

NFS4ERR_DELAY	10008	Section 13.1.1.3
NFS4ERR_DENIED	10010	Section 13.1.8.4
NFS4ERR_DQUOT	69	Section 13.1.4.2
NFS4ERR_EXIST	17	Section 13.1.4.3
NFS4ERR_EXPIRED	10011	Section 13.1.5.3
NFS4ERR_FBIG	27	Section 13.1.4.4
NFS4ERR_FHEXPIRED	10014	Section 13.1.2.2
NFS4ERR_FILE_OPEN	10046	Section 13.1.4.5
NFS4ERR_GRACE	10013	Section 13.1.9.1
NFS4ERR_INVAL	22	Section 13.1.1.4
NFS4ERR_IO	5	Section 13.1.4.6
NFS4ERR_ISDIR	21	Section 13.1.2.3
NFS4ERR_LEASE_MOVED	10031	Section 13.1.5.4
NFS4ERR_LOCKED	10012	Section 13.1.8.5
NFS4ERR_LOCKS_HELD	10037	Section 13.1.8.6
NFS4ERR_LOCK_NOTSUPP	10043	Section 13.1.8.7
NFS4ERR_LOCK_RANGE	10028	Section 13.1.8.8
NFS4ERR_MINOR_VERS_MISMATCH	10021	Section 13.1.3.2
NFS4ERR_MLINK	31	Section 13.1.4.7
NFS4ERR_MOVED	10019	Section 13.1.2.4
NFS4ERR_NAMETOOLONG	63	Section 13.1.7.3
NFS4ERR_NOENT	2	Section 13.1.4.8
NFS4ERR_NOFILEHANDLE	10020	Section 13.1.2.5
NFS4ERR_NOSPC	28	Section 13.1.4.9
NFS4ERR_NOTDIR	20	Section 13.1.2.6
NFS4ERR_NOTEMPTY	66	Section 13.1.4.10
NFS4ERR_NOTSUPP	10004	Section 13.1.1.5
NFS4ERR_NOT_SAME	10027	Section 13.1.11.3
NFS4ERR_NO_GRACE	10033	Section 13.1.9.2
NFS4ERR_NXIO	6	Section 13.1.4.11
NFS4ERR_OLD_STATEID	10024	Section 13.1.5.5
NFS4ERR_OPENMODE	10038	Section 13.1.8.9
NFS4ERR_OP_ILLEGAL	10044	Section 13.1.3.3
NFS4ERR_PERM	1	Section 13.1.6.2
NFS4ERR_RECLAIM_BAD	10034	Section 13.1.9.3
NFS4ERR_RECLAIM_CONFLICT	10035	Section 13.1.9.4
NFS4ERR_RESOURCE	10018	Section 13.1.3.4
NFS4ERR_RESTOREFH	10030	Section 13.1.4.12
NFS4ERR_ROFS	30	Section 13.1.4.13
NFS4ERR_SAME	10009	Section 13.1.11.4
NFS4ERR_SERVERFAULT	10006	Section 13.1.1.6
NFS4ERR_STALE	70	Section 13.1.2.7
NFS4ERR_STALE_CLIENTID	10022	Section 13.1.10.2
NFS4ERR_STALE_STATEID	10023	Section 13.1.5.6
NFS4ERR_SYMLINK	10029	Section 13.1.2.8
NFS4ERR_TOOSMALL	10005	Section 13.1.1.7
NFS4ERR_WRONGSEC	10016	Section 13.1.6.3

```
   | NFS4ERR_XDEV                  | 18     | Section 13.1.4.14 |
   +------------------------------+--------+-------------------+
```
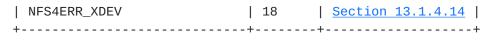
Table 5

13.1.1. General Errors

This section deals with errors that are applicable to a broad set of
different purposes.

13.1.1.1. NFS4ERR_BADXDR (Error Code 10036)

The arguments for this operation do not match those specified in the
XDR definition. This includes situations in which the request ends
before all the arguments have been seen. Note that this error
applies when fixed enumerations (these include booleans) have a value
within the input stream which is not valid for the enum. A replier
may pre-parse all operations for a Compound procedure before doing
any operation execution and return RPC-level XDR errors in that case.

13.1.1.2. NFS4ERR_BAD_COOKIE (Error Code 10003)

Used for operations that provide a set of information indexed by some
quantity provided by the client or cookie sent by the server for an
earlier invocation. Where the value cannot be used for its intended
purpose, this error results.

13.1.1.3. NFS4ERR_DELAY (Error Code 10008)

For any of a number of reasons, the replier could not process this
operation in what was deemed a reasonable time. The client should
wait and then try the request with a new RPC transaction ID.

Some example of situations that might lead to this situation:

o A server that supports hierarchical storage receives a request to
   process a file that had been migrated.

o An operation requires a delegation recall to proceed and waiting
   for this delegation recall makes processing this request in a
   timely fashion impossible.

13.1.1.4. NFS4ERR_INVAL (Error Code 22)

The arguments for this operation are not valid for some reason, even
though they do match those specified in the XDR definition for the
request.

13.1.1.5. NFS4ERR_NOTSUPP (Error Code 10004)

   Operation not supported, either because the operation is an OPTIONAL
   one and is not supported by this server or because the operation MUST
   NOT be implemented in the current minor version.

13.1.1.6. NFS4ERR_SERVERFAULT (Error Code 10006)

   An error occurred on the server which does not map to any of the
   specific legal NFSv4 protocol error values. The client should
   translate this into an appropriate error. UNIX clients may choose to
   translate this to EIO.

13.1.1.7. NFS4ERR_TOOSMALL (Error Code 10005)

   Used where an operation returns a variable amount of data, with a
   limit specified by the client. Where the data returned cannot be
   fitted within the limit specified by the client, this error results.

13.1.2. Filehandle Errors

   These errors deal with the situation in which the current or saved
   filehandle, or the filehandle passed to PUTFH intended to become the
   current filehandle, is invalid in some way. This includes situations
   in which the filehandle is a valid filehandle in general but is not
   of the appropriate object type for the current operation.

   Where the error description indicates a problem with the current or
   saved filehandle, it is to be understood that filehandles are only
   checked for the condition if they are implicit arguments of the
   operation in question.

13.1.2.1. NFS4ERR_BADHANDLE (Error Code 10001)

   Illegal NFS filehandle for the current server. The current
   filehandle failed internal consistency checks. Once accepted as
   valid (by PUTFH), no subsequent status change can cause the
   filehandle to generate this error.

13.1.2.2. NFS4ERR_FHEXPIRED (Error Code 10014)

   A current or saved filehandle which is an argument to the current
   operation is volatile and has expired at the server.

13.1.2.3. NFS4ERR_ISDIR (Error Code 21)

   The current or saved filehandle designates a directory when the
   current operation does not allow a directory to be accepted as the

target of this operation.

13.1.2.4. NFS4ERR_MOVED (Error Code 10019)

The file system which contains the current filehandle object is not
present at the server. It may have been relocated, migrated to
another server or may have never been present. The client may obtain
the new file system location by obtaining the "fs_locations" or
attribute for the current filehandle. For further discussion, refer
to Section 8.

13.1.2.5. NFS4ERR_NOFILEHANDLE (Error Code 10020)

The logical current or saved filehandle value is required by the
current operation and is not set. This may be a result of a
malformed COMPOUND operation (i.e., no PUTFH or PUTROOTFH before an
operation that requires the current filehandle be set).

13.1.2.6. NFS4ERR_NOTDIR (Error Code 20)

The current (or saved) filehandle designates an object which is not a
directory for an operation in which a directory is required.

13.1.2.7. NFS4ERR_STALE (Error Code 70)

The current or saved filehandle value designating an argument to the
current operation is invalid. The file system object referred to by
that filehandle no longer exists or access to it has been revoked.

13.1.2.8. NFS4ERR_SYMLINK (Error Code 10029)

The current filehandle designates a symbolic link when the current
operation does not allow a symbolic link as the target.

13.1.3. Compound Structure Errors

This section deals with errors that relate to overall structure of a
Compound request (by which we mean to include both COMPOUND and
CB_COMPOUND), rather than to particular operations.

There are a number of basic constraints on the operations that may
appear in a Compound request.

13.1.3.1. NFS_OK (Error code 0)

Indicates the operation completed successfully, in that all of the
constituent operations completed without error.

13.1.3.2. **NFS4ERR_MINOR_VERS_MISMATCH (Error code 10021)**

   The minor version specified is not one that the current listener
   supports. This value is returned in the overall status for the
   Compound but is not associated with a specific operation since the
   results must specify a result count of zero.

13.1.3.3. **NFS4ERR_OP_ILLEGAL (Error Code 10044)**

   The operation code is not a valid one for the current Compound
   procedure. The opcode in the result stream matched with this error
   is the ILLEGAL value, although the value that appears in the request
   stream may be different. Where an illegal value appears and the
   replier pre-parses all operations for a Compound procedure before
   doing any operation execution, an RPC-level XDR error may be returned
   in this case.

13.1.3.4. **NFS4ERR_RESOURCE (Error Code 10018)**

   For the processing of the Compound procedure, the server may exhaust
   available resources and cannot continue processing operations within
   the Compound procedure. This error will be returned from the server
   in those instances of resource exhaustion related to the processing
   of the Compound procedure.

13.1.4. **File System Errors**

   These errors describe situations which occurred in the underlying
   file system implementation rather than in the protocol or any NFSv4.x
   feature.

13.1.4.1. **NFS4ERR_BADTYPE (Error Code 10007)**

   An attempt was made to create an object with an inappropriate type
   specified to CREATE. This may be because the type is undefined,
   because it is a type not supported by the server, or because it is a
   type for which create is not intended such as a regular file or named
   attribute, for which OPEN is used to do the file creation.

13.1.4.2. **NFS4ERR_DQUOT (Error Code 69)**

   Resource (quota) hard limit exceeded. The user's resource limit on
   the server has been exceeded.

13.1.4.3. **NFS4ERR_EXIST (Error Code 17)**

   A file system object of the specified target name (when creating,
   renaming or linking) already exists.

13.1.4.4. **NFS4ERR_FBIG (Error Code 27)**

   Filesystem object too large. The operation would have caused a file
   system object to grow beyond the server's limit.

13.1.4.5. **NFS4ERR_FILE_OPEN (Error Code 10046)**

   The operation is not allowed because a file system object involved in
   the operation is currently open. Servers may, but are not required
   to disallow linking-to, removing, or renaming open file system
   objects.

13.1.4.6. **NFS4ERR_IO (Error Code 6)**

   Indicates that an I/O error occurred for which the file system was
   unable to provide recovery.

13.1.4.7. **NFS4ERR_MLINK (Error Code 31)**

   The request would have caused the server's limit for the number of
   hard links a file system object may have to be exceeded.

13.1.4.8. **NFS4ERR_NOENT (Error Code 2)**

   Indicates no such file or directory. The file system object
   referenced by the name specified does not exist.

13.1.4.9. **NFS4ERR_NOSPC (Error Code 28)**

   Indicates no space left on device. The operation would have caused
   the server's file system to exceed its limit.

13.1.4.10. **NFS4ERR_NOTEMPTY (Error Code 66)**

   An attempt was made to remove a directory that was not empty.

13.1.4.11. **NFS4ERR_NXIO (Error Code 5)**

   I/O error. No such device or address.

13.1.4.12. **NFS4ERR_RESTOREFH (Error Code 10030)**

   The RESTOREFH operation does not have a saved filehandle (identified
   by SAVEFH) to operate upon.

[13.1.4.13](#). **NFS4ERR_ROFS (Error Code 30)**

   Indicates a read-only file system. A modifying operation was
   attempted on a read-only file system.

[13.1.4.14](#). **NFS4ERR_XDEV (Error Code 18)**

   Indicates an attempt to do an operation, such as linking, that
   inappropriately crosses a boundary. This may be due to such
   boundaries as:

   o That between file systems (where the fsids are different).

   o That between different named attribute directories or between a
      named attribute directory and an ordinary directory.

   o That between regions of a file system that the file system
      implementation treats as separate (for example for space
      accounting purposes), and where cross-connection between the
      regions are not allowed.

[13.1.5](#). **State Management Errors**

   These errors indicate problems with the stateid (or one of the
   stateids) passed to a given operation. This includes situations in
   which the stateid is invalid as well as situations in which the
   stateid is valid but designates revoked locking state. Depending on
   the operation, the stateid when valid may designate opens, byte-range
   locks, or file delegations.

[13.1.5.1](#). **NFS4ERR_ADMIN_REVOKED (Error Code 10047)**

   A stateid designates locking state of any type that has been revoked
   due to administrative interaction, possibly while the lease is valid,
   or because a delegation was revoked because of failure to return it,
   while the lease was valid.

[13.1.5.2](#). **NFS4ERR_BAD_STATEID (Error Code 10025)**

   A stateid generated by the current server instance was used which
   either:

   o Does not designate any locking state (either current or
      superseded) for a current (state-owner, file) pair.

   o Designates locking state that was freed after lease expiration but
      without any lease cancelation, as may happen in the handling of
      "courtesy locks".

13.1.5.3. **NFS4ERR_EXPIRED (Error Code 10011)**

   A stateid or clientid designates locking state of any type that has
   been revoked or released due to cancellation of the client's lease,
   either immediately upon lease expiration, or following a later
   request for a conflicting lock.

13.1.5.4. **NFS4ERR_LEASE_MOVED (Error Code 10031)**

   A lease being renewed is associated with a file system that has been
   migrated to a new server.

13.1.5.5. **NFS4ERR_OLD_STATEID (Error Code 10024)**

   A stateid is provided with a seqid value that is not the most
   current.

13.1.5.6. **NFS4ERR_STALE_STATEID (Error Code 10023)**

   A stateid generated by an earlier server instance was used.

13.1.6. **Security Errors**

   These are the various permission-related errors in NFSv4.

13.1.6.1. **NFS4ERR_ACCESS (Error Code 13)**

   Indicates permission denied. The caller does not have the correct
   permission to perform the requested operation. Contrast this with
   NFS4ERR_PERM (Section 13.1.6.2), which restricts itself to owner or
   privileged user permission failures.

13.1.6.2. **NFS4ERR_PERM (Error Code 1)**

   Indicates requester is not the owner. The operation was not allowed
   because the caller is neither a privileged user (root) nor the owner
   of the target of the operation.

13.1.6.3. **NFS4ERR_WRONGSEC (Error Code 10016)**

   Indicates that the security mechanism being used by the client for
   the operation does not match the server's security policy. The
   client should change the security mechanism being used and re-send
   the operation. SECINFO can be used to determine the appropriate
   mechanism.

13.1.7. Name Errors

   Names in NFSv4 are UTF-8 strings. When the strings are not of length
   zero, the error NFS4ERR_INVAL results. When they are not valid UTF-8
   the error NFS4ERR_INVAL also results, but servers may accommodate
   file systems with different character formats and not return this
   error. Besides this, there are a number of other errors to indicate
   specific problems with names.

13.1.7.1. NFS4ERR_BADCHAR (Error Code 10040)

   A UTF-8 string contains a character which is not supported by the
   server in the context in which it being used.

13.1.7.2. NFS4ERR_BADNAME (Error Code 10041)

   A name string in a request consisted of valid UTF-8 characters
   supported by the server but the name is not supported by the server
   as a valid name for current operation. An example might be creating
   a file or directory named ".." on a server whose file system uses
   that name for links to parent directories.

   This error should not be returned due a normalization issue in a
   string. When a file system keeps names in a particular normalization
   form, it is the server's responsibility to do the appropriate
   normalization, rather than rejecting the name.

13.1.7.3. NFS4ERR_NAMETOOLONG (Error Code 63)

   Returned when the filename in an operation exceeds the server's
   implementation limit.

13.1.8. Locking Errors

   This section deal with errors related to locking, both as to share
   reservations and byte-range locking. It does not deal with errors
   specific to the process of reclaiming locks. Those are dealt with in
   the next section.

13.1.8.1. NFS4ERR_BAD_RANGE (Error Code 10042)

   The range for a LOCK, LOCKT, or LOCKU operation is not appropriate to
   the allowable range of offsets for the server. E.g., this error
   results when a server which only supports 32-bit ranges receives a
   range that cannot be handled by that server. (See Section 15.12.4).

13.1.8.2. **NFS4ERR_BAD_SEQID (Error Code 10026)**

   The sequence number (seqid) in a locking request is neither the next
   expected number or the last number processed.

13.1.8.3. **NFS4ERR_DEADLOCK (Error Code 10045)**

   The server has been able to determine a file locking deadlock
   condition for a blocking lock request.

13.1.8.4. **NFS4ERR_DENIED (Error Code 10010)**

   An attempt to lock a file is denied. Since this may be a temporary
   condition, the client is encouraged to re-send the lock request until
   the lock is accepted. See Section 9.4 for a discussion of the re-
   send.

13.1.8.5. **NFS4ERR_LOCKED (Error Code 10012)**

   A read or write operation was attempted on a file where there was a
   conflict between the I/O and an existing lock:

   o There is a share reservation inconsistent with the I/O being done.

   o The range to be read or written intersects an existing mandatory
      byte range lock.

13.1.8.6. **NFS4ERR_LOCKS_HELD (Error Code 10037)**

   An operation was prevented by the unexpected presence of locks.

13.1.8.7. **NFS4ERR_LOCK_NOTSUPP (Error Code 10043)**

   A locking request was attempted which would require the upgrade or
   downgrade of a lock range already held by the owner when the server
   does not support atomic upgrade or downgrade of locks.

13.1.8.8. **NFS4ERR_LOCK_RANGE (Error Code 10028)**

   A lock request is operating on a range that overlaps in part a
   currently held lock for the current lock owner and does not precisely
   match a single such lock where the server does not support this type
   of request, and thus does not implement POSIX locking semantics
   [fcntl]. See Section 15.12.5, Section 15.13.5, and Section 15.14.5
   for a discussion of how this applies to LOCK, LOCKT, and LOCKU
   respectively.

13.1.8.9. **NFS4ERR_OPENMODE (Error Code 10038)**

   The client attempted a READ, WRITE, LOCK or other operation not
   sanctioned by the stateid passed (e.g., writing to a file opened only
   for read).

13.1.9. **Reclaim Errors**

   These errors relate to the process of reclaiming locks after a server
   restart.

13.1.9.1. **NFS4ERR_GRACE (Error Code 10013)**

   The server is in its recovery or grace period which should at least
   match the lease period of the server. A locking request other than a
   reclaim could not be granted during that period.

13.1.9.2. **NFS4ERR_NO_GRACE (Error Code 10033)**

   The server cannot guarantee that it has not granted state to another
   client which may conflict with this client's state. No further
   reclaims from this client will succeed.

13.1.9.3. **NFS4ERR_RECLAIM_BAD (Error Code 10034)**

   The server cannot guarantee that it has not granted state to another
   client which may conflict with the requested state. However, this
   applies only to the state requested in this call; further reclaims
   may succeed.

   Unlike NFS4ERR_RECLAIM_CONFLICT, this can occur between correctly
   functioning clients and servers: the "edge condition" scenarios
   described in Section 9.6.3.1 leave only the server knowing whether
   the client's locks are still valid, and NFS4ERR_RECLAIM_BAD is the
   server's way of informing the client that they are not.

13.1.9.4. **NFS4ERR_RECLAIM_CONFLICT (Error Code 10035)**

   The reclaim attempted by the client conflicts with a lock already
   held by another client. Unlike NFS4ERR_RECLAIM_BAD, this can only
   occur if one of the clients misbehaved.

13.1.10. **Client Management Errors**

   This sections deals with errors associated with requests used to
   create and manage client IDs.

[13.1.10.1](#). **NFS4ERR_CLID_INUSE (Error Code 10017)**

   The SETCLIENTID operation has found that a client id is already in
   use by another client.

[13.1.10.2](#). **NFS4ERR_STALE_CLIENTID (Error Code 10022)**

   A client ID not recognized by the server was used in a locking or
   SETCLIENTID_CONFIRM request.

[13.1.11](#). **Attribute Handling Errors**

   This section deals with errors specific to attribute handling within
   NFSv4.

[13.1.11.1](#). **NFS4ERR_ATTRNOTSUPP (Error Code 10032)**

   An attribute specified is not supported by the server. This error
   MUST NOT be returned by the GETATTR operation.

[13.1.11.2](#). **NFS4ERR_BADOWNER (Error Code 10039)**

   Returned when an owner or owner_group attribute value or the who
   field of an ace within an ACL attribute value cannot be translated to
   a local representation.

[13.1.11.3](#). **NFS4ERR_NOT_SAME (Error Code 10027)**

   This error is returned by the VERIFY operation to signify that the
   attributes compared were not the same as those provided in the
   client's request.

[13.1.11.4](#). **NFS4ERR_SAME (Error Code 10009)**

   This error is returned by the NVERIFY operation to signify that the
   attributes compared were the same as those provided in the client's
   request.

[13.2](#). **Operations and their valid errors**

   This section contains a table which gives the valid error returns for
   each protocol operation. The error code NFS4_OK (indicating no
   error) is not listed but should be understood to be returnable by all
   operations except ILLEGAL.

               Valid error returns for each protocol operation

Operation	Errors
ACCESS	NFS4ERR_ACCESS, NFS4ERR_BADHANDLE, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CLOSE	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADHANDLE, NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED, NFS4ERR_LOCKS_HELD, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID, NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_STALE_STATEID
COMMIT	NFS4ERR_ACCESS, NFS4ERR_BADHANDLE, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE, NFS4ERR_ROFS, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_SYMLINK
CREATE	NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE, NFS4ERR_BADNAME, NFS4ERR_BADOWNER, NFS4ERR_BADTYPE, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_DQUOT, NFS4ERR_EXIST, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC, NFS4ERR_NOTDIR, NFS4ERR_PERM, NFS4ERR_RESOURCE, NFS4ERR_ROFS, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
DELEGPURGE	NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_NOTSUPP, NFS4ERR_LEASE_MOVED, NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT, NFS4ERR_STALE_CLIENTID

```
    | DELEGRETURN           | NFS4ERR_ADMIN_REVOKED, NFS4ERR_BAD_STATEID, |
    |                       | NFS4ERR_BADXDR, NFS4ERR_DELAY,               |
    |                       | NFS4ERR_EXPIRED, NFS4ERR_INVAL,             |
    |                       | NFS4ERR_LEASE_MOVED, NFS4ERR_MOVED,         |
    |                       | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTSUPP,      |
    |                       | NFS4ERR_OLD_STATEID, NFS4ERR_RESOURCE,      |
    |                       | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,         |
    |                       | NFS4ERR_STALE_STATEID                       |
    | GETATTR               | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,          |
    |                       | NFS4ERR_BADXDR, NFS4ERR_DELAY,              |
    |                       | NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,           |
    |                       | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,   |
    |                       | NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE,     |
    |                       | NFS4ERR_SERVERFAULT, NFS4ERR_STALE          |
    | GETFH                 | NFS4ERR_BADHANDLE, NFS4ERR_FHEXPIRED,       |
    |                       | NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,        |
    |                       | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,      |
    |                       | NFS4ERR_STALE                               |
    | ILLEGAL               | NFS4ERR_BADXDR, NFS4ERR_OP_ILLEGAL          |
    | LINK                  | NFS4ERR_ACCESS, NFS4ERR_BADCHAR,            |
    |                       | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,         |
    |                       | NFS4ERR_BADXDR, NFS4ERR_DELAY,              |
    |                       | NFS4ERR_DQUOT, NFS4ERR_EXIST,               |
    |                       | NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN,       |
    |                       | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR,   |
    |                       | NFS4ERR_MLINK, NFS4ERR_MOVED,               |
    |                       | NFS4ERR_NAMETOOLONG, NFS4ERR_NOENT,         |
    |                       | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,        |
    |                       | NFS4ERR_NOTDIR, NFS4ERR_NOTSUPP,            |
    |                       | NFS4ERR_RESOURCE, NFS4ERR_ROFS,             |
    |                       | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,         |
    |                       | NFS4ERR_WRONGSEC, NFS4ERR_XDEV              |
    | LOCK                  | NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,      |
    |                       | NFS4ERR_BADHANDLE, NFS4ERR_BAD_RANGE,       |
    |                       | NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,     |
    |                       | NFS4ERR_BADXDR, NFS4ERR_DEADLOCK,           |
    |                       | NFS4ERR_DELAY, NFS4ERR_DENIED,              |
    |                       | NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,         |
    |                       | NFS4ERR_GRACE, NFS4ERR_INVAL,               |
    |                       | NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,         |
    |                       | NFS4ERR_LOCK_NOTSUPP, NFS4ERR_LOCK_RANGE,   |
    |                       | NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,        |
    |                       | NFS4ERR_NO_GRACE, NFS4ERR_OLD_STATEID,      |
    |                       | NFS4ERR_OPENMODE, NFS4ERR_RECLAIM_BAD,      |
    |                       | NFS4ERR_RECLAIM_CONFLICT, NFS4ERR_RESOURCE, |
    |                       | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,         |
    |                       | NFS4ERR_STALE_CLIENTID,                     |
    |                       | NFS4ERR_STALE_STATEID                       |
```

LOCKT	NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,	
	NFS4ERR_BAD_RANGE, NFS4ERR_BADXDR,	
	NFS4ERR_DELAY, NFS4ERR_DENIED,	
	NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,	
	NFS4ERR_GRACE, NFS4ERR_INVAL,	
	NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,	
	NFS4ERR_LOCK_RANGE, NFS4ERR_MOVED,	
	NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE,	
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,	
	NFS4ERR_STALE_CLIENTID	
LOCKU	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,	
	NFS4ERR_BADHANDLE, NFS4ERR_BAD_RANGE,	
	NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,	
	NFS4ERR_BADXDR, NFS4ERR_DELAY,	
	NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,	
	NFS4ERR_GRACE, NFS4ERR_INVAL,	
	NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,	
	NFS4ERR_LOCK_RANGE, NFS4ERR_MOVED,	
	NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID,	
	NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,	
	NFS4ERR_STALE, NFS4ERR_STALE_STATEID	
LOOKUP	NFS4ERR_ACCESS, NFS4ERR_BADCHAR,	
	NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,	
	NFS4ERR_BADXDR, NFS4ERR_DELAY,	
	NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,	
	NFS4ERR_IO, NFS4ERR_MOVED,	
	NFS4ERR_NAMETOOLONG, NFS4ERR_NOENT,	
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,	
	NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,	
	NFS4ERR_STALE, NFS4ERR_SYMLINK,	
	NFS4ERR_WRONGSEC	
LOOKUPP	NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,	
	NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,	
	NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOENT,	
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,	
	NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,	
	NFS4ERR_STALE, NFS4ERR_SYMLINK,	
	NFS4ERR_WRONGSEC	
NVERIFY	NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,	
	NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE,	
	NFS4ERR_BADXDR, NFS4ERR_DELAY,	
	NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,	
	NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,	
	NFS4ERR_NOFILEHANDLE, NFS4ERR_SAME,	
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE	

OPEN	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,
	NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR,
	NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,
	NFS4ERR_BADOWNER, NFS4ERR_BADXDR,
	NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,
	NFS4ERR_DELAY, NFS4ERR_DQUOT,
	NFS4ERR_EXIST, NFS4ERR_EXPIRED,
	NFS4ERR_FBIG, NFS4ERR_FHEXPIRED,
	NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO,
	NFS4ERR_ISDIR, NFS4ERR_MOVED,
	NFS4ERR_NAMETOOLONG, NFS4ERR_NOENT,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,
	NFS4ERR_NOTDIR, NFS4ERR_NOTSUP,
	NFS4ERR_NO_GRACE, NFS4ERR_OLD_STATEID,
	NFS4ERR_PERM, NFS4ERR_RECLAIM_BAD,
	NFS4ERR_RECLAIM_CONFLICT, NFS4ERR_RESOURCE,
	NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,
	NFS4ERR_SHARE_DENIED, NFS4ERR_STALE,
	NFS4ERR_STALE_CLIENTID, NFS4ERR_SYMLINK,
	NFS4ERR_WRONGSEC
OPENATTR	NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,
	NFS4ERR_BADXDR, NFS4ERR_DELAY,
	NFS4ERR_DQUOT, NFS4ERR_FHEXPIRED,
	NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOENT,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,
	NFS4ERR_NOTSUPP, NFS4ERR_RESOURCE,
	NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE
OPEN_CONFIRM	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADHANDLE,
	NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,
	NFS4ERR_BADXDR, NFS4ERR_EXPIRED,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,
	NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,
	NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OLD_STATEID, NFS4ERR_RESOURCE,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_STALE_STATEID
OPEN_DOWNGRADE	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADHANDLE,
	NFS4ERR_BADXDR, NFS4ERR_BAD_SEQID,
	NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,
	NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,
	NFS4ERR_INVAL, NFS4ERR_LEASE_MOVED,
	NFS4ERR_LOCKS_HELD, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID,
	NFS4ERR_RESOURCE, NFS4ERR_ROFS,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_STALE_STATEID

```
| PUTFH              | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,     |
|                    | NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,      |
|                    | NFS4ERR_MOVED, NFS4ERR_SERVERFAULT,    |
|                    | NFS4ERR_STALE, NFS4ERR_WRONGSEC        |
| PUTPUBFH           | NFS4ERR_DELAY, NFS4ERR_SERVERFAULT,    |
|                    | NFS4ERR_WRONGSEC                       |
| PUTROOTFH          | NFS4ERR_DELAY, NFS4ERR_SERVERFAULT,    |
|                    | NFS4ERR_WRONGSEC                       |
| READ               | NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, |
|                    | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,     |
|                    | NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,    |
|                    | NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,    |
|                    | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO, |
|                    | NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,    |
|                    | NFS4ERR_LOCKED, NFS4ERR_MOVED,         |
|                    | NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID, |
|                    | NFS4ERR_OPENMODE, NFS4ERR_RESOURCE,    |
|                    | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,    |
|                    | NFS4ERR_STALE_STATEID, NFS4ERR_SYMLINK |
| READDIR            | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,     |
|                    | NFS4ERR_BADXDR, NFS4ERR_BAD_COOKIE,    |
|                    | NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,      |
|                    | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, |
|                    | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,  |
|                    | NFS4ERR_NOT_SAME, NFS4ERR_RESOURCE,    |
|                    | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,    |
|                    | NFS4ERR_TOOSMALL                       |
| READLINK           | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,     |
|                    | NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,      |
|                    | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR, |
|                    | NFS4ERR_MOVED, NFS4ERR_NOTSUP,         |
|                    | NFS4ERR_RESOURCE, NFS4ERR_NOFILEHANDLE, |
|                    | NFS4ERR_SERVERFAULT, NFS4ERR_STALE     |
| RELEASE_LOCKOWNER  | NFS4ERR_BADXDR, NFS4ERR_EXPIRED,       |
|                    | NFS4ERR_LEASE_MOVED, NFS4ERR_LOCKS_HELD, |
|                    | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT, |
|                    | NFS4ERR_STALE_CLIENTID                 |
| REMOVE             | NFS4ERR_ACCESS, NFS4ERR_BADCHAR,       |
|                    | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,    |
|                    | NFS4ERR_BADXDR, NFS4ERR_DELAY,         |
|                    | NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN,  |
|                    | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO, |
|                    | NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,    |
|                    | NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE,   |
|                    | NFS4ERR_NOTDIR, NFS4ERR_NOTEMPTY,      |
|                    | NFS4ERR_RESOURCE, NFS4ERR_ROFS,        |
|                    | NFS4ERR_SERVERFAULT, NFS4ERR_STALE     |
```

```
|  RENAME               |  NFS4ERR_ACCESS, NFS4ERR_BADCHAR,      |
|                       |  NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,   |
|                       |  NFS4ERR_BADXDR, NFS4ERR_DELAY,        |
|                       |  NFS4ERR_DQUOT, NFS4ERR_EXIST,         |
|                       |  NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN, |
|                       |  NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO, |
|                       |  NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,   |
|                       |  NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE,  |
|                       |  NFS4ERR_NOSPC, NFS4ERR_NOTDIR,        |
|                       |  NFS4ERR_NOTEMPTY, NFS4ERR_RESOURCE,   |
|                       |  NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,    |
|                       |  NFS4ERR_STALE, NFS4ERR_WRONGSEC,      |
|                       |  NFS4ERR_XDEV                          |
|  RENEW                |  NFS4ERR_ACCESS, NFS4ERR_BADXDR,       |
|                       |  NFS4ERR_CB_PATH_DOWN, NFS4ERR_EXPIRED, |
|                       |  NFS4ERR_LEASE_MOVED, NFS4ERR_RESOURCE, |
|                       |  NFS4ERR_SERVERFAULT, NFS4ERR_STALE_CLIENTID |
|  RESTOREFH            |  NFS4ERR_BADHANDLE, NFS4ERR_FHEXPIRED, |
|                       |  NFS4ERR_MOVED, NFS4ERR_RESOURCE,      |
|                       |  NFS4ERR_RESTOREFH, NFS4ERR_SERVERFAULT, |
|                       |  NFS4ERR_STALE, NFS4ERR_WRONGSEC       |
|  SAVEFH               |  NFS4ERR_BADHANDLE, NFS4ERR_FHEXPIRED, |
|                       |  NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,  |
|                       |  NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT, |
|                       |  NFS4ERR_STALE                         |
|  SECINFO              |  NFS4ERR_ACCESS, NFS4ERR_BADCHAR,      |
|                       |  NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,   |
|                       |  NFS4ERR_BADXDR, NFS4ERR_DELAY,        |
|                       |  NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,     |
|                       |  NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,   |
|                       |  NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE,  |
|                       |  NFS4ERR_NOTDIR, NFS4ERR_RESOURCE,     |
|                       |  NFS4ERR_SERVERFAULT, NFS4ERR_STALE    |
|  SETATTR              |  NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, |
|                       |  NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR, |
|                       |  NFS4ERR_BADHANDLE, NFS4ERR_BADOWNER,  |
|                       |  NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID,  |
|                       |  NFS4ERR_DELAY, NFS4ERR_DQUOT,         |
|                       |  NFS4ERR_EXPIRED, NFS4ERR_FBIG,        |
|                       |  NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,     |
|                       |  NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR, |
|                       |  NFS4ERR_LEASE_MOVED, NFS4ERR_LOCKED,  |
|                       |  NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,  |
|                       |  NFS4ERR_NOSPC, NFS4ERR_OLD_STATEID,   |
|                       |  NFS4ERR_OPENMODE, NFS4ERR_PERM,       |
|                       |  NFS4ERR_RESOURCE, NFS4ERR_ROFS,       |
|                       |  NFS4ERR_SERVERFAULT, NFS4ERR_STALE,   |
|                       |  NFS4ERR_STALE_STATEID                 |
```

```
    |  SETCLIENTID         | NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,          |
    |                      | NFS4ERR_DELAY, NFS4ERR_INVAL,                |
    |                      | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT        |
    |  SETCLIENTID_CONFIRM | NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,          |
    |                      | NFS4ERR_DELAY, NFS4ERR_RESOURCE,             |
    |                      | NFS4ERR_SERVERFAULT, NFS4ERR_STALE_CLIENTID |
    |  VERIFY              | NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,         |
    |                      | NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE,          |
    |                      | NFS4ERR_BADXDR, NFS4ERR_DELAY,               |
    |                      | NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,            |
    |                      | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,    |
    |                      | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOT_SAME,      |
    |                      | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,       |
    |                      | NFS4ERR_STALE                               |
    |  WRITE               | NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,       |
    |                      | NFS4ERR_BADXDR, NFS4ERR_BADHANDLE,           |
    |                      | NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,          |
    |                      | NFS4ERR_DQUOT, NFS4ERR_EXPIRED,              |
    |                      | NFS4ERR_FBIG, NFS4ERR_FHEXPIRED,             |
    |                      | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO,    |
    |                      | NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,          |
    |                      | NFS4ERR_LOCKED, NFS4ERR_MOVED,               |
    |                      | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,         |
    |                      | NFS4ERR_NXIO, NFS4ERR_OLD_STATEID,           |
    |                      | NFS4ERR_OPENMODE, NFS4ERR_RESOURCE,          |
    |                      | NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,           |
    |                      | NFS4ERR_STALE, NFS4ERR_STALE_STATEID,        |
    |                      | NFS4ERR_SYMLINK                             |
    +--------------------+---------------------------------------------+
```

                                Table 6

13.3. Callback operations and their valid errors

   This section contains a table which gives the valid error returns for
   each callback operation. The error code NFS4_OK (indicating no
   error) is not listed but should be understood to be returnable by all
   callback operations with the exception of CB_ILLEGAL.

                Valid error returns for each protocol callback operation

```
+-------------+--------------------------------------------------------+
| Callback    | Errors                                                 |
| Operation   |                                                        |
+-------------+--------------------------------------------------------+
| CB_GETATTR  | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR, NFS4ERR_DELAY,      |
|             | NFS4ERR_INVAL, NFS4ERR_SERVERFAULT                     |
| CB_ILLEGAL  | NFS4ERR_BADXDR, NFS4ERR_OP_ILLEGAL                     |
| CB_RECALL   | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,                     |
|             | NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,                   |
|             | NFS4ERR_SERVERFAULT                                    |
+-------------+--------------------------------------------------------+
```

                                Table 7

13.4. Errors and the operations that use them

```
+--------------------------+-----------------------------------------+
| Error                    | Operations                              |
+--------------------------+-----------------------------------------+
| NFS4ERR_ACCESS           | ACCESS, COMMIT, CREATE, GETATTR, LINK,  |
|                          | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP,    |
|                          | NVERIFY, OPEN, OPENATTR, READ,          |
|                          | READDIR, READLINK, REMOVE, RENAME,      |
|                          | RENEW, SECINFO, SETATTR, VERIFY, WRITE  |
| NFS4ERR_ADMIN_REVOKED    | CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN,  |
|                          | OPEN_CONFIRM, OPEN_DOWNGRADE, READ,     |
|                          | SETATTR, WRITE                          |
| NFS4ERR_ATTRNOTSUPP      | CREATE, NVERIFY, OPEN, SETATTR, VERIFY  |
| NFS4ERR_BADCHAR          | CREATE, LINK, LOOKUP, NVERIFY, OPEN,    |
|                          | REMOVE, RENAME, SECINFO, SETATTR,       |
|                          | VERIFY                                  |
| NFS4ERR_BADHANDLE        | ACCESS, CB_GETATTR, CB_RECALL, CLOSE,   |
|                          | COMMIT, CREATE, GETATTR, GETFH, LINK,   |
|                          | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP,    |
|                          | NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM,  |
|                          | OPEN_DOWNGRADE, PUTFH, READ, READDIR,   |
|                          | READLINK, REMOVE, RENAME, RESTOREFH,    |
|                          | SAVEFH, SECINFO, SETATTR, VERIFY,       |
|                          | WRITE                                   |
| NFS4ERR_BADNAME          | CREATE, LINK, LOOKUP, OPEN, REMOVE,     |
|                          | RENAME, SECINFO                         |
| NFS4ERR_BADOWNER         | CREATE, OPEN, SETATTR                   |
| NFS4ERR_BADTYPE          | CREATE                                  |
```

```
| NFS4ERR_BADXDR         | ACCESS, CB_GETATTR, CB_ILLEGAL,       |
|                        | CB_RECALL, CLOSE, COMMIT, CREATE,     |
|                        | DELEGPURGE, DELEGRETURN, GETATTR,     |
|                        | ILLEGAL, LINK, LOCK, LOCKT, LOCKU,    |
|                        | LOOKUP, NVERIFY, OPEN, OPENATTR,      |
|                        | OPEN_CONFIRM, OPEN_DOWNGRADE, PUTFH,  |
|                        | READ, READDIR, RELEASE_LOCKOWNER,     |
|                        | REMOVE, RENAME, RENEW, SECINFO,       |
|                        | SETATTR, SETCLIENTID,                 |
|                        | SETCLIENTID_CONFIRM, VERIFY, WRITE    |
| NFS4ERR_BAD_COOKIE     | READDIR                               |
| NFS4ERR_BAD_RANGE      | LOCK, LOCKT, LOCKU                     |
| NFS4ERR_BAD_SEQID      | CLOSE, LOCK, LOCKU, OPEN,             |
|                        | OPEN_CONFIRM, OPEN_DOWNGRADE          |
| NFS4ERR_BAD_STATEID    | CB_RECALL, CLOSE, DELEGRETURN, LOCK,  |
|                        | LOCKU, OPEN, OPEN_CONFIRM,            |
|                        | OPEN_DOWNGRADE, READ, SETATTR, WRITE  |
| NFS4ERR_CB_PATH_DOWN   | RENEW                                 |
| NFS4ERR_CLID_INUSE     | SETCLIENTID, SETCLIENTID_CONFIRM      |
| NFS4ERR_DEADLOCK       | LOCK                                  |
| NFS4ERR_DELAY          | ACCESS, CB_GETATTR, CB_RECALL, CLOSE, |
|                        | COMMIT, CREATE, DELEGPURGE,           |
|                        | DELEGRETURN, GETATTR, LINK, LOCK,     |
|                        | LOCKT, LOCKU, LOOKUP, LOOKUPP,        |
|                        | NVERIFY, OPEN, OPENATTR,              |
|                        | OPEN_DOWNGRADE, PUTFH, PUTPUBFH,      |
|                        | PUTROOTFH, READ, READDIR, READLINK,   |
|                        | REMOVE, RENAME, SECINFO, SETATTR,     |
|                        | SETCLIENTID, SETCLIENTID_CONFIRM,     |
|                        | VERIFY, WRITE                         |
| NFS4ERR_DENIED         | LOCK, LOCKT                           |
| NFS4ERR_DQUOT          | CREATE, LINK, OPEN, OPENATTR, RENAME, |
|                        | SETATTR, WRITE                        |
| NFS4ERR_EXIST          | CREATE, LINK, OPEN, RENAME            |
| NFS4ERR_EXPIRED        | CLOSE, DELEGRETURN, LOCK, LOCKT,      |
|                        | LOCKU, OPEN, OPEN_CONFIRM,            |
|                        | OPEN_DOWNGRADE, READ,                 |
|                        | RELEASE_LOCKOWNER, RENEW, SETATTR,    |
|                        | WRITE                                 |
| NFS4ERR_FBIG           | OPEN, SETATTR, WRITE                  |
| NFS4ERR_FHEXPIRED      | ACCESS, CLOSE, COMMIT, CREATE,        |
|                        | GETATTR, GETFH, LINK, LOCK, LOCKT,    |
|                        | LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN,|
|                        | OPENATTR, OPEN_CONFIRM,               |
|                        | OPEN_DOWNGRADE, PUTFH, READ, READDIR, |
|                        | READLINK, REMOVE, RENAME, RESTOREFH,  |
|                        | SAVEFH, SECINFO, SETATTR, VERIFY,     |
|                        | WRITE                                 |
```

NFS4ERR_FILE_OPEN	LINK, REMOVE, RENAME
NFS4ERR_GRACE	GETATTR, LOCK, LOCKT, LOCKU, NVERIFY, OPEN, READ, REMOVE, RENAME, SETATTR, VERIFY, WRITE
NFS4ERR_INVAL	ACCESS, CB_GETATTR, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, LINK, LOCK, LOCKT, LOCKU, LOOKUP, NVERIFY, OPEN, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, READDIR, READLINK, REMOVE, RENAME, SECINFO, SETATTR, SETCLIENTID, VERIFY, WRITE
NFS4ERR_IO	ACCESS, COMMIT, CREATE, GETATTR, LINK, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, READ, READDIR, READLINK, REMOVE, RENAME, SETATTR, VERIFY, WRITE
NFS4ERR_ISDIR	CLOSE, COMMIT, LINK, LOCK, LOCKT, LOCKU, OPEN, OPEN_CONFIRM, READ, READLINK, SETATTR, WRITE
NFS4ERR_LEASE_MOVED	CLOSE, DELEGPURGE, DELEGRETURN, LOCK, LOCKT, LOCKU, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, RELEASE_LOCKOWNER, RENEW, SETATTR, WRITE
NFS4ERR_LOCKED	READ, SETATTR, WRITE
NFS4ERR_LOCKS_HELD	CLOSE, OPEN_DOWNGRADE, RELEASE_LOCKOWNER
NFS4ERR_LOCK_NOTSUPP	LOCK
NFS4ERR_LOCK_RANGE	LOCK, LOCKT, LOCKU
NFS4ERR_MLINK	LINK
NFS4ERR_MOVED	ACCESS, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, GETFH, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SETATTR, VERIFY, WRITE
NFS4ERR_NAMETOOLONG	CREATE, LINK, LOOKUP, OPEN, REMOVE, RENAME, SECINFO
NFS4ERR_NOENT	LINK, LOOKUP, LOOKUPP, OPEN, OPENATTR, REMOVE, RENAME, SECINFO
NFS4ERR_NOFILEHANDLE	ACCESS, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, GETFH, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, READDIR, READLINK, REMOVE, RENAME, SAVEFH, SECINFO, SETATTR, VERIFY, WRITE

```
| NFS4ERR_NOSPC            | CREATE, LINK, OPEN, OPENATTR, RENAME, |
|                          | SETATTR, WRITE                        |
| NFS4ERR_NOTDIR           | CREATE, LINK, LOOKUP, LOOKUPP, OPEN,  |
|                          | READDIR, REMOVE, RENAME, SECINFO      |
| NFS4ERR_NOTEMPTY         | REMOVE, RENAME                        |
| NFS4ERR_NOTSUP           | OPEN, READLINK                        |
| NFS4ERR_NOTSUPP          | DELEGPURGE, DELEGRETURN, LINK,        |
|                          | OPENATTR                              |
| NFS4ERR_NOT_SAME         | READDIR, VERIFY                       |
| NFS4ERR_NO_GRACE         | LOCK, OPEN                            |
| NFS4ERR_NXIO             | WRITE                                 |
| NFS4ERR_OLD_STATEID      | CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, |
|                          | OPEN_CONFIRM, OPEN_DOWNGRADE, READ,   |
|                          | SETATTR, WRITE                        |
| NFS4ERR_OPENMODE         | LOCK, READ, SETATTR, WRITE            |
| NFS4ERR_OP_ILLEGAL       | CB_ILLEGAL, ILLEGAL                   |
| NFS4ERR_PERM             | CREATE, OPEN, SETATTR                 |
| NFS4ERR_RECLAIM_BAD      | LOCK, OPEN                            |
| NFS4ERR_RECLAIM_CONFLICT | LOCK, OPEN                            |
| NFS4ERR_RESOURCE         | ACCESS, CLOSE, COMMIT, CREATE,        |
|                          | DELEGPURGE, DELEGRETURN, GETATTR,     |
|                          | GETFH, LINK, LOCK, LOCKT, LOCKU,      |
|                          | LOOKUP, LOOKUPP, OPEN, OPENATTR,      |
|                          | OPEN_CONFIRM, OPEN_DOWNGRADE, READ,   |
|                          | READDIR, READLINK, RELEASE_LOCKOWNER, |
|                          | REMOVE, RENAME, RENEW, RESTOREFH,     |
|                          | SAVEFH, SECINFO, SETATTR, SETCLIENTID, |
|                          | SETCLIENTID_CONFIRM, VERIFY, WRITE    |
| NFS4ERR_RESTOREFH        | RESTOREFH                             |
| NFS4ERR_ROFS             | COMMIT, CREATE, LINK, OPEN, OPENATTR, |
|                          | OPEN_DOWNGRADE, REMOVE, RENAME,       |
|                          | SETATTR, WRITE                        |
| NFS4ERR_SAME             | NVERIFY                               |
| NFS4ERR_SERVERFAULT      | ACCESS, CB_GETATTR, CB_RECALL, CLOSE, |
|                          | COMMIT, CREATE, DELEGPURGE,           |
|                          | DELEGRETURN, GETATTR, GETFH, LINK,    |
|                          | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP,  |
|                          | NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, |
|                          | OPEN_DOWNGRADE, PUTFH, PUTPUBFH,      |
|                          | PUTROOTFH, READ, READDIR, READLINK,   |
|                          | RELEASE_LOCKOWNER, REMOVE, RENAME,    |
|                          | RENEW, RESTOREFH, SAVEFH, SECINFO,    |
|                          | SETATTR, SETCLIENTID,                 |
|                          | SETCLIENTID_CONFIRM, VERIFY, WRITE    |
| NFS4ERR_SHARE_DENIED     | OPEN                                  |
```

```
    | NFS4ERR_STALE           | ACCESS, CLOSE, COMMIT, CREATE,         |
    |                         | DELEGRETURN, GETATTR, GETFH, LINK,     |
    |                         | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP,   |
    |                         | NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, |
    |                         | OPEN_DOWNGRADE, PUTFH, READ, READDIR,  |
    |                         | READLINK, REMOVE, RENAME, RESTOREFH,   |
    |                         | SAVEFH, SECINFO, SETATTR, VERIFY,      |
    |                         | WRITE                                  |
    | NFS4ERR_STALE_CLIENTID  | DELEGPURGE, LOCK, LOCKT, OPEN,         |
    |                         | RELEASE_LOCKOWNER, RENEW,              |
    |                         | SETCLIENTID_CONFIRM                    |
    | NFS4ERR_STALE_STATEID   | CLOSE, DELEGRETURN, LOCK, LOCKU,       |
    |                         | OPEN_CONFIRM, OPEN_DOWNGRADE, READ,    |
    |                         | SETATTR, WRITE                         |
    | NFS4ERR_SYMLINK         | COMMIT, LOOKUP, LOOKUPP, OPEN, READ,   |
    |                         | WRITE                                  |
    | NFS4ERR_TOOSMALL        | READDIR                                |
    | NFS4ERR_WRONGSEC        | LINK, LOOKUP, LOOKUPP, OPEN, PUTFH,    |
    |                         | PUTPUBFH, PUTROOTFH, RENAME, RESTOREFH |
    | NFS4ERR_XDEV            | LINK, RENAME                           |
    +-------------------------+----------------------------------------+
```

                             Table 8

14. NFSv4 Requests

   For the NFSv4 RPC program, there are two traditional RPC procedures:
   NULL and COMPOUND. All other functionality is defined as a set of
   operations and these operations are defined in normal XDR/RPC syntax
   and semantics. However, these operations are encapsulated within the
   COMPOUND procedure. This requires that the client combine one or
   more of the NFSv4 operations into a single request.

   The NFS4_CALLBACK program is used to provide server to client
   signaling and is constructed in a similar fashion as the NFSv4
   program. The procedures CB_NULL and CB_COMPOUND are defined in the
   same way as NULL and COMPOUND are within the NFS program. The
   CB_COMPOUND request also encapsulates the remaining operations of the
   NFS4_CALLBACK program. There is no predefined RPC program number for
   the NFS4_CALLBACK program. It is up to the client to specify a
   program number in the "transient" program range. The program and
   port number of the NFS4_CALLBACK program are provided by the client
   as part of the SETCLIENTID/SETCLIENTID_CONFIRM sequence. The program
   and port can be changed by another SETCLIENTID/SETCLIENTID_CONFIRM
   sequence, and it is possible to use the sequence to change them
   within a client incarnation without removing relevant leased client
   state.

14.1. **Compound Procedure**

   The COMPOUND procedure provides the opportunity for better
   performance within high latency networks. The client can avoid
   cumulative latency of multiple RPCs by combining multiple dependent
   operations into a single COMPOUND procedure. A compound operation
   may provide for protocol simplification by allowing the client to
   combine basic procedures into a single request that is customized for
   the client's environment.

   The CB_COMPOUND procedure precisely parallels the features of
   COMPOUND as described above.

   The basic structure of the COMPOUND procedure is:

```
+-----+--------------+--------+-----------+-----------+-----------+--
| tag | minorversion | numops | op + args | op + args | op + args |
+-----+--------------+--------+-----------+-----------+-----------+--
```

   and the reply's structure is:

```
  +------------+-----+--------+----------------------+--
  |last status | tag | numres | status + op + results |
  +------------+-----+--------+----------------------+--
```

   The numops and numres fields, used in the depiction above, represent
   the count for the counted array encoding use to signify the number of
   arguments or results encoded in the request and response. As per the
   XDR encoding, these counts must match exactly the number of operation
   arguments or results encoded.

14.2. **Evaluation of a Compound Request**

   The server will process the COMPOUND procedure by evaluating each of
   the operations within the COMPOUND procedure in order. Each
   component operation consists of a 32 bit operation code, followed by
   the argument of length determined by the type of operation. The
   results of each operation are encoded in sequence into a reply
   buffer. The results of each operation are preceded by the opcode and
   a status code (normally zero). If an operation results in a non-zero
   status code, the status will be encoded and evaluation of the
   compound sequence will halt and the reply will be returned. Note
   that evaluation stops even in the event of "non error" conditions
   such as NFS4ERR_SAME.

   There are no atomicity requirements for the operations contained
   within the COMPOUND procedure. The operations being evaluated as
   part of a COMPOUND request may be evaluated simultaneously with other

COMPOUND requests that the server receives.

A COMPOUND is not a transaction and it is the client's responsibility
for recovering from any partially completed COMPOUND procedure.
These may occur at any point due to errors such as NFS4ERR_RESOURCE
and NFS4ERR_DELAY. Note that these errors can occur in an otherwise
valid operation string. Further, a server reboot which occurs in the
middle of processing a COMPOUND procedure may leave the client with
the difficult task of determining how far COMPOUND processing has
proceeded. Therefore, the client should avoid overly complex
COMPOUND procedures in the event of the failure of an operation
within the procedure.

Each operation assumes a "current" and "saved" filehandle that is
available as part of the execution context of the compound request.
Operations may set, change, or return the current filehandle. The
"saved" filehandle is used for temporary storage of a filehandle
value and as operands for the RENAME and LINK operations.

14.3. Synchronous Modifying Operations

NFSv4 operations that modify the file system are synchronous. When
an operation is successfully completed at the server, the client can
depend that any data associated with the request is now on stable
storage (the one exception is in the case of the file data in a WRITE
operation with the UNSTABLE4 option specified).

This implies that any previous operations within the same compound
request are also reflected in stable storage. This behavior enables
the client's ability to recover from a partially executed compound
request which may resulted from the failure of the server. For
example, if a compound request contains operations A and B and the
server is unable to send a response to the client, depending on the
progress the server made in servicing the request the result of both
operations may be reflected in stable storage or just operation A may
be reflected. The server must not have just the results of operation
B in stable storage.

14.4. Operation Values

The operations encoded in the COMPOUND procedure are identified by
operation values. To avoid overlap with the RPC procedure numbers,
operations 0 (zero) and 1 are not defined. Operation 2 is not
defined but reserved for future use with minor versioning.

[15](#). NFSv4 Procedures

[15.1](#). Procedure 0: NULL - No Operation

[15.1.1](#). SYNOPSIS

    <null>

[15.1.2](#). ARGUMENT

    void;

[15.1.3](#). RESULT

    void;

[15.1.4](#). DESCRIPTION

    Standard NULL procedure. Void argument, void response. This
    procedure has no functionality associated with it. Because of this
    it is sometimes used to measure the overhead of processing a service
    request. Therefore, the server should ensure that no unnecessary
    work is done in servicing this procedure.

[15.2](#). Procedure 1: COMPOUND - Compound Operations

[15.2.1](#). SYNOPSIS

    compoundargs -> compoundres

[15.2.2](#). ARGUMENT

    union nfs_argop4 switch (nfs_opnum4 argop) {
            case <OPCODE>: <argument>;
            ...
    };

    struct COMPOUND4args {
            utf8str_cs   tag;
            uint32_t   minorversion;
            nfs_argop4   argarray<>;
    };

15.2.3. RESULT

```
   union nfs_resop4 switch (nfs_opnum4 resop) {
           case <OPCODE>: <argument>;
           ...
   };


   struct COMPOUND4res {
           nfsstat4        status;
           utf8str_cs      tag;
           nfs_resop4      resarray<>;
   };
```

15.2.4. DESCRIPTION

   The COMPOUND procedure is used to combine one or more of the NFS
   operations into a single RPC request. The main NFS RPC program has
   two main procedures: NULL and COMPOUND. All other operations use the
   COMPOUND procedure as a wrapper.

   The COMPOUND procedure is used to combine individual operations into
   a single RPC request. The server interprets each of the operations
   in turn. If an operation is executed by the server and the status of
   that operation is NFS4_OK, then the next operation in the COMPOUND
   procedure is executed. The server continues this process until there
   are no more operations to be executed or one of the operations has a
   status value other than NFS4_OK.

   In the processing of the COMPOUND procedure, the server may find that
   it does not have the available resources to execute any or all of the
   operations within the COMPOUND sequence. In this case, the error
   NFS4ERR_RESOURCE will be returned for the particular operation within
   the COMPOUND procedure where the resource exhaustion occurred. This
   assumes that all previous operations within the COMPOUND sequence
   have been evaluated successfully. The results for all of the
   evaluated operations must be returned to the client.

   The server will generally choose between two methods of decoding the
   client's request. The first would be the traditional one-pass XDR
   decode, in which decoding of the entire COMPOUND precedes execution
   of any operation within it. If there is an XDR decoding error in
   this case, an RPC XDR decode error would be returned. The second
   method would be to make an initial pass to decode the basic COMPOUND
   request and then to XDR decode each of the individual operations, as
   the server is ready to execute it. In this case, the server may
   encounter an XDR decode error during such an operation decode, after
   previous operations within the COMPOUND have been executed. In this

case, the server would return the error NFS4ERR_BADXDR to signify the
decode error.

The COMPOUND arguments contain a "minorversion" field. The initial
and default value for this field is 0 (zero). This field will be
used by future minor versions such that the client can communicate to
the server what minor version is being requested. If the server
receives a COMPOUND procedure with a minorversion field value that it
does not support, the server MUST return an error of
NFS4ERR_MINOR_VERS_MISMATCH and a zero length resultdata array.

Contained within the COMPOUND results is a "status" field. If the
results array length is non-zero, this status must be equivalent to
the status of the last operation that was executed within the
COMPOUND procedure. Therefore, if an operation incurred an error
then the "status" value will be the same error value as is being
returned for the operation that failed.

Note that operations, 0 (zero) and 1 (one) are not defined for the
COMPOUND procedure. Operation 2 is not defined but reserved for
future definition and use with minor versioning. If the server
receives a operation array that contains operation 2 and the
minorversion field has a value of 0 (zero), an error of
NFS4ERR_OP_ILLEGAL, as described in the next paragraph, is returned
to the client. If an operation array contains an operation 2 and the
minorversion field is non-zero and the server does not support the
minor version, the server returns an error of
NFS4ERR_MINOR_VERS_MISMATCH. Therefore, the
NFS4ERR_MINOR_VERS_MISMATCH error takes precedence over all other
errors.

It is possible that the server receives a request that contains an
operation that is less than the first legal operation (OP_ACCESS) or
greater than the last legal operation (OP_RELEASE_LOCKOWNER). In
this case, the server's response will encode the opcode OP_ILLEGAL
rather than the illegal opcode of the request. The status field in
the ILLEGAL return results will set to NFS4ERR_OP_ILLEGAL. The
COMPOUND procedure's return results will also be NFS4ERR_OP_ILLEGAL.

The definition of the "tag" in the request is left to the
implementor. It may be used to summarize the content of the compound
request for the benefit of packet sniffers and engineers debugging
implementations. However, the value of "tag" in the response SHOULD
be the same value as provided in the request. This applies to the
tag field of the CB_COMPOUND procedure as well.

15.2.4.1. Current Filehandle

   The current and saved filehandle are used throughout the protocol.
   Most operations implicitly use the current filehandle as a argument
   and many set the current filehandle as part of the results. The
   combination of client specified sequences of operations and current
   and saved filehandle arguments and results allows for greater
   protocol flexibility. The best or easiest example of current
   filehandle usage is a sequence like the following:

     PUTFH fh1   {fh1}
     LOOKUP "compA"   {fh2}
     GETATTR   {fh2}
     LOOKUP "compB"   {fh3}
     GETATTR   {fh3}
     LOOKUP "compC"   {fh4}
     GETATTR   {fh4}
     GETFH

                               Figure 1

   In this example, the PUTFH (Section 15.22) operation explicitly sets
   the current filehandle value while the result of each LOOKUP
   operation sets the current filehandle value to the resultant file
   system object. Also, the client is able to insert GETATTR operations
   using the current filehandle as an argument.

   The PUTROOTFH (Section 15.24) and PUTPUBFH (Section 15.24) operations
   also set the current filehandle. The above example would replace
   "PUTFH fh1" with PUTROOTFH or PUTPUBFH with no filehandle argument in
   order to achieve the same effect (on the assumption that "compA" is
   directly below the root of the namespace).

   Along with the current filehandle, there is a saved filehandle.
   While the current filehandle is set as the result of operations like
   LOOKUP, the saved filehandle must be set directly with the use of the
   SAVEFH operation. The SAVEFH operations copies the current
   filehandle value to the saved value. The saved filehandle value is
   used in combination with the current filehandle value for the LINK
   and RENAME operations. The RESTOREFH operation will copy the saved
   filehandle value to the current filehandle value; as a result, the
   saved filehandle value may be used a sort of "scratch" area for the
   client's series of operations.

[15.2.5](). IMPLEMENTATION

Since an error of any type may occur after only a portion of the operations have been evaluated, the client must be prepared to recover from any failure. If the source of an NFS4ERR_RESOURCE error was a complex or lengthy set of operations, it is likely that if the number of operations were reduced the server would be able to evaluate them successfully. Therefore, the client is responsible for dealing with this type of complexity in recovery.

The client SHOULD NOT construct a COMPOUND which mixes operations for different client IDs.

[15.3](). Operation 3: ACCESS - Check Access Rights

[15.3.1](). SYNOPSIS

(cfh), accessreq -> supported, accessrights

[15.3.2](). ARGUMENT

```
const ACCESS4_READ      = 0x00000001;
const ACCESS4_LOOKUP    = 0x00000002;
const ACCESS4_MODIFY    = 0x00000004;
const ACCESS4_EXTEND    = 0x00000008;
const ACCESS4_DELETE    = 0x00000010;
const ACCESS4_EXECUTE   = 0x00000020;

struct ACCESS4args {
        /* CURRENT_FH: object */
        uint32_t        access;
};
```

[15.3.3](#). RESULT

```
struct ACCESS4resok {
        uint32_t        supported;
        uint32_t        access;
};

union ACCESS4res switch (nfsstat4 status) {
 case NFS4_OK:
        ACCESS4resok   resok4;
 default:
        void;
};
```

[15.3.4](#). DESCRIPTION

ACCESS determines the access rights that a user, as identified by the
credentials in the RPC request, has with respect to the file system
object specified by the current filehandle. The client encodes the
set of access rights that are to be checked in the bit mask "access".
The server checks the permissions encoded in the bit mask. If a
status of NFS4_OK is returned, two bit masks are included in the
response. The first, "supported", represents the access rights for
which the server can verify reliably. The second, "access",
represents the access rights available to the user for the filehandle
provided. On success, the current filehandle retains its value.

Note that the supported field will contain only as many values as
were originally sent in the arguments. For example, if the client
sends an ACCESS operation with only the ACCESS4_READ value set and
the server supports this value, the server will return only
ACCESS4_READ even if it could have reliably checked other values.

The results of this operation are necessarily advisory in nature. A
return status of NFS4_OK and the appropriate bit set in the bit mask
does not imply that such access will be allowed to the file system
object in the future. This is because access rights can be revoked
by the server at any time.

The following access permissions may be requested:

ACCESS4_READ: Read data from file or read a directory.

ACCESS4_LOOKUP: Look up a name in a directory (no meaning for non-
   directory objects).

ACCESS4_MODIFY: Rewrite existing file data or modify existing
   directory entries.

ACCESS4_EXTEND: Write new data or add directory entries.

ACCESS4_DELETE: Delete an existing directory entry.

ACCESS4_EXECUTE: Execute file (no meaning for a directory).

On success, the current filehandle retains its value.

15.3.5. IMPLEMENTATION

In general, it is not sufficient for the client to attempt to deduce
access permissions by inspecting the uid, gid, and mode fields in the
file attributes or by attempting to interpret the contents of the ACL
attribute. This is because the server may perform uid or gid mapping
or enforce additional access control restrictions. It is also
possible that the server may not be in the same ID space as the
client. In these cases (and perhaps others), the client cannot
reliably perform an access check with only current file attributes.

In the NFSv2 protocol, the only reliable way to determine whether an
operation was allowed was to try it and see if it succeeded or
failed. Using the ACCESS operation in the NFSv4 protocol, the client
can ask the server to indicate whether or not one or more classes of
operations are permitted. The ACCESS operation is provided to allow
clients to check before doing a series of operations which might
result in an access failure. The OPEN operation provides a point
where the server can verify access to the file object and method to
return that information to the client. The ACCESS operation is still
useful for directory operations or for use in the case the UNIX API
"access" is used on the client.

The information returned by the server in response to an ACCESS call
is not permanent. It was correct at the exact time that the server
performed the checks, but not necessarily afterward. The server can
revoke access permission at any time.

The client should use the effective credentials of the user to build
the authentication information in the ACCESS request used to
determine access rights. It is the effective user and group
credentials that are used in subsequent read and write operations.

Many implementations do not directly support the ACCESS4_DELETE
permission. Operating systems like UNIX will ignore the
ACCESS4_DELETE bit if set on an access request on a non-directory
object. In these systems, delete permission on a file is determined

by the access permissions on the directory in which the file resides,
instead of being determined by the permissions of the file itself.
Therefore, the mask returned enumerating which access rights can be
determined will have the ACCESS4_DELETE value set to 0. This
indicates to the client that the server was unable to check that
particular access right. The ACCESS4_DELETE bit in the access mask
returned will then be ignored by the client.

15.4. Operation 4: CLOSE - Close File

15.4.1. SYNOPSIS

    (cfh), seqid, open_stateid -> open_stateid

15.4.2. ARGUMENT

```
struct CLOSE4args {
        /* CURRENT_FH: object */
        seqid4          seqid;
        stateid4        open_stateid;
};
```

15.4.3. RESULT

```
union CLOSE4res switch (nfsstat4 status) {
 case NFS4_OK:
        stateid4        open_stateid;
 default:
        void;
};
```

15.4.4. DESCRIPTION

The CLOSE operation releases share reservations for the regular or
named attribute file as specified by the current filehandle. The
share reservations and other state information released at the server
as a result of this CLOSE is only associated with the supplied
stateid. The sequence id provides for the correct ordering. State
associated with other OPENs is not affected.

If byte-range locks are held, the client SHOULD release all locks
before issuing a CLOSE. The server MAY free all outstanding locks on
CLOSE but some servers may not support the CLOSE of a file that still
has byte-range locks held. The server MUST return failure if any
locks would exist after the CLOSE.

On success, the current filehandle retains its value.

[15.4.5](#). IMPLEMENTATION

Even though CLOSE returns a stateid, this stateid is not useful to
the client and should be treated as deprecated. CLOSE "shuts down"
the state associated with all OPENs for the file by a single open-
owner. As noted above, CLOSE will either release all file locking
state or return an error. Therefore, the stateid returned by CLOSE
is not useful for operations that follow.

[15.5](#). Operation 5: COMMIT - Commit Cached Data

[15.5.1](#). SYNOPSIS

   (cfh), offset, count -> verifier

[15.5.2](#). ARGUMENT

```
struct COMMIT4args {
        /* CURRENT_FH: file */
        offset4         offset;
        count4          count;
};
```

[15.5.3](#). RESULT

```
struct COMMIT4resok {
        verifier4       writeverf;
};

union COMMIT4res switch (nfsstat4 status) {
 case NFS4_OK:
        COMMIT4resok   resok4;
 default:
        void;
};
```

[15.5.4](#). DESCRIPTION

The COMMIT operation forces or flushes data to stable storage for the
file specified by the current filehandle. The flushed data is that
which was previously written with a WRITE operation which had the
stable field set to UNSTABLE4.

The offset specifies the position within the file where the flush is

to begin. An offset value of 0 (zero) means to flush data starting
at the beginning of the file. The count specifies the number of
bytes of data to flush. If count is 0 (zero), a flush from offset to
the end of the file is done.

The server returns a write verifier upon successful completion of the
COMMIT. The write verifier is used by the client to determine if the
server has restarted or rebooted between the initial WRITE(s) and the
COMMIT. The client does this by comparing the write verifier
returned from the initial writes and the verifier returned by the
COMMIT operation. The server must vary the value of the write
verifier at each server event or instantiation that may lead to a
loss of uncommitted data. Most commonly this occurs when the server
is rebooted; however, other events at the server may result in
uncommitted data loss as well.

On success, the current filehandle retains its value.

15.5.5. IMPLEMENTATION

The COMMIT operation is similar in operation and semantics to the
POSIX fsync() [fsync] system call that synchronizes a file's state
with the disk (file data and metadata is flushed to disk or stable
storage). COMMIT performs the same operation for a client, flushing
any unsynchronized data and metadata on the server to the server's
disk or stable storage for the specified file. Like fsync(), it may
be that there is some modified data or no modified data to
synchronize. The data may have been synchronized by the server's
normal periodic buffer synchronization activity. COMMIT should
return NFS4_OK, unless there has been an unexpected error.

COMMIT differs from fsync() in that it is possible for the client to
flush a range of the file (most likely triggered by a buffer-
reclamation scheme on the client before file has been completely
written).

The server implementation of COMMIT is reasonably simple. If the
server receives a full file COMMIT request, that is starting at
offset 0 and count 0, it should do the equivalent of fsync()'ing the
file. Otherwise, it should arrange to have the cached data in the
range specified by offset and count to be flushed to stable storage.
In both cases, any metadata associated with the file must be flushed
to stable storage before returning. It is not an error for there to
be nothing to flush on the server. This means that the data and
metadata that needed to be flushed have already been flushed or lost
during the last server failure.

The client implementation of COMMIT is a little more complex. There

are two reasons for wanting to commit a client buffer to stable
storage. The first is that the client wants to reuse a buffer. In
this case, the offset and count of the buffer are sent to the server
in the COMMIT request. The server then flushes any cached data based
on the offset and count, and flushes any metadata associated with the
file. It then returns the status of the flush and the write
verifier. The other reason for the client to generate a COMMIT is
for a full file flush, such as may be done at close. In this case,
the client would gather all of the buffers for this file that contain
uncommitted data, do the COMMIT operation with an offset of 0 and
count of 0, and then free all of those buffers. Any other dirty
buffers would be sent to the server in the normal fashion.

After a buffer is written by the client with the stable parameter set
to UNSTABLE4, the buffer must be considered as modified by the client
until the buffer has either been flushed via a COMMIT operation or
written via a WRITE operation with stable parameter set to FILE_SYNC4
or DATA_SYNC4. This is done to prevent the buffer from being freed
and reused before the data can be flushed to stable storage on the
server.

When a response is returned from either a WRITE or a COMMIT operation
and it contains a write verifier that is different than previously
returned by the server, the client will need to retransmit all of the
buffers containing uncommitted cached data to the server. How this
is to be done is up to the implementor. If there is only one buffer
of interest, then it should probably be sent back over in a WRITE
request with the appropriate stable parameter. If there is more than
one buffer, it might be worthwhile retransmitting all of the buffers
in WRITE requests with the stable parameter set to UNSTABLE4 and then
retransmitting the COMMIT operation to flush all of the data on the
server to stable storage. The timing of these retransmissions is
left to the implementor.

The above description applies to page-cache-based systems as well as
buffer-cache-based systems. In those systems, the virtual memory
system will need to be modified instead of the buffer cache.

15.6. Operation 6: CREATE - Create a Non-Regular File Object

15.6.1. SYNOPSIS

   (cfh), name, type, attrs -> (cfh), cinfo, attrset

[15.6.2](#). **ARGUMENT**

```
   union createtype4 switch (nfs_ftype4 type) {
    case NF4LNK:
            linktext4 linkdata;
    case NF4BLK:
    case NF4CHR:
            specdata4 devdata;
    case NF4SOCK:
    case NF4FIFO:
    case NF4DIR:
            void;
    default:
            void;  /* server should return NFS4ERR_BADTYPE */
   };

   struct CREATE4args {
           /* CURRENT_FH: directory for creation */
           createtype4     objtype;
           component4      objname;
           fattr4          createattrs;
   };
```

[15.6.3](#). **RESULT**

```
   struct CREATE4resok {
           change_info4    cinfo;
           bitmap4         attrset;        /* attributes set */
   };

   union CREATE4res switch (nfsstat4 status) {
    case NFS4_OK:
            CREATE4resok resok4;
    default:
            void;
   };
```

[15.6.4](#). **DESCRIPTION**

The CREATE operation creates a non-regular file object in a directory
with a given name. The OPEN operation MUST be used to create a
regular file.

The objname specifies the name for the new object. The objtype
determines the type of object to be created: directory, symlink, etc.

   If an object of the same name already exists in the directory, the
   server will return the error NFS4ERR_EXIST.

   For the directory where the new file object was created, the server
   returns change_info4 information in cinfo. With the atomic field of
   the change_info4 struct, the server will indicate if the before and
   after change attributes were obtained atomically with respect to the
   file object creation.

   If the objname is of zero length, NFS4ERR_INVAL will be returned.
   The objname is also subject to the normal UTF-8, character support,
   and name checks. See Section 12.5 for further discussion.

   If the objname has a length of 0 (zero), or if objname does not obey
   the UTF-8 definition, the error NFS4ERR_INVAL will be returned.

   The current filehandle is replaced by that of the new object.

   The createattrs specifies the initial set of attributes for the
   object. The set of attributes may include any writable attribute
   valid for the object type. When the operation is successful, the
   server will return to the client an attribute mask signifying which
   attributes were successfully set for the object.

   If createattrs includes neither the owner attribute nor an ACL with
   an ACE for the owner, and if the server's file system both supports
   and requires an owner attribute (or an owner ACE) then the server
   MUST derive the owner (or the owner ACE). This would typically be
   from the principal indicated in the RPC credentials of the call, but
   the server's operating environment or file system semantics may
   dictate other methods of derivation. Similarly, if createattrs
   includes neither the group attribute nor a group ACE, and if the
   server's file system both supports and requires the notion of a group
   attribute (or group ACE), the server MUST derive the group attribute
   (or the corresponding owner ACE) for the file. This could be from
   the RPC call's credentials, such as the group principal if the
   credentials include it (such as with AUTH_SYS), from the group
   identifier associated with the principal in the credentials (e.g.,
   POSIX systems have a user database [getpwnam] that has the group
   identifier for every user identifier), inherited from directory the
   object is created in, or whatever else the server's operating
   environment or file system semantics dictate. This applies to the
   OPEN operation too.

   Conversely, it is possible the client will specify in createattrs an
   owner attribute or group attribute or ACL that the principal
   indicated the RPC call's credentials does not have permissions to
   create files for. The error to be returned in this instance is

NFS4ERR_PERM.   This applies to the OPEN operation too.

15.6.5. IMPLEMENTATION

If the client desires to set attribute values after the create, a
SETATTR operation can be added to the COMPOUND request so that the
appropriate attributes will be set.

15.7. Operation 7: DELEGPURGE - Purge Delegations Awaiting Recovery

15.7.1. SYNOPSIS

```
   clientid ->
```

15.7.2. ARGUMENT

```
struct DELEGPURGE4args {
        clientid4       clientid;
};
```

15.7.3. RESULT

```
struct DELEGPURGE4res {
        nfsstat4        status;
};
```

15.7.4. DESCRIPTION

Purges all of the delegations awaiting recovery for a given client.
This is useful for clients which do not commit delegation information
to stable storage to indicate that conflicting requests need not be
delayed by the server awaiting recovery of delegation information.

This operation in provided to support clients that record delegation
information on stable storage on the client. In this case,
DELEGPURGE should be issued immediately after doing delegation
recovery (using CLAIM_DELEGATE_PREV) on all delegations known to the
client. Doing so will notify the server that no additional
delegations for the client will be recovered allowing it to free
resources, and avoid delaying other clients who make requests that
conflict with the unrecovered delegations. All client SHOULD use
DELEGPURGE as part of recovery once it is known that no further
CLAIM_DELEGATE_PREV recovery will be done. This includes clients
that do not record delegation information on stable storage, who
would then do a DELEGPURGE immediately after SETCLIENTID_CONFIRM.

The set of delegations known to the server and the client may be
different. The reasons for this include:

o A client may fail after making a request which resulted in
   delegation but before it received the results and committed them
   to the client's stable storage.

o A client may fail after deleting its indication that a delegation
   exists but before the delegation return is fully processed by the
   server.

o In the case in which the server and the client restart, the server
   may have limited persistent recording of delegation to a subset of
   those in existence.

o A client may have only persistently recorded information about a
   subset of delegations.

The server MAY support DELEGPURGE, but its support or non-support
should match that of CLAIM_DELEGATE_PREV:

o A server may support both DELEGPURGE and CLAIM_DELEGATE_PREV.

o A server may support neither DELEGPURGE nor CLAIM_DELEGATE_PREV.

This fact allows a client starting up to determine if the server is
prepared to support persistent storage of delegation information and
thus whether it may use write-back caching to local persistent
storage, relying on CLAIM_DELEGATE_PREV recovery to allow such
changed data to be flushed safely to the server in the event of
client restart.

15.8. Operation 8: DELEGRETURN - Return Delegation

15.8.1. SYNOPSIS

   (cfh), stateid ->

15.8.2. ARGUMENT

```
struct DELEGRETURN4args {
        /* CURRENT_FH: delegated file */
        stateid4        deleg_stateid;
};
```

[15.8.3](#). RESULT

```
struct DELEGRETURN4res {
        nfsstat4         status;
};
```

[15.8.4](#). DESCRIPTION

Returns the delegation represented by the current filehandle and
stateid.

Delegations may be returned when recalled or voluntarily (i.e.,
before the server has recalled them). In either case the client must
properly propagate state changed under the context of the delegation
to the server before returning the delegation.

[15.9](#). Operation 9: GETATTR - Get Attributes

[15.9.1](#). SYNOPSIS

```
   (cfh), attrbits -> attrbits, attrvals
```

[15.9.2](#). ARGUMENT

```
struct GETATTR4args {
        /* CURRENT_FH: directory or file */
        bitmap4          attr_request;
};
```

[15.9.3](#). RESULT

```
struct GETATTR4resok {
        fattr4           obj_attributes;
};

union GETATTR4res switch (nfsstat4 status) {
 case NFS4_OK:
        GETATTR4resok  resok4;
 default:
        void;
};
```

15.9.4. DESCRIPTION

   The GETATTR operation will obtain attributes for the file system
   object specified by the current filehandle. The client sets a bit in
   the bitmap argument for each attribute value that it would like the
   server to return. The server returns an attribute bitmap that
   indicates the attribute values for which it was able to return
   values, followed by the attribute values ordered lowest attribute
   number first.

   The server MUST return a value for each attribute that the client
   requests if the attribute is supported by the server. If the server
   does not support an attribute or cannot approximate a useful value
   then it MUST NOT return the attribute value and MUST NOT set the
   attribute bit in the result bitmap. The server MUST return an error
   if it supports an attribute on the target but cannot obtain its
   value. In that case no attribute values will be returned.

   File systems which are absent should be treated as having support for
   a very small set of attributes as described in GETATTR Within an
   Absent File System (Section 8.3.1), even if previously, when the file
   system was present, more attributes were supported.

   All servers MUST support the REQUIRED attributes as specified in the
   section File Attributes (Section 5), for all file systems, with the
   exception of absent file systems.

   On success, the current filehandle retains its value.

15.9.5. IMPLEMENTATION

   Suppose there is a OPEN_DELEGATE_WRITE delegation held by another
   client for file in question and size and/or change are among the set
   of attributes being interrogated. The server has two choices.
   First, the server can obtain the actual current value of these
   attributes from the client holding the delegation by using the
   CB_GETATTR callback. Second, the server, particularly when the
   delegated client is unresponsive, can recall the delegation in
   question. The GETATTR MUST NOT proceed until one of the following
   occurs:

   o The requested attribute values are returned in the response to
      CB_GETATTR.

   o The OPEN_DELEGATE_WRITE delegation is returned.

   o The OPEN_DELEGATE_WRITE delegation is revoked.

    Unless one of the above happens very quickly, one or more
    NFS4ERR_DELAY errors will be returned while a delegation is
    outstanding.

[15.10](#). Operation 10: GETFH - Get Current Filehandle

[15.10.1](#). SYNOPSIS

    (cfh) -> filehandle

[15.10.2](#). ARGUMENT

    /* CURRENT_FH: */
    void;

[15.10.3](#). RESULT

```
    struct GETFH4resok {
            nfs_fh4         object;
    };

    union GETFH4res switch (nfsstat4 status) {
     case NFS4_OK:
            GETFH4resok     resok4;
     default:
            void;
    };
```

[15.10.4](#). DESCRIPTION

    This operation returns the current filehandle value.

    On success, the current filehandle retains its value.

[15.10.5](#). IMPLEMENTATION

    Operations that change the current filehandle like LOOKUP or CREATE
    do not automatically return the new filehandle as a result. For
    instance, if a client needs to lookup a directory entry and obtain
    its filehandle then the following request is needed.

```
      PUTFH  (directory filehandle)
      LOOKUP (entry name)
      GETFH
```

[15.11](#). **Operation 11: LINK - Create Link to a File**

[15.11.1](#). **SYNOPSIS**

```
   (sfh), (cfh), newname -> (cfh), cinfo
```

[15.11.2](#). **ARGUMENT**

```
   struct LINK4args {
           /* SAVED_FH: source object */
           /* CURRENT_FH: target directory */
           component4      newname;
   };
```

[15.11.3](#). **RESULT**

```
   struct LINK4resok {
           change_info4    cinfo;
   };

   union LINK4res switch (nfsstat4 status) {
    case NFS4_OK:
           LINK4resok resok4;
    default:
           void;
   };
```

[15.11.4](#). **DESCRIPTION**

   The LINK operation creates an additional newname for the file
   represented by the saved filehandle, as set by the SAVEFH operation,
   in the directory represented by the current filehandle. The existing
   file and the target directory must reside within the same file system
   on the server. On success, the current filehandle will continue to
   be the target directory. If an object exists in the target directory
   with the same name as newname, the server must return NFS4ERR_EXIST.

   For the target directory, the server returns change_info4 information
   in cinfo. With the atomic field of the change_info4 struct, the
   server will indicate if the before and after change attributes were
   obtained atomically with respect to the link creation.

   If the newname has a length of 0 (zero), or if newname does not obey
   the UTF-8 definition, the error NFS4ERR_INVAL will be returned.

15.11.5. **IMPLEMENTATION**

   Changes to any property of the "hard" linked files are reflected in
   all of the linked files. When a link is made to a file, the
   attributes for the file should have a value for numlinks that is one
   greater than the value before the LINK operation.

   The statement "file and the target directory must reside within the
   same file system on the server" means that the fsid fields in the
   attributes for the objects are the same. If they reside on different
   file systems, the error, NFS4ERR_XDEV, is returned. On some servers,
   the filenames, "." and "..", are illegal as newname.

   In the case that newname is already linked to the file represented by
   the saved filehandle, the server will return NFS4ERR_EXIST.

   Note that symbolic links are created with the CREATE operation.

15.12. **Operation 12: LOCK - Create Lock**

15.12.1. **SYNOPSIS**

    (cfh) locktype, reclaim, offset, length, locker -> stateid

15.12.2. **ARGUMENT**

   enum nfs_lock_type4 {
           READ_LT   = 1,
           WRITE_LT   = 2,
           READW_LT   = 3,   /* blocking read */
           WRITEW_LT   = 4   /* blocking write */
   };

```
   /*
    * For LOCK, transition from open_owner to new lock_owner
    */
   struct open_to_lock_owner4 {
           seqid4          open_seqid;
           stateid4        open_stateid;
           seqid4          lock_seqid;
           lock_owner4     lock_owner;
   };

   /*
    * For LOCK, existing lock_owner continues to request file locks
    */
   struct exist_lock_owner4 {
           stateid4        lock_stateid;
           seqid4          lock_seqid;
   };

   union locker4 switch (bool new_lock_owner) {
    case TRUE:
           open_to_lock_owner4     open_owner;
    case FALSE:
           exist_lock_owner4       lock_owner;
   };

   /*
    * LOCK/LOCKT/LOCKU: Record lock management
    */
   struct LOCK4args {
           /* CURRENT_FH: file */
           nfs_lock_type4  locktype;
           bool            reclaim;
           offset4         offset;
           length4         length;
           locker4         locker;
   };
```

[15.12.3](#). RESULT

```
struct LOCK4denied {
        offset4         offset;
        length4         length;
        nfs_lock_type4  locktype;
        lock_owner4     owner;
};

struct LOCK4resok {
        stateid4        lock_stateid;
};

union LOCK4res switch (nfsstat4 status) {
 case NFS4_OK:
        LOCK4resok      resok4;
 case NFS4ERR_DENIED:
        LOCK4denied     denied;
 default:
        void;
};
```

[15.12.4](#). DESCRIPTION

The LOCK operation requests a byte-range lock for the byte range
specified by the offset and length parameters. The lock type is also
specified to be one of the nfs_lock_type4s. If this is a reclaim
request, the reclaim parameter will be TRUE;

Bytes in a file may be locked even if those bytes are not currently
allocated to the file. To lock the file from a specific offset
through the end-of-file (no matter how long the file actually is) use
a length field with all bits set to 1 (one). If the length is zero,
or if a length which is not all bits set to one is specified, and
length when added to the offset exceeds the maximum 64-bit unsigned
integer value, the error NFS4ERR_INVAL will result.

Some servers may only support locking for byte offsets that fit
within 32 bits. If the client specifies a range that includes a byte
beyond the last byte offset of the 32-bit range, but does not include
the last byte offset of the 32-bit and all of the byte offsets beyond
it, up to the end of the valid 64-bit range, such a 32-bit server
MUST return the error NFS4ERR_BAD_RANGE.

In the case that the lock is denied, the owner, offset, and length of
a conflicting lock are returned.

On success, the current filehandle retains its value.

15.12.5. IMPLEMENTATION

If the server is unable to determine the exact offset and length of
the conflicting lock, the same offset and length that were provided
in the arguments should be returned in the denied results. Section 9
contains a full description of this and the other file locking
operations.

LOCK operations are subject to permission checks and to checks
against the access type of the associated file. However, the
specific right and modes required for various type of locks, reflect
the semantics of the server-exported file system, and are not
specified by the protocol. For example, Windows 2000 allows a write
lock of a file open for READ, while a POSIX-compliant system does
not.

When the client makes a lock request that corresponds to a range that
the lock-owner has locked already (with the same or different lock
type), or to a sub-region of such a range, or to a region which
includes multiple locks already granted to that lock-owner, in whole
or in part, and the server does not support such locking operations
(i.e., does not support POSIX locking semantics), the server will
return the error NFS4ERR_LOCK_RANGE. In that case, the client may
return an error, or it may emulate the required operations, using
only LOCK for ranges that do not include any bytes already locked by
that lock-owner and LOCKU of locks held by that lock-owner
(specifying an exactly-matching range and type). Similarly, when the
client makes a lock request that amounts to upgrading (changing from
a read lock to a write lock) or downgrading (changing from write lock
to a read lock) an existing record lock, and the server does not
support such a lock, the server will return NFS4ERR_LOCK_NOTSUPP.
Such operations may not perfectly reflect the required semantics in
the face of conflicting lock requests from other clients.

When a client holds an OPEN_DELEGATE_WRITE delegation, the client
holding that delegation is assured that there are no opens by other
clients. Thus, there can be no conflicting LOCK operations from such
clients. Therefore, the client may be handling locking requests
locally, without doing LOCK operations on the server. If it does
that, it must be prepared to update the lock status on the server, by
sending appropriate LOCK and LOCKU operations before returning the
delegation.

When one or more clients hold OPEN_DELEGATE_READ delegations, any
LOCK operation where the server is implementing mandatory locking
semantics MUST result in the recall of all such delegations. The

LOCK operation may not be granted until all such delegations are
returned or revoked. Except where this happens very quickly, one or
more NFS4ERR_DELAY errors will be returned to requests made while the
delegation remains outstanding.

The locker argument specifies the lock-owner that is associated with
the LOCK request. The locker4 structure is a switched union that
indicates whether the client has already created byte-range locking
state associated with the current open file and lock-owner. There
are multiple cases to be considered, corresponding to possible
combinations of whether locking state has been created for the
current open file and lock-owner, and whether the boolean
new_lock_owner is set. In all of the cases, there is a lock_seqid
specified, whether the lock-owner is specified explicitly or
implicitly. This seqid value is used for checking lock-owner
sequencing/replay issues. When the given lock-owner is not known to
the server, this establishes an initial sequence value for the new
lock-owner.

o   In the case in which the state has been created and the boolean is
    false, the only part of the argument other than lock_seqid is just
    a stateid representing the set of locks associated with that open
    file and lock-owner.

o   In the case in which the state has been created and the boolean is
    true, the server rejects the request with the error
    NFS4ERR_BAD_SEQID. The only exception is where there is a
    retransmission of a previous request in which the boolean was
    true. In this case, the lock_seqid will match the original
    request and the response will reflect the final case, below.

o   In the case where no byte-range locking state has been established
    and the boolean is true, the argument contains an
    open_to_lock_owner structure which specifies the stateid of the
    open file and the lock-owner to be used for the lock. Note that
    although the open-owner is not given explicitly, the open_seqid
    associated with it is used to check for open-owner sequencing
    issues. This case provides a method to use the established state
    of the open_stateid to transition to the use of a lock stateid.

15.13. Operation 13: LOCKT - Test For Lock

15.13.1. SYNOPSIS

    (cfh) locktype, offset, length, owner -> {void, NFS4ERR_DENIED ->
    owner}

[15.13.2](#).   ARGUMENT

```
   struct LOCKT4args {
           /* CURRENT_FH: file */
           nfs_lock_type4  locktype;
           offset4         offset;
           length4         length;
           lock_owner4     owner;
   };
```

[15.13.3](#).   RESULT

```
   union LOCKT4res switch (nfsstat4 status) {
    case NFS4ERR_DENIED:
           LOCK4denied     denied;
    case NFS4_OK:
           void;
    default:
           void;
   };
```

[15.13.4](#).   DESCRIPTION

   The LOCKT operation tests the lock as specified in the arguments. If
   a conflicting lock exists, the owner, offset, length, and type of the
   conflicting lock are returned; if no lock is held, nothing other than
   NFS4_OK is returned. Lock types READ_LT and READW_LT are processed
   in the same way in that a conflicting lock test is done without
   regard to blocking or non-blocking. The same is true for WRITE_LT
   and WRITEW_LT.

   The ranges are specified as for LOCK. The NFS4ERR_INVAL and
   NFS4ERR_BAD_RANGE errors are returned under the same circumstances as
   for LOCK.

   On success, the current filehandle retains its value.

[15.13.5](#).   IMPLEMENTATION

   If the server is unable to determine the exact offset and length of
   the conflicting lock, the same offset and length that were provided
   in the arguments should be returned in the denied results. [Section 9](#)
   contains further discussion of the file locking mechanisms.

   LOCKT uses a lock_owner4 rather a stateid4, as is used in LOCK to
   identify the owner. This is because the client does not have to open

the file to test for the existence of a lock, so a stateid may not be
available.

The test for conflicting locks SHOULD exclude locks for the current
lock-owner. Note that since such locks are not examined the possible
existence of overlapping ranges may not affect the results of LOCKT.
If the server does examine locks that match the lock-owner for the
purpose of range checking, NFS4ERR_LOCK_RANGE may be returned. In
the event that it returns NFS4_OK, clients may do a LOCK and receive
NFS4ERR_LOCK_RANGE on the LOCK request because of the flexibility
provided to the server.

When a client holds an OPEN_DELEGATE_WRITE delegation, it may choose
(see Section 15.12.5)) to handle LOCK requests locally. In such a
case, LOCKT requests will similarly be handled locally.

15.14. Operation 14: LOCKU - Unlock File

15.14.1. SYNOPSIS

   (cfh) type, seqid, stateid, offset, length -> stateid

15.14.2. ARGUMENT

```
struct LOCKU4args {
        /* CURRENT_FH: file */
        nfs_lock_type4  locktype;
        seqid4          seqid;
        stateid4        lock_stateid;
        offset4         offset;
        length4         length;
};
```

15.14.3. RESULT

```
union LOCKU4res switch (nfsstat4 status) {
 case   NFS4_OK:
        stateid4        lock_stateid;
 default:
        void;
};
```

[15.14.4](#). **DESCRIPTION**

   The LOCKU operation unlocks the byte-range lock specified by the
   parameters. The client may set the locktype field to any value that
   is legal for the nfs_lock_type4 enumerated type, and the server MUST
   accept any legal value for locktype. Any legal value for locktype
   has no effect on the success or failure of the LOCKU operation.

   The ranges are specified as for LOCK. The NFS4ERR_INVAL and
   NFS4ERR_BAD_RANGE errors are returned under the same circumstances as
   for LOCK.

   On success, the current filehandle retains its value.

[15.14.5](#). **IMPLEMENTATION**

   If the area to be unlocked does not correspond exactly to a lock
   actually held by the lock-owner the server may return the error
   NFS4ERR_LOCK_RANGE. This includes the case in which the area is not
   locked, where the area is a sub-range of the area locked, where it
   overlaps the area locked without matching exactly or the area
   specified includes multiple locks held by the lock-owner. In all of
   these cases, allowed by POSIX locking [fcntl] semantics, a client
   receiving this error, should if it desires support for such
   operations, simulate the operation using LOCKU on ranges
   corresponding to locks it actually holds, possibly followed by LOCK
   requests for the sub-ranges not being unlocked.

   When a client holds an OPEN_DELEGATE_WRITE delegation, it may choose
   (see [Section 15.12.5](#))) to handle LOCK requests locally. In such a
   case, LOCKU requests will similarly be handled locally.

[15.15](#). **Operation 15: LOOKUP - Lookup Filename**

[15.15.1](#). **SYNOPSIS**

      (cfh), component -> (cfh)

[15.15.2](#). **ARGUMENT**

   struct LOOKUP4args {
           /* CURRENT_FH: directory */
           component4   objname;
   };

[15.15.3](#). **RESULT**

```
struct LOOKUP4res {
        /* CURRENT_FH: object */
        nfsstat4        status;
};
```

[15.15.4](#). **DESCRIPTION**

This operation LOOKUPs or finds a file system object using the
directory specified by the current filehandle. LOOKUP evaluates the
component and if the object exists the current filehandle is replaced
with the component's filehandle.

If the component cannot be evaluated either because it does not exist
or because the client does not have permission to evaluate the
component, then an error will be returned and the current filehandle
will be unchanged.

If the component is of zero length, NFS4ERR_INVAL will be returned.
The component is also subject to the normal UTF-8, character support,
and name checks. See [Section 12.5](#) for further discussion.

[15.15.5](#). **IMPLEMENTATION**

If the client wants to achieve the effect of a multi-component
lookup, it may construct a COMPOUND request such as (and obtain each
filehandle):

```
  PUTFH  (directory filehandle)
  LOOKUP "pub"
  GETFH
  LOOKUP "foo"
  GETFH
  LOOKUP "bar"
  GETFH
```

NFSv4 servers depart from the semantics of previous NFS versions in
allowing LOOKUP requests to cross mount points on the server. The
client can detect a mount point crossing by comparing the fsid
attribute of the directory with the fsid attribute of the directory
looked up. If the fsids are different then the new directory is a
server mount point. UNIX clients that detect a mount point crossing
will need to mount the server's file system. This needs to be done
to maintain the file object identity checking mechanisms common to
UNIX clients.

Servers that limit NFS access to "shares" or "exported" file systems
should provide a pseudo-file system into which the exported file
systems can be integrated, so that clients can browse the server's
name space. The clients' view of a pseudo file system will be
limited to paths that lead to exported file systems.

Note: previous versions of the protocol assigned special semantics to
the names "." and "..". NFSv4 assigns no special semantics to these
names. The LOOKUPP operator must be used to lookup a parent
directory.

Note that this operation does not follow symbolic links. The client
is responsible for all parsing of filenames including filenames that
are modified by symbolic links encountered during the lookup process.

If the current filehandle supplied is not a directory but a symbolic
link, the error NFS4ERR_SYMLINK is returned as the error. For all
other non-directory file types, the error NFS4ERR_NOTDIR is returned.

15.16. Operation 16: LOOKUPP - Lookup Parent Directory

15.16.1. SYNOPSIS

```
(cfh) -> (cfh)
```

15.16.2. ARGUMENT

```
/* CURRENT_FH: object */
void;
```

15.16.3. RESULT

```
struct LOOKUPP4res {
        /* CURRENT_FH: directory */
        nfsstat4        status;
};
```

15.16.4. DESCRIPTION

The current filehandle is assumed to refer to a regular directory or
a named attribute directory. LOOKUPP assigns the filehandle for its
parent directory to be the current filehandle. If there is no parent
directory an NFS4ERR_NOENT error must be returned. Therefore,
NFS4ERR_NOENT will be returned by the server when the current
filehandle is at the root or top of the server's file tree.

15.16.5. **IMPLEMENTATION**

   As for LOOKUP, LOOKUPP will also cross mount points.

   If the current filehandle is not a directory or named attribute
   directory, the error NFS4ERR_NOTDIR is returned.

15.17. **Operation 17: NVERIFY - Verify Difference in Attributes**

15.17.1. **SYNOPSIS**

   (cfh), fattr -> -

15.17.2. **ARGUMENT**

```
struct NVERIFY4args {
        /* CURRENT_FH: object */
        fattr4          obj_attributes;
};
```

15.17.3. **RESULT**

```
struct NVERIFY4res {
        nfsstat4        status;
};
```

15.17.4. **DESCRIPTION**

   This operation is used to prefix a sequence of operations to be
   performed if one or more attributes have changed on some file system
   object. If all the attributes match then the error NFS4ERR_SAME must
   be returned.

   On success, the current filehandle retains its value.

15.17.5. **IMPLEMENTATION**

   This operation is useful as a cache validation operator. If the
   object to which the attributes belong has changed then the following
   operations may obtain new data associated with that object. For
   instance, to check if a file has been changed and obtain new data if
   it has:

```
   PUTFH  (public)
   LOOKUP "foobar"
   NVERIFY attrbits attrs
```

```
   READ 0 32767
```

In the case that a recommended attribute is specified in the NVERIFY
operation and the server does not support that attribute for the file
system object, the error NFS4ERR_ATTRNOTSUPP is returned to the
client.

When the attribute rdattr_error or any write-only attribute (e.g.,
time_modify_set) is specified, the error NFS4ERR_INVAL is returned to
the client.

15.18. Operation 18: OPEN - Open a Regular File

15.18.1. SYNOPSIS

```
   (cfh), seqid, share_access, share_deny, owner, openhow, claim ->
   (cfh), stateid, cinfo, rflags, attrset, delegation
```

15.18.2. ARGUMENT

```
  /*
   * Various definitions for OPEN
   */
  enum createmode4 {
          UNCHECKED4      = 0,
          GUARDED4        = 1,
          EXCLUSIVE4      = 2
  };

  union createhow4 switch (createmode4 mode) {
   case UNCHECKED4:
   case GUARDED4:
          fattr4          createattrs;
   case EXCLUSIVE4:
          verifier4       createverf;
  };

  enum opentype4 {
          OPEN4_NOCREATE  = 0,
          OPEN4_CREATE    = 1
  };

  union openflag4 switch (opentype4 opentype) {
   case OPEN4_CREATE:
          createhow4      how;
   default:
          void;
  };
```

```
   /* Next definitions used for OPEN delegation */
   enum limit_by4 {
          NFS_LIMIT_SIZE          = 1,
          NFS_LIMIT_BLOCKS        = 2
          /* others as needed */
   };

   struct nfs_modified_limit4 {
          uint32_t          num_blocks;
          uint32_t          bytes_per_block;
   };

   union nfs_space_limit4 switch (limit_by4 limitby) {
    /* limit specified as file size */
    case NFS_LIMIT_SIZE:
           uint64_t                 filesize;
    /* limit specified by number of blocks */
    case NFS_LIMIT_BLOCKS:
           nfs_modified_limit4    mod_blocks;
   } ;

   enum open_delegation_type4 {
          OPEN_DELEGATE_NONE      = 0,
          OPEN_DELEGATE_READ      = 1,
          OPEN_DELEGATE_WRITE     = 2
   };

   enum open_claim_type4 {
          CLAIM_NULL              = 0,
          CLAIM_PREVIOUS          = 1,
          CLAIM_DELEGATE_CUR      = 2,
          CLAIM_DELEGATE_PREV     = 3
   };

   struct open_claim_delegate_cur4 {
          stateid4          delegate_stateid;
          component4      file;
   };

   union open_claim4 switch (open_claim_type4 claim) {
    /*
     * No special rights to file.
     * Ordinary OPEN of the specified file.
     */
    case CLAIM_NULL:
           /* CURRENT_FH: directory */
           component4      file;
    /*
```

```
      * Right to the file established by an
      * open previous to server reboot. File
      * identified by filehandle obtained at
      * that time rather than by name.
      */
     case CLAIM_PREVIOUS:
            /* CURRENT_FH: file being reclaimed */
            open_delegation_type4   delegate_type;

      /*
       * Right to file based on a delegation
       * granted by the server. File is
       * specified by name.
       */
     case CLAIM_DELEGATE_CUR:
            /* CURRENT_FH: directory */
            open_claim_delegate_cur4         delegate_cur_info;

      /*
       * Right to file based on a delegation
       * granted to a previous boot instance
       * of the client.  File is specified by name.
       */
     case CLAIM_DELEGATE_PREV:
             /* CURRENT_FH: directory */
            component4      file_delegate_prev;
    };

    /*
     * OPEN: Open a file, potentially receiving an open delegation
     */
    struct OPEN4args {
            seqid4          seqid;
            uint32_t        share_access;
            uint32_t        share_deny;
            open_owner4     owner;
            openflag4       openhow;
            open_claim4     claim;
    };
```

15.18.3. RESULT

```
    struct open_read_delegation4 {
     stateid4 stateid;     /* Stateid for delegation*/
     bool     recall;      /* Pre-recalled flag for
                              delegations obtained
                              by reclaim (CLAIM_PREVIOUS) */
```

```
    nfsace4 permissions; /* Defines users who don't
                            need an ACCESS call to
                            open for read */
   };

   struct open_write_delegation4 {
    stateid4 stateid;      /* Stateid for delegation */
    bool     recall;       /* Pre-recalled flag for
                              delegations obtained
                              by reclaim
                              (CLAIM_PREVIOUS) */

    nfs_space_limit4
             space_limit; /* Defines condition that
                             the client must check to
                             determine whether the
                             file needs to be flushed
                             to the server on close.  */

    nfsace4   permissions; /* Defines users who don't
                              need an ACCESS call as
                              part of a delegated
                              open. */
   };

   union open_delegation4
   switch (open_delegation_type4 delegation_type) {
           case OPEN_DELEGATE_NONE:
                   void;
           case OPEN_DELEGATE_READ:
                   open_read_delegation4 read;
           case OPEN_DELEGATE_WRITE:
                   open_write_delegation4 write;
   };

   /*
    * Result flags
    */

   /* Client must confirm open */
   const OPEN4_RESULT_CONFIRM      = 0x00000002;
   /* Type of file locking behavior at the server */
   const OPEN4_RESULT_LOCKTYPE_POSIX = 0x00000004;

   struct OPEN4resok {
    stateid4       stateid;      /* Stateid for open */
    change_info4   cinfo;        /* Directory Change Info */
    uint32_t       rflags;       /* Result flags */
```

```
    bitmap4         attrset;      /* attribute set for create*/
    open_delegation4 delegation; /* Info on any open
                                     delegation */
   };

   union OPEN4res switch (nfsstat4 status) {
    case NFS4_OK:
          /* CURRENT_FH: opened file */
          OPEN4resok      resok4;
    default:
          void;
   };
```

15.18.4. Warning to Client Implementors

OPEN resembles LOOKUP in that it generates a filehandle for the
client to use. Unlike LOOKUP though, OPEN creates server state on
the filehandle. In normal circumstances, the client can only release
this state with a CLOSE operation. CLOSE uses the current filehandle
to determine which file to close. Therefore, the client MUST follow
every OPEN operation with a GETFH operation in the same COMPOUND
procedure. This will supply the client with the filehandle such that
CLOSE can be used appropriately.

Simply waiting for the lease on the file to expire is insufficient
because the server may maintain the state indefinitely as long as
another client does not attempt to make a conflicting access to the
same file.

15.18.5. DESCRIPTION

The OPEN operation creates and/or opens a regular file in a directory
with the provided name. If the file does not exist at the server and
creation is desired, specification of the method of creation is
provided by the openhow parameter. The client has the choice of
three creation methods: UNCHECKED4, GUARDED4, or EXCLUSIVE4.

If the current filehandle is a named attribute directory, OPEN will
then create or open a named attribute file. Note that exclusive
create of a named attribute is not supported. If the createmode is
EXCLUSIVE4 and the current filehandle is a named attribute directory,
the server will return EINVAL.

UNCHECKED4 means that the file should be created if a file of that
name does not exist and encountering an existing regular file of that
name is not an error. For this type of create, createattrs specifies
the initial set of attributes for the file. The set of attributes

Haynes & Noveck   Expires February 17, 2014   [Page 231]

may include any writable attribute valid for regular files. When an
UNCHECKED4 create encounters an existing file, the attributes
specified by createattrs are not used, except that when an size of
zero is specified, the existing file is truncated. If GUARDED4 is
specified, the server checks for the presence of a duplicate object
by name before performing the create. If a duplicate exists, an
error of NFS4ERR_EXIST is returned as the status. If the object does
not exist, the request is performed as described for UNCHECKED4. For
each of these cases (UNCHECKED4 and GUARDED4) where the operation is
successful, the server will return to the client an attribute mask
signifying which attributes were successfully set for the object.

EXCLUSIVE4 specifies that the server is to follow exclusive creation
semantics, using the verifier to ensure exclusive creation of the
target. The server should check for the presence of a duplicate
object by name. If the object does not exist, the server creates the
object and stores the verifier with the object. If the object does
exist and the stored verifier matches the client provided verifier,
the server uses the existing object as the newly created object. If
the stored verifier does not match, then an error of NFS4ERR_EXIST is
returned. No attributes may be provided in this case, since the
server may use an attribute of the target object to store the
verifier. If the server uses an attribute to store the exclusive
create verifier, it will signify which attribute by setting the
appropriate bit in the attribute mask that is returned in the
results.

For the target directory, the server returns change_info4 information
in cinfo. With the atomic field of the change_info4 struct, the
server will indicate if the before and after change attributes were
obtained atomically with respect to the link creation.

Upon successful creation, the current filehandle is replaced by that
of the new object.

The OPEN operation provides for Windows share reservation capability
with the use of the share_access and share_deny fields of the OPEN
arguments. The client specifies at OPEN the required share_access
and share_deny modes. For clients that do not directly support
SHAREs (i.e., UNIX), the expected deny value is DENY_NONE. In the
case that there is a existing SHARE reservation that conflicts with
the OPEN request, the server returns the error NFS4ERR_SHARE_DENIED.
For a complete SHARE request, the client must provide values for the
owner and seqid fields for the OPEN argument. For additional
discussion of SHARE semantics see Section 9.9.

In the case that the client is recovering state from a server
failure, the claim field of the OPEN argument is used to signify that

the request is meant to reclaim state previously held.

The "claim" field of the OPEN argument is used to specify the file to
be opened and the state information which the client claims to
possess. There are four basic claim types which cover the various
situations for an OPEN. They are as follows:

CLAIM_NULL: For the client, this is a new OPEN request and there is
   no previous state associate with the file for the client.

CLAIM_PREVIOUS: The client is claiming basic OPEN state for a file
   that was held previous to a server reboot. Generally used when a
   server is returning persistent filehandles; the client may not
   have the file name to reclaim the OPEN.

CLAIM_DELEGATE_CUR: The client is claiming a delegation for OPEN as
   granted by the server. Generally this is done as part of
   recalling a delegation.

CLAIM_DELEGATE_PREV: The client is claiming a delegation granted to
   a previous client instance. This claim type is for use after a
   SETCLIENTID_CONFIRM and before the corresponding DELEGPURGE in two
   situations: after a client reboot and after a lease expiration
   that resulted in loss of all lock state. The server MAY support
   CLAIM_DELEGATE_PREV. If it does support CLAIM_DELEGATE_PREV,
   SETCLIENTID_CONFIRM MUST NOT remove the client's delegation state,
   and the server MUST support the DELEGPURGE operation.

The following errors apply to use of the CLAIM_DELEGATE_PREV claim
type:

o NFS4ERR_NOTSUPP is returned if the server does not support this
   claim type.

o NFS4ERR_INVAL is returned if the reclaim is done at an
   inappropriate time, e.g., after DELEGPURGE has been done.

o NFS4ERR_BAD_RECLAIM is returned if the other error conditions do
   not apply and the server has no record of the delegation whose
   reclaim is being attempted.

For OPEN requests whose claim type is other than CLAIM_PREVIOUS
(i.e., requests other than those devoted to reclaiming opens after a
server reboot) that reach the server during its grace or lease
expiration period, the server returns an error of NFS4ERR_GRACE.

For any OPEN request, the server may return an open delegation, which
allows further opens and closes to be handled locally on the client

as described in Section 10.4. Note that delegation is up to the
server to decide. The client should never assume that delegation
will or will not be granted in a particular instance. It should
always be prepared for either case. A partial exception is the
reclaim (CLAIM_PREVIOUS) case, in which a delegation type is claimed.
In this case, delegation will always be granted, although the server
may specify an immediate recall in the delegation structure.

The rflags returned by a successful OPEN allow the server to return
information governing how the open file is to be handled.

OPEN4_RESULT_CONFIRM indicates that the client MUST execute an
OPEN_CONFIRM operation before using the open file.
OPEN4_RESULT_LOCKTYPE_POSIX indicates the server's file locking
behavior supports the complete set of Posix locking techniques
[fcntl]. From this the client can choose to manage file locking
state in a way to handle a mis-match of file locking management.

If the component is of zero length, NFS4ERR_INVAL will be returned.
The component is also subject to the normal UTF-8, character support,
and name checks. See Section 12.5 for further discussion.

When an OPEN is done and the specified open-owner already has the
resulting filehandle open, the result is to "OR" together the new
share and deny status together with the existing status. In this
case, only a single CLOSE need be done, even though multiple OPENs
were completed. When such an OPEN is done, checking of share
reservations for the new OPEN proceeds normally, with no exception
for the existing OPEN held by the same owner. In this case, the
stateid returned as an "other" field that matches that of the
previous open while the "seqid" field is incremented to reflect the
change status due to the new open.

If the underlying file system at the server is only accessible in a
read-only mode and the OPEN request has specified ACCESS_WRITE or
ACCESS_BOTH, the server will return NFS4ERR_ROFS to indicate a read-
only file system.

As with the CREATE operation, the server MUST derive the owner, owner
ACE, group, or group ACE if any of the four attributes are required
and supported by the server's file system. For an OPEN with the
EXCLUSIVE4 createmode, the server has no choice, since such OPEN
calls do not include the createattrs field. Conversely, if
createattrs is specified, and includes owner or group (or
corresponding ACEs) that the principal in the RPC call's credentials
does not have authorization to create files for, then the server may
return NFS4ERR_PERM.

In the case of a OPEN which specifies a size of zero (e.g.,
truncation) and the file has named attributes, the named attributes
are left as is. They are not removed.

15.18.6. IMPLEMENTATION

The OPEN operation contains support for EXCLUSIVE4 create. The
mechanism is similar to the support in NFSv3 [RFC1813]. As in NFSv3,
this mechanism provides reliable exclusive creation. Exclusive
create is invoked when the how parameter is EXCLUSIVE4. In this
case, the client provides a verifier that can reasonably be expected
to be unique. A combination of a client identifier, perhaps the
client network address, and a unique number generated by the client,
perhaps the RPC transaction identifier, may be appropriate.

If the object does not exist, the server creates the object and
stores the verifier in stable storage. For file systems that do not
provide a mechanism for the storage of arbitrary file attributes, the
server may use one or more elements of the object meta-data to store
the verifier. The verifier must be stored in stable storage to
prevent erroneous failure on retransmission of the request. It is
assumed that an exclusive create is being performed because exclusive
semantics are critical to the application. Because of the expected
usage, exclusive CREATE does not rely solely on the normally volatile
duplicate request cache for storage of the verifier. The duplicate
request cache in volatile storage does not survive a crash and may
actually flush on a long network partition, opening failure windows.
In the UNIX local file system environment, the expected storage
location for the verifier on creation is the meta-data (time stamps)
of the object. For this reason, an exclusive object create may not
include initial attributes because the server would have nowhere to
store the verifier.

If the server cannot support these exclusive create semantics,
possibly because of the requirement to commit the verifier to stable
storage, it should fail the OPEN request with the error,
NFS4ERR_NOTSUPP.

During an exclusive CREATE request, if the object already exists, the
server reconstructs the object's verifier and compares it with the
verifier in the request. If they match, the server treats the
request as a success. The request is presumed to be a duplicate of
an earlier, successful request for which the reply was lost and that
the server duplicate request cache mechanism did not detect. If the
verifiers do not match, the request is rejected with the status,
NFS4ERR_EXIST.

Once the client has performed a successful exclusive create, it must

issue a SETATTR to set the correct object attributes. Until it does
so, it should not rely upon any of the object attributes, since the
server implementation may need to overload object meta-data to store
the verifier. The subsequent SETATTR must not occur in the same
COMPOUND request as the OPEN. This separation will guarantee that
the exclusive create mechanism will continue to function properly in
the face of retransmission of the request.

Use of the GUARDED4 attribute does not provide exactly-once
semantics. In particular, if a reply is lost and the server does not
detect the retransmission of the request, the operation can fail with
NFS4ERR_EXIST, even though the create was performed successfully.
The client would use this behavior in the case that the application
has not requested an exclusive create but has asked to have the file
truncated when the file is opened. In the case of the client timing
out and retransmitting the create request, the client can use
GUARDED4 to prevent against a sequence like: create, write, create
(retransmitted) from occurring.

For SHARE reservations, the client must specify a value for
share_access that is one of READ, WRITE, or BOTH. For share_deny,
the client must specify one of NONE, READ, WRITE, or BOTH. If the
client fails to do this, the server must return NFS4ERR_INVAL.

Based on the share_access value (READ, WRITE, or BOTH) the client
should check that the requester has the proper access rights to
perform the specified operation. This would generally be the results
of applying the ACL access rules to the file for the current
requester. However, just as with the ACCESS operation, the client
should not attempt to second-guess the server's decisions, as access
rights may change and may be subject to server administrative
controls outside the ACL framework. If the requester is not
authorized to READ or WRITE (depending on the share_access value),
the server must return NFS4ERR_ACCESS. Note that since the NFS
version 4 protocol does not impose any requirement that READs and
WRITEs issued for an open file have the same credentials as the OPEN
itself, the server still must do appropriate access checking on the
READs and WRITEs themselves.

If the component provided to OPEN resolves to something other than a
regular file, an error will be returned to the client. If it is a
directory, NFS4ERR_ISDIR is returned; otherwise, NFS4ERR_SYMLINK is
returned. Note that NFS4ERR_SYMLINK is returned for both symlinks
and for special files of other types; NFS4ERR_INVAL would be
inappropriate, since the arguments provided by the client were
correct, and the client cannot necessarily know at the time it sent
the OPEN that the component would resolve to a non-regular file.

   If the current filehandle is not a directory, the error
   NFS4ERR_NOTDIR will be returned.

   If a COMPOUND contains an OPEN which establishes a
   OPEN_DELEGATE_WRITE delegation, then a subsequent GETATTR inside that
   COMPOUND SHOULD NOT result in a CB_GETATTR to the client. The server
   SHOULD understand the GETATTR to be for the same client ID and avoid
   querying the client, which will not be able to respond. This
   sequence of OPEN, GETATTR SHOULD be understood as an atomic retrieval
   of the initial size and change attribute. Further, the client SHOULD
   NOT construct a COMPOUND which mixes operations for different client
   IDs.

15.19. Operation 19: OPENATTR - Open Named Attribute Directory

15.19.1. SYNOPSIS

      (cfh) createdir -> (cfh)

15.19.2. ARGUMENT

   struct OPENATTR4args {
           /* CURRENT_FH: object */
           bool   createdir;
   };

15.19.3. RESULT

   struct OPENATTR4res {
           /* CURRENT_FH: named attr directory */
           nfsstat4   status;
   };

15.19.4. DESCRIPTION

   The OPENATTR operation is used to obtain the filehandle of the named
   attribute directory associated with the current filehandle. The
   result of the OPENATTR will be a filehandle to an object of type
   NF4ATTRDIR. From this filehandle, READDIR and LOOKUP operations can
   be used to obtain filehandles for the various named attributes
   associated with the original file system object. Filehandles
   returned within the named attribute directory will have a type of
   NF4NAMEDATTR.

   The createdir argument allows the client to signify if a named
   attribute directory should be created as a result of the OPENATTR

operation. Some clients may use the OPENATTR operation with a value
of FALSE for createdir to determine if any named attributes exist for
the object. If none exist, then NFS4ERR_NOENT will be returned. If
createdir has a value of TRUE and no named attribute directory
exists, one is created. The creation of a named attribute directory
assumes that the server has implemented named attribute support in
this fashion and is not required to do so by this definition.

15.19.5. IMPLEMENTATION

If the server does not support named attributes for the current
filehandle, an error of NFS4ERR_NOTSUPP will be returned to the
client.

15.20. Operation 20: OPEN_CONFIRM - Confirm Open

15.20.1. SYNOPSIS

     (cfh), seqid, stateid -> stateid

15.20.2. ARGUMENT

```
struct OPEN_CONFIRM4args {
        /* CURRENT_FH: opened file */
        stateid4        open_stateid;
        seqid4          seqid;
};
```

15.20.3. RESULT

```
struct OPEN_CONFIRM4resok {
        stateid4        open_stateid;
};

union OPEN_CONFIRM4res switch (nfsstat4 status) {
 case NFS4_OK:
        OPEN_CONFIRM4resok      resok4;
 default:
        void;
};
```

15.20.4. DESCRIPTION

This operation is used to confirm the sequence id usage for the first
time that a open-owner is used by a client. The stateid returned
from the OPEN operation is used as the argument for this operation

along with the next sequence id for the open-owner. The sequence id
passed to the OPEN_CONFIRM must be 1 (one) greater than the seqid
passed to the OPEN operation. If the server receives an unexpected
sequence id with respect to the original open, then the server
assumes that the client will not confirm the original OPEN and all
state associated with the original OPEN is released by the server.

On success, the current filehandle retains its value.

15.20.5. IMPLEMENTATION

A given client might generate many open_owner4 data structures for a
given client ID. The client will periodically either dispose of its
open_owner4s or stop using them for indefinite periods of time. The
latter situation is why the NFSv4 protocol does not have an explicit
operation to exit an open_owner4: such an operation is of no use in
that situation. Instead, to avoid unbounded memory use, the server
needs to implement a strategy for disposing of open_owner4s that have
no current open state for any files and have not been used recently.
The time period used to determine when to dispose of open_owner4s is
an implementation choice. The time period should certainly be no
less than the lease time plus any grace period the server wishes to
implement beyond a lease time. The OPEN_CONFIRM operation allows the
server to safely dispose of unused open_owner4 data structures.

In the case that a client issues an OPEN operation and the server no
longer has a record of the open_owner4, the server needs to ensure
that this is a new OPEN and not a replay or retransmission.

Servers MUST NOT require confirmation on OPENs that grant delegations
or are doing reclaim operations. See Section 9.1.10 for details.
The server can easily avoid this by noting whether it has disposed of
one open_owner4 for the given client ID. If the server does not
support delegation, it might simply maintain a single bit that notes
whether any open_owner4 (for any client) has been disposed of.

The server must hold unconfirmed OPEN state until one of three events
occur. First, the client sends an OPEN_CONFIRM request with the
appropriate sequence id and stateid within the lease period. In this
case, the OPEN state on the server goes to confirmed, and the
open_owner4 on the server is fully established.

Second, the client sends another OPEN request with a sequence id that
is incorrect for the open_owner4 (out of sequence). In this case,
the server assumes the second OPEN request is valid and the first one
is a replay. The server cancels the OPEN state of the first OPEN
request, establishes an unconfirmed OPEN state for the second OPEN
request, and responds to the second OPEN request with an indication

that an OPEN_CONFIRM is needed. The process then repeats itself.
While there is a potential for a denial of service attack on the
client, it is mitigated if the client and server require the use of a
security flavor based on Kerberos V5 or some other flavor that uses
cryptography.

What if the server is in the unconfirmed OPEN state for a given
open_owner4, and it receives an operation on the open_owner4 that has
a stateid but the operation is not OPEN, or it is OPEN_CONFIRM but
with the wrong stateid? Then, even if the seqid is correct, the
server returns NFS4ERR_BAD_STATEID, because the server assumes the
operation is a replay: if the server has no established OPEN state,
then there is no way, for example, a LOCK operation could be valid.

Third, neither of the two aforementioned events occur for the
open_owner4 within the lease period. In this case, the OPEN state is
canceled and disposal of the open_owner4 can occur.

15.21. Operation 21: OPEN_DOWNGRADE - Reduce Open File Access

15.21.1. SYNOPSIS

    (cfh), stateid, seqid, access, deny -> stateid

15.21.2. ARGUMENT

```
struct OPEN_DOWNGRADE4args {
        /* CURRENT_FH: opened file */
        stateid4        open_stateid;
        seqid4          seqid;
        uint32_t        share_access;
        uint32_t        share_deny;
};
```

15.21.3. RESULT

```
struct OPEN_DOWNGRADE4resok {
        stateid4        open_stateid;
};

union OPEN_DOWNGRADE4res switch(nfsstat4 status) {
 case NFS4_OK:
        OPEN_DOWNGRADE4resok    resok4;
 default:
        void;
};
```

15.21.4. **DESCRIPTION**

   This operation is used to adjust the share_access and share_deny bits
   for a given open. This is necessary when a given open-owner opens
   the same file multiple times with different share_access and
   share_deny flags. In this situation, a close of one of the opens may
   change the appropriate share_access and share_deny flags to remove
   bits associated with opens no longer in effect.

   The share_access and share_deny bits specified in this operation
   replace the current ones for the specified open file. The
   share_access and share_deny bits specified must be exactly equal to
   the union of the share_access and share_deny bits specified for some
   subset of the OPENs in effect for current open-owner on the current
   file. If that constraint is not respected, the error NFS4ERR_INVAL
   should be returned. Since share_access and share_deny bits are
   subsets of those already granted, it is not possible for this request
   to be denied because of conflicting share reservations.

   As the OPEN_DOWNGRADE may change a file to be not-open-for-write and
   a write byte-range lock might be held, the server may have to reject
   the OPEN_DOWNGRADE with a NFS4ERR_LOCKS_HELD.

   On success, the current filehandle retains its value.

15.22. **Operation 22: PUTFH - Set Current Filehandle**

15.22.1. **SYNOPSIS**

      filehandle -> (cfh)

15.22.2. **ARGUMENT**

   struct PUTFH4args {
           nfs_fh4   object;
   };

15.22.3. **RESULT**

   struct PUTFH4res {
           /* CURRENT_FH: */
           nfsstat4   status;
   };

15.22.4. **DESCRIPTION**

   Replaces the current filehandle with the filehandle provided as an
   argument.

   If the security mechanism used by the requester does not meet the
   requirements of the filehandle provided to this operation, the server
   MUST return NFS4ERR_WRONGSEC.

   See Section 15.2.4.1 for more details on the current filehandle.

15.22.5. **IMPLEMENTATION**

   Commonly used as the first operator in an NFS request to set the
   context for following operations.

15.23. **Operation 23: PUTPUBFH - Set Public Filehandle**

15.23.1. **SYNOPSIS**

   - -> (cfh)

15.23.2. **ARGUMENT**

   void;

15.23.3. **RESULT**

   struct PUTPUBFH4res {
           /* CURRENT_FH: public fh */
           nfsstat4   status;
   };

15.23.4. **DESCRIPTION**

   Replaces the current filehandle with the filehandle that represents
   the public filehandle of the server's name space. This filehandle
   may be different from the "root" filehandle which may be associated
   with some other directory on the server.

   The public filehandle represents the concepts embodied in [RFC2054],
   [RFC2055], [RFC2224]. The intent for NFSv4 is that the public
   filehandle (represented by the PUTPUBFH operation) be used as a
   method of providing WebNFS server compatibility with NFSv2 and NFSv3.

   The public filehandle and the root filehandle (represented by the
   PUTROOTFH operation) should be equivalent. If the public and root

filehandles are not equivalent, then the public filehandle MUST be a
descendant of the root filehandle.

15.23.5. IMPLEMENTATION

Used as the first operator in an NFS request to set the context for
following operations.

With the NFSv2 and 3 public filehandle, the client is able to specify
whether the path name provided in the LOOKUP should be evaluated as
either an absolute path relative to the server's root or relative to
the public filehandle. [RFC2224] contains further discussion of the
functionality. With NFSv4, that type of specification is not
directly available in the LOOKUP operation. The reason for this is
because the component separators needed to specify absolute vs.
relative are not allowed in NFSv4. Therefore, the client is
responsible for constructing its request such that the use of either
PUTROOTFH or PUTPUBFH are used to signify absolute or relative
evaluation of an NFS URL respectively.

Note that there are warnings mentioned in [RFC2224] with respect to
the use of absolute evaluation and the restrictions the server may
place on that evaluation with respect to how much of its namespace
has been made available. These same warnings apply to NFSv4. It is
likely, therefore that because of server implementation details, an
NFSv3 absolute public filehandle lookup may behave differently than
an NFSv4 absolute resolution.

There is a form of security negotiation as described in [RFC2755]
that uses the public filehandle a method of employing Simple and
Protected GSSAPI Negotiation Mechanism (SNEGO) [RFC4178]. This
method is not available with NFSv4 as filehandles are not overloaded
with special meaning and therefore do not provide the same framework
as NFSv2 and NFSv3. Clients should therefore use the security
negotiation mechanisms described in this RFC.

15.24. Operation 24: PUTROOTFH - Set Root Filehandle

15.24.1. SYNOPSIS

   - -> (cfh)

15.24.2. ARGUMENT

   void;

15.24.3. **RESULT**

```
struct PUTROOTFH4res {
        /* CURRENT_FH: root fh */
        nfsstat4        status;
};
```

15.24.4. **DESCRIPTION**

Replaces the current filehandle with the filehandle that represents
the root of the server's name space. From this filehandle a LOOKUP
operation can locate any other filehandle on the server. This
filehandle may be different from the "public" filehandle which may be
associated with some other directory on the server.

See Section 15.2.4.1 for more details on the current filehandle.

15.24.5. **IMPLEMENTATION**

Commonly used as the first operator in an NFS request to set the
context for following operations.

15.25. **Operation 25: READ - Read from File**

15.25.1. **SYNOPSIS**

```
    (cfh), stateid, offset, count -> eof, data
```

15.25.2. **ARGUMENT**

```
struct READ4args {
        /* CURRENT_FH: file */
        stateid4        stateid;
        offset4         offset;
        count4          count;
};
```

[15.25.3](#). **RESULT**

```
struct READ4resok {
        bool            eof;
        opaque          data<>;
};

union READ4res switch (nfsstat4 status) {
 case NFS4_OK:
        READ4resok      resok4;
 default:
        void;
};
```

[15.25.4](#). **DESCRIPTION**

   The READ operation reads data from the regular file identified by the
   current filehandle.

   The client provides an offset of where the READ is to start and a
   count of how many bytes are to be read. An offset of 0 (zero) means
   to read data starting at the beginning of the file. If offset is
   greater than or equal to the size of the file, the status, NFS4_OK,
   is returned with a data length set to 0 (zero) and eof is set to
   TRUE. The READ is subject to access permissions checking.

   If the client specifies a count value of 0 (zero), the READ succeeds
   and returns 0 (zero) bytes of data again subject to access
   permissions checking. The server may choose to return fewer bytes
   than specified by the client. The client needs to check for this
   condition and handle the condition appropriately.

   The stateid value for a READ request represents a value returned from
   a previous byte-range lock or share reservation request or the
   stateid associated with a delegation. The stateid is used by the
   server to verify that the associated share reservation and any byte-
   range locks are still valid and to update lease timeouts for the
   client.

   If the read ended at the end-of-file (formally, in a correctly formed
   READ request, if offset + count is equal to the size of the file), or
   the read request extends beyond the size of the file (if offset +
   count is greater than the size of the file), eof is returned as TRUE;
   otherwise it is FALSE. A successful READ of an empty file will
   always return eof as TRUE.

   If the current filehandle is not a regular file, an error will be

   returned to the client. In the case the current filehandle
   represents a directory, NFS4ERR_ISDIR is returned; otherwise,
   NFS4ERR_INVAL is returned.

   For a READ with a stateid value of all bits 0, the server MAY allow
   the READ to be serviced subject to mandatory file locks or the
   current share deny modes for the file. For a READ with a stateid
   value of all bits 1, the server MAY allow READ operations to bypass
   locking checks at the server.

   On success, the current filehandle retains its value.

15.25.5. IMPLEMENTATION

   If the server returns a "short read" (i.e., fewer data than requested
   and eof is set to FALSE), the client should send another READ to get
   the remaining data. A server may return less data than requested
   under several circumstances. The file may have been truncated by
   another client or perhaps on the server itself, changing the file
   size from what the requesting client believes to be the case. This
   would reduce the actual amount of data available to the client. It
   is possible that the server reduces the transfer size and so returns
   a short read result. Server resource exhaustion may also result in a
   short read.

   If mandatory byte-range locking is in effect for the file, and if the
   byte-range corresponding to the data to be read from the file is
   WRITE_LT locked by an owner not associated with the stateid, the
   server will return the NFS4ERR_LOCKED error. The client should try
   to get the appropriate READ_LT via the LOCK operation before
   reattempting the READ. When the READ completes, the client should
   release the byte-range lock via LOCKU.

   If another client has an OPEN_DELEGATE_WRITE delegation for the file
   being read, the delegation must be recalled, and the operation cannot
   proceed until that delegation is returned or revoked. Except where
   this happens very quickly, one or more NFS4ERR_DELAY errors will be
   returned to requests made while the delegation remains outstanding.
   Normally, delegations will not be recalled as a result of a READ
   operation since the recall will occur as a result of an earlier OPEN.
   However, since it is possible for a READ to be done with a special
   stateid, the server needs to check for this case even though the
   client should have done an OPEN previously.

15.26. Operation 26: READDIR - Read Directory

[15.26.1](#). SYNOPSIS

```
(cfh), cookie, cookieverf, dircount, maxcount, attr_request ->
cookieverf { cookie, name, attrs }
```

[15.26.2](#). ARGUMENT

```
struct READDIR4args {
        /* CURRENT_FH: directory */
        nfs_cookie4     cookie;
        verifier4       cookieverf;
        count4          dircount;
        count4          maxcount;
        bitmap4         attr_request;
};
```

[15.26.3](#). RESULT

```
struct entry4 {
        nfs_cookie4     cookie;
        component4      name;
        fattr4          attrs;
        entry4          *nextentry;
};

struct dirlist4 {
        entry4          *entries;
        bool            eof;
};

struct READDIR4resok {
        verifier4       cookieverf;
        dirlist4        reply;
};


union READDIR4res switch (nfsstat4 status) {
 case NFS4_OK:
        READDIR4resok  resok4;
 default:
        void;
};
```

15.26.4. **DESCRIPTION**

   The READDIR operation retrieves a variable number of entries from a
   file system directory and returns client requested attributes for
   each entry along with information to allow the client to request
   additional directory entries in a subsequent READDIR.

   The arguments contain a cookie value that represents where the
   READDIR should start within the directory. A value of 0 (zero) for
   the cookie is used to start reading at the beginning of the
   directory. For subsequent READDIR requests, the client specifies a
   cookie value that is provided by the server on a previous READDIR
   request.

   The cookieverf value should be set to 0 (zero) when the cookie value
   is 0 (zero) (first directory read). On subsequent requests, it
   should be a cookieverf as returned by the server. The cookieverf
   must match that returned by the READDIR in which the cookie was
   acquired. If the server determines that the cookieverf is no longer
   valid for the directory, the error NFS4ERR_NOT_SAME must be returned.

   The dircount portion of the argument is a hint of the maximum number
   of bytes of directory information that should be returned. This
   value represents the length of the names of the directory entries and
   the cookie value for these entries. This length represents the XDR
   encoding of the data (names and cookies) and not the length in the
   native format of the server.

   The maxcount value of the argument is the maximum number of bytes for
   the result. This maximum size represents all of the data being
   returned within the READDIR4resok structure and includes the XDR
   overhead. The server may return less data. If the server is unable
   to return a single directory entry within the maxcount limit, the
   error NFS4ERR_TOOSMALL will be returned to the client.

   Finally, attr_request represents the list of attributes to be
   returned for each directory entry supplied by the server.

   On successful return, the server's response will provide a list of
   directory entries. Each of these entries contains the name of the
   directory entry, a cookie value for that entry, and the associated
   attributes as requested. The "eof" flag has a value of TRUE if there
   are no more entries in the directory.

   The cookie value is only meaningful to the server and is used as a
   "bookmark" for the directory entry. As mentioned, this cookie is
   used by the client for subsequent READDIR operations so that it may
   continue reading a directory. The cookie is similar in concept to a

READ offset but should not be interpreted as such by the client.
Ideally, the cookie value should not change if the directory is
modified since the client may be caching these values.

In some cases, the server may encounter an error while obtaining the
attributes for a directory entry. Instead of returning an error for
the entire READDIR operation, the server can instead return the
attribute 'fattr4_rdattr_error'. With this, the server is able to
communicate the failure to the client and not fail the entire
operation in the instance of what might be a transient failure.
Obviously, the client must request the fattr4_rdattr_error attribute
for this method to work properly. If the client does not request the
attribute, the server has no choice but to return failure for the
entire READDIR operation.

For some file system environments, the directory entries "." and ".."
have special meaning and in other environments, they may not. If the
server supports these special entries within a directory, they should
not be returned to the client as part of the READDIR response. To
enable some client environments, the cookie values of 0, 1, and 2 are
to be considered reserved. Note that the UNIX client will use these
values when combining the server's response and local representations
to enable a fully formed UNIX directory presentation to the
application.

For READDIR arguments, cookie values of 1 and 2 SHOULD NOT be used
and for READDIR results cookie values of 0, 1, and 2 MUST NOT be
returned.

On success, the current filehandle retains its value.

15.26.5. IMPLEMENTATION

The server's file system directory representations can differ
greatly. A client's programming interfaces may also be bound to the
local operating environment in a way that does not translate well
into the NFS protocol. Therefore the use of the dircount and
maxcount fields are provided to allow the client the ability to
provide guidelines to the server. If the client is aggressive about
attribute collection during a READDIR, the server has an idea of how
to limit the encoded response. The dircount field provides a hint on
the number of entries based solely on the names of the directory
entries. Since it is a hint, it may be possible that a dircount
value is zero. In this case, the server is free to ignore the
dircount value and return directory information based on the
specified maxcount value.

The cookieverf may be used by the server to help manage cookie values

that may become stale. It should be a rare occurrence that a server
is unable to continue properly reading a directory with the provided
cookie/cookieverf pair. The server should make every effort to avoid
this condition since the application at the client may not be able to
properly handle this type of failure.

The use of the cookieverf will also protect the client from using
READDIR cookie values that may be stale. For example, if the file
system has been migrated, the server may or may not be able to use
the same cookie values to service READDIR as the previous server
used. With the client providing the cookieverf, the server is able
to provide the appropriate response to the client. This prevents the
case where the server may accept a cookie value but the underlying
directory has changed and the response is invalid from the client's
context of its previous READDIR.

Since some servers will not be returning "." and ".." entries as has
been done with previous versions of the NFS protocol, the client that
requires these entries be present in READDIR responses must fabricate
them.

15.27. Operation 27: READLINK - Read Symbolic Link

15.27.1. SYNOPSIS

    (cfh) -> linktext

15.27.2. ARGUMENT

    /* CURRENT_FH: symlink */
    void;

15.27.3. RESULT

```
struct READLINK4resok {
        linktext4       link;
};

union READLINK4res switch (nfsstat4 status) {
 case NFS4_OK:
        READLINK4resok resok4;
 default:
        void;
};
```

15.27.4. **DESCRIPTION**

   READLINK reads the data associated with a symbolic link. The data is
   a UTF-8 string that is opaque to the server. That is, whether
   created by an NFS client or created locally on the server, the data
   in a symbolic link is not interpreted when created, but is simply
   stored.

   On success, the current filehandle retains its value.

15.27.5. **IMPLEMENTATION**

   A symbolic link is nominally a pointer to another file. The data is
   not necessarily interpreted by the server, just stored in the file.
   It is possible for a client implementation to store a path name that
   is not meaningful to the server operating system in a symbolic link.
   A READLINK operation returns the data to the client for
   interpretation. If different implementations want to share access to
   symbolic links, then they must agree on the interpretation of the
   data in the symbolic link.

   The READLINK operation is only allowed on objects of type NF4LNK.
   The server should return the error, NFS4ERR_INVAL, if the object is
   not of type, NF4LNK.

15.28. **Operation 28: REMOVE - Remove Filesystem Object**

15.28.1. **SYNOPSIS**

      (cfh), filename -> change_info

15.28.2. **ARGUMENT**

   struct REMOVE4args {
           /* CURRENT_FH: directory */
           component4   target;
   };

[15.28.3](#). **RESULT**

```
struct REMOVE4resok {
        change_info4    cinfo;
};

union REMOVE4res switch (nfsstat4 status) {
 case NFS4_OK:
        REMOVE4resok    resok4;
 default:
        void;
};
```

[15.28.4](#). **DESCRIPTION**

   The REMOVE operation removes (deletes) a directory entry M named by
   filename from the directory corresponding to the current filehandle.
   If the entry in the directory was the last reference to the
   corresponding file system object, the object may be destroyed.

   For the directory where the filename was removed, the server returns
   change_info4 information in cinfo. With the atomic field of the
   change_info4 struct, the server will indicate if the before and after
   change attributes were obtained atomically with respect to the
   removal.

   If the target is of zero length, NFS4ERR_INVAL will be returned. The
   target is also subject to the normal UTF-8, character support, and
   name checks. See [Section 12.5](#) for further discussion.

   On success, the current filehandle retains its value.

[15.28.5](#). **IMPLEMENTATION**

   NFSv3 required a different operator RMDIR for directory removal and
   REMOVE for non-directory removal. This allowed clients to skip
   checking the file type when being passed a non-directory delete
   system call (e.g., unlink() [[unlink](#)] in POSIX) to remove a directory,
   as well as the converse (e.g., a rmdir() on a non-directory) because
   they knew the server would check the file type. NFSv4 REMOVE can be
   used to delete any directory entry independent of its file type. The
   implementor of an NFSv4 client's entry points from the unlink() and
   rmdir() system calls should first check the file type against the
   types the system call is allowed to remove before issuing a REMOVE.
   Alternatively, the implementor can produce a COMPOUND call that
   includes a LOOKUP/VERIFY sequence to verify the file type before a
   REMOVE operation in the same COMPOUND call.

The concept of last reference is server specific. However, if the
numlinks field in the previous attributes of the object had the value
1, the client should not rely on referring to the object via a
filehandle. Likewise, the client should not rely on the resources
(disk space, directory entry, and so on) formerly associated with the
object becoming immediately available. Thus, if a client needs to be
able to continue to access a file after using REMOVE to remove it,
the client should take steps to make sure that the file will still be
accessible. The usual mechanism used is to RENAME the file from its
old name to a new hidden name.

If the server finds that the file is still open when the REMOVE
arrives:

o The server SHOULD NOT delete the file's directory entry if the
   file was opened with OPEN4_SHARE_DENY_WRITE or
   OPEN4_SHARE_DENY_BOTH.

o If the file was not opened with OPEN4_SHARE_DENY_WRITE or
   OPEN4_SHARE_DENY_BOTH, the server SHOULD delete the file's
   directory entry. However, until last CLOSE of the file, the
   server MAY continue to allow access to the file via its
   filehandle.

15.29. Operation 29: RENAME - Rename Directory Entry

15.29.1. SYNOPSIS

   (sfh), oldname, (cfh), newname -> source_cinfo, target_cinfo

15.29.2. ARGUMENT

```
struct RENAME4args {
        /* SAVED_FH: source directory */
        component4      oldname;
        /* CURRENT_FH: target directory */
        component4      newname;
};
```

[15.29.3](#). RESULT

```
struct RENAME4resok {
        change_info4    source_cinfo;
        change_info4    target_cinfo;
};

union RENAME4res switch (nfsstat4 status) {
 case NFS4_OK:
        RENAME4resok    resok4;
 default:
        void;
};
```

[15.29.4](#). DESCRIPTION

The RENAME operation renames the object identified by oldname in the
source directory corresponding to the saved filehandle, as set by the
SAVEFH operation, to newname in the target directory corresponding to
the current filehandle. The operation is required to be atomic to
the client. Source and target directories must reside on the same
file system on the server. On success, the current filehandle will
continue to be the target directory.

If the target directory already contains an entry with the name,
newname, the source object must be compatible with the target: either
both are non-directories or both are directories and the target must
be empty. If compatible, the existing target is removed before the
rename occurs (See [Section 15.28](#) for client and server actions
whenever a target is removed). If they are not compatible or if the
target is a directory but not empty, the server will return the
error, NFS4ERR_EXIST.

If oldname and newname both refer to the same file (they might be
hard links of each other), then RENAME should perform no action and
return success.

For both directories involved in the RENAME, the server returns
change_info4 information. With the atomic field of the change_info4
struct, the server will indicate if the before and after change
attributes were obtained atomically with respect to the rename.

If the oldname refers to a named attribute and the saved and current
filehandles refer to different file system objects, the server will
return NFS4ERR_XDEV just as if the saved and current filehandles
represented directories on different file systems.

If the oldname or newname is of zero length, NFS4ERR_INVAL will be
returned. The oldname and newname are also subject to the normal
UTF-8, character support, and name checks. See Section 12.5 for
further discussion.

15.29.5. IMPLEMENTATION

The RENAME operation must be atomic to the client. The statement
"source and target directories must reside on the same file system on
the server" means that the fsid fields in the attributes for the
directories are the same. If they reside on different file systems,
the error, NFS4ERR_XDEV, is returned.

Based on the value of the fh_expire_type attribute for the object,
the filehandle may or may not expire on a RENAME. However, server
implementors are strongly encouraged to attempt to keep filehandles
from expiring in this fashion.

On some servers, the file names "." and ".." are illegal as either
oldname or newname, and will result in the error NFS4ERR_BADNAME. In
addition, on many servers the case of oldname or newname being an
alias for the source directory will be checked for. Such servers
will return the error NFS4ERR_INVAL in these cases.

If either of the source or target filehandles are not directories,
the server will return NFS4ERR_NOTDIR.

15.30. Operation 30: RENEW - Renew a Lease

15.30.1. SYNOPSIS

```
   clientid -> ()
```

15.30.2. ARGUMENT

```
struct RENEW4args {
        clientid4       clientid;
};
```

15.30.3. RESULT

```
struct RENEW4res {
        nfsstat4        status;
};
```

15.30.4. **DESCRIPTION**

   The RENEW operation is used by the client to renew leases which it
   currently holds at a server. In processing the RENEW request, the
   server renews all leases associated with the client. The associated
   leases are determined by the clientid provided via the SETCLIENTID
   operation.

15.30.5. **IMPLEMENTATION**

   When the client holds delegations, it needs to use RENEW to detect
   when the server has determined that the callback path is down. When
   the server has made such a determination, only the RENEW operation
   will renew the lease on delegations. If the server determines the
   callback path is down, it returns NFS4ERR_CB_PATH_DOWN. Even though
   it returns NFS4ERR_CB_PATH_DOWN, the server MUST renew the lease on
   the byte-range locks and share reservations that the client has
   established on the server. If for some reason the lock and share
   reservation lease cannot be renewed, then the server MUST return an
   error other than NFS4ERR_CB_PATH_DOWN, even if the callback path is
   also down. In the event that the server has conditions such that it
   could return either NFS4ERR_CB_PATH_DOWN or NFS4ERR_LEASE_MOVED,
   NFS4ERR_LEASE_MOVED MUST be handled first.

   The client that issues RENEW MUST choose the principal, RPC security
   flavor, and if applicable, GSS-API mechanism and service via one of
   the following algorithms:

   o The client uses the same principal, RPC security flavor -- and if
      the flavor was RPCSEC_GSS -- the same mechanism and service that
      was used when the client id was established via
      SETCLIENTID_CONFIRM.

   o The client uses any principal, RPC security flavor mechanism and
      service combination that currently has an OPEN file on the server.
      I.e., the same principal had a successful OPEN operation, the file
      is still open by that principal, and the flavor, mechanism, and
      service of RENEW match that of the previous OPEN.

   The server MUST reject a RENEW that does not use one the
   aforementioned algorithms, with the error NFS4ERR_ACCESS.

15.31. **Operation 31: RESTOREFH - Restore Saved Filehandle**

15.31.1. **SYNOPSIS**

      (sfh) -> (cfh)

15.31.2. **ARGUMENT**

```
   /* SAVED_FH: */
   void;
```

15.31.3. **RESULT**

```
   struct RESTOREFH4res {
           /* CURRENT_FH: value of saved fh */
           nfsstat4        status;
   };
```

15.31.4. **DESCRIPTION**

   Set the current filehandle to the value in the saved filehandle. If
   there is no saved filehandle then return the error NFS4ERR_RESTOREFH.

15.31.5. **IMPLEMENTATION**

   Operations like OPEN and LOOKUP use the current filehandle to
   represent a directory and replace it with a new filehandle. Assuming
   the previous filehandle was saved with a SAVEFH operator, the
   previous filehandle can be restored as the current filehandle. This
   is commonly used to obtain post-operation attributes for the
   directory, e.g.,

```
     PUTFH (directory filehandle)
     SAVEFH
     GETATTR attrbits      (pre-op dir attrs)
     CREATE optbits "foo" attrs
     GETATTR attrbits      (file attributes)
     RESTOREFH
     GETATTR attrbits      (post-op dir attrs)
```

15.32. **Operation 32: SAVEFH - Save Current Filehandle**

15.32.1. **SYNOPSIS**

```
   (cfh) -> (sfh)
```

15.32.2. **ARGUMENT**

```
   /* CURRENT_FH: */
   void;
```

[15.32.3](#). **RESULT**

```
struct SAVEFH4res {
        /* SAVED_FH: value of current fh */
        nfsstat4        status;
};
```

[15.32.4](#). **DESCRIPTION**

Save the current filehandle. If a previous filehandle was saved then
it is no longer accessible. The saved filehandle can be restored as
the current filehandle with the RESTOREFH operator.

On success, the current filehandle retains its value.

[15.32.5](#). **IMPLEMENTATION**

[15.33](#). **Operation 33: SECINFO - Obtain Available Security**

[15.33.1](#). **SYNOPSIS**

```
  (cfh), name -> { secinfo }
```

[15.33.2](#). **ARGUMENT**

```
struct SECINFO4args {
        /* CURRENT_FH: directory */
        component4      name;
};
```

15.33.3. RESULT

```
/*
 * From RFC 2203
 */
enum rpc_gss_svc_t {
        RPC_GSS_SVC_NONE        = 1,
        RPC_GSS_SVC_INTEGRITY   = 2,
        RPC_GSS_SVC_PRIVACY     = 3
};

struct rpcsec_gss_info {
        sec_oid4        oid;
        qop4            qop;
        rpc_gss_svc_t   service;
};

/* RPCSEC_GSS has a value of '6' - See RFC 2203 */
union secinfo4 switch (uint32_t flavor) {
 case RPCSEC_GSS:
        rpcsec_gss_info         flavor_info;
 default:
        void;
};

typedef secinfo4 SECINFO4resok<>;

union SECINFO4res switch (nfsstat4 status) {
 case NFS4_OK:
        SECINFO4resok resok4;
 default:
        void;
};
```

15.33.4. DESCRIPTION

The SECINFO operation is used by the client to obtain a list of valid
RPC authentication flavors for a specific directory filehandle, file
name pair. SECINFO should apply the same access methodology used for
LOOKUP when evaluating the name. Therefore, if the requester does
not have the appropriate access to LOOKUP the name then SECINFO must
behave the same way and return NFS4ERR_ACCESS.

The result will contain an array which represents the security
mechanisms available, with an order corresponding to server's
preferences, the most preferred being first in the array. The client
is free to pick whatever security mechanism it both desires and

supports, or to pick in the server's preference order the first one it supports. The array entries are represented by the secinfo4 structure. The field 'flavor' will contain a value of AUTH_NONE, AUTH_SYS (as defined in [RFC5531]), or RPCSEC_GSS (as defined in [RFC2203]).

For the flavors AUTH_NONE and AUTH_SYS, no additional security information is returned. For a return value of RPCSEC_GSS, a security triple is returned that contains the mechanism object id (as defined in [RFC2743]), the quality of protection (as defined in [RFC2743]) and the service type (as defined in [RFC2203]). It is possible for SECINFO to return multiple entries with flavor equal to RPCSEC_GSS with different security triple values.

On success, the current filehandle retains its value.

If the name has a length of 0 (zero), or if name does not obey the UTF-8 definition, the error NFS4ERR_INVAL will be returned.

15.33.5. IMPLEMENTATION

The SECINFO operation is expected to be used by the NFS client when the error value of NFS4ERR_WRONGSEC is returned from another NFS operation. This signifies to the client that the server's security policy is different from what the client is currently using. At this point, the client is expected to obtain a list of possible security flavors and choose what best suits its policies.

As mentioned, the server's security policies will determine when a client request receives NFS4ERR_WRONGSEC. The operations which may receive this error are: LINK, LOOKUP, LOOKUPP, OPEN, PUTFH, PUTPUBFH, PUTROOTFH, RENAME, RESTOREFH, and indirectly READDIR. LINK and RENAME will only receive this error if the security used for the operation is inappropriate for saved filehandle. With the exception of READDIR, these operations represent the point at which the client can instantiate a filehandle into the "current filehandle" at the server. The filehandle is either provided by the client (PUTFH, PUTPUBFH, PUTROOTFH) or generated as a result of a name to filehandle translation (LOOKUP and OPEN). RESTOREFH is different because the filehandle is a result of a previous SAVEFH. Even though the filehandle, for RESTOREFH, might have previously passed the server's inspection for a security match, the server will check it again on RESTOREFH to ensure that the security policy has not changed.

If the client wants to resolve an error return of NFS4ERR_WRONGSEC, the following will occur:

o For LOOKUP and OPEN, the client will use SECINFO with the same
   current filehandle and name as provided in the original LOOKUP or
   OPEN to enumerate the available security triples.

o For LINK, PUTFH, RENAME, and RESTOREFH, the client will use
   SECINFO and provide the parent directory filehandle and object
   name which corresponds to the filehandle originally provided by
   the PUTFH RESTOREFH, or for LINK and RENAME, the SAVEFH.

o For LOOKUPP, PUTROOTFH and PUTPUBFH, the client will be unable to
   use the SECINFO operation since SECINFO requires a current
   filehandle and none exist for these two operations. Therefore,
   the client must iterate through the security triples available at
   the client and reattempt the PUTROOTFH or PUTPUBFH operation. In
   the unfortunate event none of the MANDATORY security triples are
   supported by the client and server, the client SHOULD try using
   others that support integrity. Failing that, the client can try
   using AUTH_NONE, but because such forms lack integrity checks,
   this puts the client at risk. Nonetheless, the server SHOULD
   allow the client to use whatever security form the client requests
   and the server supports, since the risks of doing so are on the
   client.

The READDIR operation will not directly return the NFS4ERR_WRONGSEC
error. However, if the READDIR request included a request for
attributes, it is possible that the READDIR request's security triple
does not match that of a directory entry. If this is the case and
the client has requested the rdattr_error attribute, the server will
return the NFS4ERR_WRONGSEC error in rdattr_error for the entry.

Note that a server MAY use the AUTH_NONE flavor to signify that the
client is allowed to attempt to use authentication flavors that are
not explicitly listed in the SECINFO results. Instead of using a
listed flavor, the client might then, for instance opt to use an
otherwise unlisted RPCSEC_GSS mechanism instead of AUTH_NONE. It may
wish to do so in order to meet an application requirement for data
integrity or privacy. In choosing to use an unlisted flavor, the
client SHOULD always be prepared to handle a failure by falling back
to using AUTH_NONE or another listed flavor. It MUST NOT assume that
identity mapping is supported, and should be prepared for the fact
that its identity is squashed.

See Section 17 for a discussion on the recommendations for security
flavor used by SECINFO.

[15.34](#). **Operation 34: SETATTR - Set Attributes**

[15.34.1](#). **SYNOPSIS**

      (cfh), stateid, attrmask, attr_vals -> attrsset

[15.34.2](#). **ARGUMENT**

    struct SETATTR4args {
            /* CURRENT_FH: target object */
            stateid4   stateid;
            fattr4   obj_attributes;
    };

[15.34.3](#). **RESULT**

    struct SETATTR4res {
            nfsstat4   status;
            bitmap4   attrsset;
    };

[15.34.4](#). **DESCRIPTION**

    The SETATTR operation changes one or more of the attributes of a file
    system object. The new attributes are specified with a bitmap and
    the attributes that follow the bitmap in bit order.

    The stateid argument for SETATTR is used to provide byte-range
    locking context that is necessary for SETATTR requests that set the
    size attribute. Since setting the size attribute modifies the file's
    data, it has the same locking requirements as a corresponding WRITE.
    Any SETATTR that sets the size attribute is incompatible with a share
    reservation that specifies OPEN4_SHARE_DENY_WRITE. The area between
    the old end-of-file and the new end-of-file is considered to be
    modified just as would have been the case had the area in question
    been specified as the target of WRITE, for the purpose of checking
    conflicts with byte-range locks, for those cases in which a server is
    implementing mandatory byte-range locking behavior. A valid stateid
    SHOULD always be specified. When the file size attribute is not set,
    the special stateid consisting of all bits zero MAY be passed.

    On either success or failure of the operation, the server will return
    the attrsset bitmask to represent what (if any) attributes were
    successfully set. The attrsset in the response is a subset of the
    bitmap4 that is part of the obj_attributes in the argument.

On success, the current filehandle retains its value.

15.34.5. IMPLEMENTATION

If the request specifies the owner attribute to be set, the server
SHOULD allow the operation to succeed if the current owner of the
object matches the value specified in the request. Some servers may
be implemented in a way as to prohibit the setting of the owner
attribute unless the requester has privilege to do so. If the server
is lenient in this one case of matching owner values, the client
implementation may be simplified in cases of creation of an object
(e.g., an exclusive create via OPEN) followed by a SETATTR.

The file size attribute is used to request changes to the size of a
file. A value of zero causes the file to be truncated, a value less
than the current size of the file causes data from new size to the
end of the file to be discarded, and a size greater than the current
size of the file causes logically zeroed data bytes to be added to
the end of the file. Servers are free to implement this using holes
or actual zero data bytes. Clients should not make any assumptions
regarding a server's implementation of this feature, beyond that the
bytes returned will be zeroed. Servers MUST support extending the
file size via SETATTR.

SETATTR is not guaranteed atomic. A failed SETATTR may partially
change a file's attributes, hence the reason why the reply always
includes the status and the list of attributes that were set.

If the object whose attributes are being changed has a file
delegation that is held by a client other than the one doing the
SETATTR, the delegation(s) must be recalled, and the operation cannot
proceed to actually change an attribute until each such delegation is
returned or revoked. In all cases in which delegations are recalled,
the server is likely to return one or more NFS4ERR_DELAY errors while
the delegation(s) remains outstanding, although it might not do that
if the delegations are returned quickly.

Changing the size of a file with SETATTR indirectly changes the
time_modify and change attributes. A client must account for this as
size changes can result in data deletion.

The attributes time_access_set and time_modify_set are write-only
attributes constructed as a switched union so the client can direct
the server in setting the time values. If the switched union
specifies SET_TO_CLIENT_TIME4, the client has provided an nfstime4 to
be used for the operation. If the switch union does not specify
SET_TO_CLIENT_TIME4, the server is to use its current time for the
SETATTR operation.

If server and client times differ, programs that compare client time
to file times can break. A time maintenance protocol should be used
to limit client/server time skew.

Use of a COMPOUND containing a VERIFY operation specifying only the
change attribute, immediately followed by a SETATTR, provides a means
whereby a client may specify a request that emulates the
functionality of the SETATTR guard mechanism of NFSv3. Since the
function of the guard mechanism is to avoid changes to the file
attributes based on stale information, delays between checking of the
guard condition and the setting of the attributes have the potential
to compromise this function, as would the corresponding delay in the
NFSv4 emulation. Therefore, NFSv4 servers should take care to avoid
such delays, to the degree possible, when executing such a request.

If the server does not support an attribute as requested by the
client, the server should return NFS4ERR_ATTRNOTSUPP.

A mask of the attributes actually set is returned by SETATTR in all
cases. That mask MUST NOT include attribute bits not requested to be
set by the client. If the attribute masks in the request and reply
are equal, the status field in the reply MUST be NFS4_OK.

[15.35](). Operation 35: SETCLIENTID - Negotiate Client ID

[15.35.1](). SYNOPSIS

   client, callback, callback_ident -> clientid, setclientid_confirm

[15.35.2](). ARGUMENT

```
struct SETCLIENTID4args {
        nfs_client_id4  client;
        cb_client4      callback;
        uint32_t        callback_ident;
};
```

15.35.3. **RESULT**

```
struct SETCLIENTID4resok {
        clientid4       clientid;
        verifier4       setclientid_confirm;
};

union SETCLIENTID4res switch (nfsstat4 status) {
 case NFS4_OK:
        SETCLIENTID4resok      resok4;
 case NFS4ERR_CLID_INUSE:
        clientaddr4    client_using;
 default:
        void;
};
```

15.35.4. **DESCRIPTION**

The client uses the SETCLIENTID operation to notify the server of its
intention to use a particular client identifier, callback, and
callback_ident for subsequent requests that entail creating lock,
share reservation, and delegation state on the server. Upon
successful completion the server will return a shorthand client ID
which, if confirmed via a separate step, will be used in subsequent
file locking and file open requests. Confirmation of the client ID
must be done via the SETCLIENTID_CONFIRM operation to return the
client ID and setclientid_confirm values, as verifiers, to the
server. The reason why two verifiers are necessary is that it is
possible to use SETCLIENTID and SETCLIENTID_CONFIRM to modify the
callback and callback_ident information but not the shorthand client
ID. In that event, the setclientid_confirm value is effectively the
only verifier.

The callback information provided in this operation will be used if
the client is provided an open delegation at a future point.
Therefore, the client must correctly reflect the program and port
numbers for the callback program at the time SETCLIENTID is used.

The callback_ident value is used by the server on the callback. The
client can leverage the callback_ident to eliminate the need for more
than one callback RPC program number, while still being able to
determine which server is initiating the callback.

15.35.5. **IMPLEMENTATION**

To understand how to implement SETCLIENTID, make the following
notations. Let:

x   be the value of the client.id subfield of the SETCLIENTID4args
    structure.

v   be the value of the client.verifier subfield of the
    SETCLIENTID4args structure.

c   be the value of the client ID field returned in the
    SETCLIENTID4resok structure.

k   represent the value combination of the fields callback and
    callback_ident fields of the SETCLIENTID4args structure.

s   be the setclientid_confirm value returned in the SETCLIENTID4resok
    structure.

{ v, x, c, k, s } be a quintuple for a client record. A client
    record is confirmed if there has been a SETCLIENTID_CONFIRM
    operation to confirm it. Otherwise it is unconfirmed. An
    unconfirmed record is established by a SETCLIENTID call.

Since SETCLIENTID is a non-idempotent operation, let us assume that
the server is implementing the duplicate request cache (DRC).

When the server gets a SETCLIENTID { v, x, k } request, it processes
it in the following manner.

o   It first looks up the request in the DRC. If there is a hit, it
    returns the result cached in the DRC. The server does NOT remove
    client state (locks, shares, delegations) nor does it modify any
    recorded callback and callback_ident information for client { x }.

    For any DRC miss, the server takes the client id string x, and
    searches for client records for x that the server may have
    recorded from previous SETCLIENTID calls. For any confirmed
    record with the same id string x, if the recorded principal does
    not match that of SETCLIENTID call, then the server returns a
    NFS4ERR_CLID_INUSE error.

    For brevity of discussion, the remaining description of the
    processing assumes that there was a DRC miss, and that where the
    server has previously recorded a confirmed record for client x,
    the aforementioned principal check has successfully passed.

o   The server checks if it has recorded a confirmed record for { v,
    x, c, l, s }, where l may or may not equal k. If so, and since
    the id verifier v of the request matches that which is confirmed
    and recorded, the server treats this as a probable callback
    information update and records an unconfirmed { v, x, c, k, t }

and leaves the confirmed { v, x, c, l, s } in place, such that t
!= s. It does not matter if k equals l or not. Any pre-existing
unconfirmed { v, x, c, *, * } is removed.

The server returns { c, t }. It is indeed returning the old
clientid4 value c, because the client apparently only wants to
update callback value k to value l. It's possible this request is
one from the Byzantine router that has stale callback information,
but this is not a problem. The callback information update is
only confirmed if followed up by a SETCLIENTID_CONFIRM { c, t }.

The server awaits confirmation of k via SETCLIENTID_CONFIRM { c, t
}.

The server does NOT remove client (lock/share/delegation) state
for x.

o The server has previously recorded a confirmed { u, x, c, l, s }
   record such that v != u, l may or may not equal k, and has not
   recorded any unconfirmed { *, x, *, *, * } record for x. The
   server records an unconfirmed { v, x, d, k, t } (d != c, t != s).

   The server returns { d, t }.

   The server awaits confirmation of { d, k } via SETCLIENTID_CONFIRM
   { d, t }.

   The server does NOT remove client (lock/share/delegation) state
   for x.

o The server has previously recorded a confirmed { u, x, c, l, s }
   record such that v != u, l may or may not equal k, and recorded an
   unconfirmed { w, x, d, m, t } record such that c != d, t != s, m
   may or may not equal k, m may or may not equal l, and k may or may
   not equal l. Whether w == v or w != v makes no difference. The
   server simply removes the unconfirmed { w, x, d, m, t } record and
   replaces it with an unconfirmed { v, x, e, k, r } record, such
   that e != d, e != c, r != t, r != s.

   The server returns { e, r }.

   The server awaits confirmation of { e, k } via SETCLIENTID_CONFIRM
   { e, r }.

   The server does NOT remove client (lock/share/delegation) state
   for x.

   o The server has no confirmed { *, x, *, *, * } for x. It may or
      may not have recorded an unconfirmed { u, x, c, l, s }, where l
      may or may not equal k, and u may or may not equal v. Any
      unconfirmed record { u, x, c, l, * }, regardless whether u == v or
      l == k, is replaced with an unconfirmed record { v, x, d, k, t }
      where d != c, t != s.

      The server returns { d, t }.

      The server awaits confirmation of { d, k } via SETCLIENTID_CONFIRM
      { d, t }. The server does NOT remove client (lock/share/
      delegation) state for x.

   The server generates the clientid and setclientid_confirm values and
   must take care to ensure that these values are extremely unlikely to
   ever be regenerated.

15.36. Operation 36: SETCLIENTID_CONFIRM - Confirm Client ID

15.36.1. SYNOPSIS

      clientid, setclientid_confirm -> -

15.36.2. ARGUMENT

   struct SETCLIENTID_CONFIRM4args {
           clientid4   clientid;
           verifier4   setclientid_confirm;
   };

15.36.3. RESULT

   struct SETCLIENTID_CONFIRM4res {
           nfsstat4   status;
   };

15.36.4. DESCRIPTION

   This operation is used by the client to confirm the results from a
   previous call to SETCLIENTID. The client provides the server
   supplied (from a SETCLIENTID response) client ID. The server
   responds with a simple status of success or failure.

15.36.5. IMPLEMENTATION

   The client must use the SETCLIENTID_CONFIRM operation to confirm the
   following two distinct cases:

   o The client's use of a new shorthand client identifier (as returned
      from the server in the response to SETCLIENTID), a new callback
      value (as specified in the arguments to SETCLIENTID) and a new
      callback_ident (as specified in the arguments to SETCLIENTID)
      value. The client's use of SETCLIENTID_CONFIRM in this case also
      confirms the removal of any of the client's previous relevant
      leased state. Relevant leased client state includes byte-range
      locks, share reservations, and where the server does not support
      the CLAIM_DELEGATE_PREV claim type, delegations. If the server
      supports CLAIM_DELEGATE_PREV, then SETCLIENTID_CONFIRM MUST NOT
      remove delegations for this client; relevant leased client state
      would then just include byte-range locks and share reservations.

   o The client's re-use of an old, previously confirmed, shorthand
      client identifier, a new callback value, and a new callback_ident
      value. The client's use of SETCLIENTID_CONFIRM in this case MUST
      NOT result in the removal of any previous leased state (locks,
      share reservations, and delegations)

   We use the same notation and definitions for v, x, c, k, s, and
   unconfirmed and confirmed client records as introduced in the
   description of the SETCLIENTID operation. The arguments to
   SETCLIENTID_CONFIRM are indicated by the notation { c, s }, where c
   is a value of type clientid4, and s is a value of type verifier4
   corresponding to the setclientid_confirm field.

   As with SETCLIENTID, SETCLIENTID_CONFIRM is a non-idempotent
   operation, and we assume that the server is implementing the
   duplicate request cache (DRC).

   When the server gets a SETCLIENTID_CONFIRM { c, s } request, it
   processes it in the following manner.

   o It first looks up the request in the DRC. If there is a hit, it
      returns the result cached in the DRC. The server does not remove
      any relevant leased client state nor does it modify any recorded
      callback and callback_ident information for client { x } as
      represented by the shorthand value c.

   For a DRC miss, the server checks for client records that match the
   shorthand value c. The processing cases are as follows:

o The server has recorded an unconfirmed { v, x, c, k, s } record
   and a confirmed { v, x, c, l, t } record, such that s != t. If
   the principals of the records do not match that of the
   SETCLIENTID_CONFIRM, the server returns NFS4ERR_CLID_INUSE, and no
   relevant leased client state is removed and no recorded callback
   and callback_ident information for client { x } is changed.
   Otherwise, the confirmed { v, x, c, l, t } record is removed and
   the unconfirmed { v, x, c, k, s } is marked as confirmed, thereby
   modifying recorded and confirmed callback and callback_ident
   information for client { x }.

   The server does not remove any relevant leased client state.

   The server returns NFS4_OK.

o The server has not recorded an unconfirmed { v, x, c, *, * } and
   has recorded a confirmed { v, x, c, *, s }. If the principals of
   the record and of SETCLIENTID_CONFIRM do not match, the server
   returns NFS4ERR_CLID_INUSE without removing any relevant leased
   client state and without changing recorded callback and
   callback_ident values for client { x }.

   If the principals match, then what has likely happened is that the
   client never got the response from the SETCLIENTID_CONFIRM, and
   the DRC entry has been purged. Whatever the scenario, since the
   principals match, as well as { c, s } matching a confirmed record,
   the server leaves client x's relevant leased client state intact,
   leaves its callback and callback_ident values unmodified, and
   returns NFS4_OK.

o The server has not recorded a confirmed { *, *, c, *, * }, and has
   recorded an unconfirmed { *, x, c, k, s }. Even if this is a
   retry from client, nonetheless the client's first
   SETCLIENTID_CONFIRM attempt was not received by the server. Retry
   or not, the server doesn't know, but it processes it as if were a
   first try. If the principal of the unconfirmed { *, x, c, k, s }
   record mismatches that of the SETCLIENTID_CONFIRM request the
   server returns NFS4ERR_CLID_INUSE without removing any relevant
   leased client state.

   Otherwise, the server records a confirmed { *, x, c, k, s }. If
   there is also a confirmed { *, x, d, *, t }, the server MUST
   remove the client x's relevant leased client state, and overwrite
   the callback state with k. The confirmed record { *, x, d, *, t }
   is removed.

   Server returns NFS4_OK.

o The server has no record of a confirmed or unconfirmed { *, *, c,
   *, s }. The server returns NFS4ERR_STALE_CLIENTID. The server
   does not remove any relevant leased client state, nor does it
   modify any recorded callback and callback_ident information for
   any client.

The server needs to cache unconfirmed { v, x, c, k, s } client
records and await for some time their confirmation. As should be
clear from the record processing discussions for SETCLIENTID and
SETCLIENTID_CONFIRM, there are cases where the server does not
deterministically remove unconfirmed client records. To avoid
running out of resources, the server is not required to hold
unconfirmed records indefinitely. One strategy the server might use
is to set a limit on how many unconfirmed client records it will
maintain, and then when the limit would be exceeded, remove the
oldest record. Another strategy might be to remove an unconfirmed
record when some amount of time has elapsed. The choice of the
amount of time is fairly arbitrary but it is surely no higher than
the server's lease time period. Consider that leases need to be
renewed before the lease time expires via an operation from the
client. If the client cannot issue a SETCLIENTID_CONFIRM after a
SETCLIENTID before a period of time equal to that of a lease expires,
then the client is unlikely to be able maintain state on the server
during steady state operation.

If the client does send a SETCLIENTID_CONFIRM for an unconfirmed
record that the server has already deleted, the client will get
NFS4ERR_STALE_CLIENTID back. If so, the client should then start
over, and send SETCLIENTID to reestablish an unconfirmed client
record and get back an unconfirmed client ID and setclientid_confirm
verifier. The client should then send the SETCLIENTID_CONFIRM to
confirm the client ID.

SETCLIENTID_CONFIRM does not establish or renew a lease. However, if
SETCLIENTID_CONFIRM removes relevant leased client state, and that
state does not include existing delegations, the server MUST allow
the client a period of time no less than the value of lease_time
attribute, to reclaim, (via the CLAIM_DELEGATE_PREV claim type of the
OPEN operation) its delegations before removing unreclaimed
delegations.

15.37. Operation 37: VERIFY - Verify Same Attributes

15.37.1. SYNOPSIS

   (cfh), fattr -> -

15.37.2. ARGUMENT

```
struct VERIFY4args {
        /* CURRENT_FH: object */
        fattr4          obj_attributes;
};
```

15.37.3. RESULT

```
struct VERIFY4res {
        nfsstat4        status;
};
```

15.37.4. DESCRIPTION

The VERIFY operation is used to verify that attributes have a value
assumed by the client before proceeding with following operations in
the compound request. If any of the attributes do not match then the
error NFS4ERR_NOT_SAME must be returned. The current filehandle
retains its value after successful completion of the operation.

15.37.5. IMPLEMENTATION

One possible use of the VERIFY operation is the following compound
sequence. With this the client is attempting to verify that the file
being removed will match what the client expects to be removed. This
sequence can help prevent the unintended deletion of a file.

```
  PUTFH (directory filehandle)
  LOOKUP (file name)
  VERIFY (filehandle == fh)
  PUTFH (directory filehandle)
  REMOVE (file name)
```

This sequence does not prevent a second client from removing and
creating a new file in the middle of this sequence but it does help
avoid the unintended result.

In the case that a recommended attribute is specified in the VERIFY
operation and the server does not support that attribute for the file
system object, the error NFS4ERR_ATTRNOTSUPP is returned to the
client.

When the attribute rdattr_error or any write-only attribute (e.g.,
time_modify_set) is specified, the error NFS4ERR_INVAL is returned to
the client.

[15.38](#). **Operation 38: WRITE - Write to File**

[15.38.1](#). **SYNOPSIS**

    (cfh), stateid, offset, stable, data -> count, committed, writeverf

[15.38.2](#). **ARGUMENT**

```
enum stable_how4 {
        UNSTABLE4       = 0,
        DATA_SYNC4      = 1,
        FILE_SYNC4      = 2
};

struct WRITE4args {
        /* CURRENT_FH: file */
        stateid4        stateid;
        offset4         offset;
        stable_how4     stable;
        opaque          data<>;
};
```

[15.38.3](#). **RESULT**

```
struct WRITE4resok {
        count4          count;
        stable_how4     committed;
        verifier4       writeverf;
};

union WRITE4res switch (nfsstat4 status) {
 case NFS4_OK:
        WRITE4resok     resok4;
 default:
        void;
};
```

[15.38.4](#). **DESCRIPTION**

   The WRITE operation is used to write data to a regular file. The
   target file is specified by the current filehandle. The offset
   specifies the offset where the data should be written. An offset of
   0 (zero) specifies that the write should start at the beginning of
   the file. The count, as encoded as part of the opaque data
   parameter, represents the number of bytes of data that are to be
   written. If the count is 0 (zero), the WRITE will succeed and return

a count of 0 (zero) subject to permissions checking. The server may
choose to write fewer bytes than requested by the client.

Part of the write request is a specification of how the write is to
be performed. The client specifies with the stable parameter the
method of how the data is to be processed by the server. If stable
is FILE_SYNC4, the server must commit the data written plus all file
system metadata to stable storage before returning results. This
corresponds to the NFS version 2 protocol semantics. Any other
behavior constitutes a protocol violation. If stable is DATA_SYNC4,
then the server must commit all of the data to stable storage and
enough of the metadata to retrieve the data before returning. The
server implementor is free to implement DATA_SYNC4 in the same
fashion as FILE_SYNC4, but with a possible performance drop. If
stable is UNSTABLE4, the server is free to commit any part of the
data and the metadata to stable storage, including all or none,
before returning a reply to the client. There is no guarantee
whether or when any uncommitted data will subsequently be committed
to stable storage. The only guarantees made by the server are that
it will not destroy any data without changing the value of verf and
that it will not commit the data and metadata at a level less than
that requested by the client.

The stateid value for a WRITE request represents a value returned
from a previous byte-range lock or share reservation request or the
stateid associated with a delegation. The stateid is used by the
server to verify that the associated share reservation and any byte-
range locks are still valid and to update lease timeouts for the
client.

Upon successful completion, the following results are returned. The
count result is the number of bytes of data written to the file. The
server may write fewer bytes than requested. If so, the actual
number of bytes written starting at location, offset, is returned.

The server also returns an indication of the level of commitment of
the data and metadata via committed. If the server committed all
data and metadata to stable storage, committed should be set to
FILE_SYNC4. If the level of commitment was at least as strong as
DATA_SYNC4, then committed should be set to DATA_SYNC4. Otherwise,
committed must be returned as UNSTABLE4. If stable was FILE4_SYNC,
then committed must also be FILE_SYNC4: anything else constitutes a
protocol violation. If stable was DATA_SYNC4, then committed may be
FILE_SYNC4 or DATA_SYNC4: anything else constitutes a protocol
violation. If stable was UNSTABLE4, then committed may be either
FILE_SYNC4, DATA_SYNC4, or UNSTABLE4.

The final portion of the result is the write verifier. The write

verifier is a cookie that the client can use to determine whether the
server has changed instance (boot) state between a call to WRITE and
a subsequent call to either WRITE or COMMIT. This cookie must be
consistent during a single instance of the NFSv4 protocol service and
must be unique between instances of the NFSv4 protocol server, where
uncommitted data may be lost.

If a client writes data to the server with the stable argument set to
UNSTABLE4 and the reply yields a committed response of DATA_SYNC4 or
UNSTABLE4, the client will follow up some time in the future with a
COMMIT operation to synchronize outstanding asynchronous data and
metadata with the server's stable storage, barring client error. It
is possible that due to client crash or other error that a subsequent
COMMIT will not be received by the server.

For a WRITE with a stateid value of all bits 0, the server MAY allow
the WRITE to be serviced subject to mandatory file locks or the
current share deny modes for the file. For a WRITE with a stateid
value of all bits 1, the server MUST NOT allow the WRITE operation to
bypass locking checks at the server and are treated exactly the same
as if a stateid of all bits 0 were used.

On success, the current filehandle retains its value.

15.38.5. IMPLEMENTATION

It is possible for the server to write fewer bytes of data than
requested by the client. In this case, the server should not return
an error unless no data was written at all. If the server writes
less than the number of bytes specified, the client should issue
another WRITE to write the remaining data.

It is assumed that the act of writing data to a file will cause the
time_modified of the file to be updated. However, the time_modified
of the file should not be changed unless the contents of the file are
changed. Thus, a WRITE request with count set to 0 should not cause
the time_modified of the file to be updated.

The definition of stable storage has been historically a point of
contention. The following expected properties of stable storage may
help in resolving design issues in the implementation. Stable
storage is persistent storage that survives:

1. Repeated power failures.

2. Hardware failures (of any board, power supply, etc.).

3. Repeated software crashes, including reboot cycle.

This definition does not address failure of the stable storage module
itself.

The verifier is defined to allow a client to detect different
instances of an NFSv4 protocol server over which cached, uncommitted
data may be lost. In the most likely case, the verifier allows the
client to detect server reboots. This information is required so
that the client can safely determine whether the server could have
lost cached data. If the server fails unexpectedly and the client
has uncommitted data from previous WRITE requests (done with the
stable argument set to UNSTABLE4 and in which the result committed
was returned as UNSTABLE4 as well) it may not have flushed cached
data to stable storage. The burden of recovery is on the client and
the client will need to retransmit the data to the server.

A suggested verifier would be to use the time that the server was
booted or the time the server was last started (if restarting the
server without a reboot results in lost buffers).

The committed field in the results allows the client to do more
effective caching. If the server is committing all WRITE requests to
stable storage, then it should return with committed set to
FILE_SYNC4, regardless of the value of the stable field in the
arguments. A server that uses an NVRAM accelerator may choose to
implement this policy. The client can use this to increase the
effectiveness of the cache by discarding cached data that has already
been committed on the server.

Some implementations may return NFS4ERR_NOSPC instead of
NFS4ERR_DQUOT when a user's quota is exceeded. In the case that the
current filehandle is a directory, the server will return
NFS4ERR_ISDIR. If the current filehandle is not a regular file or a
directory, the server will return NFS4ERR_INVAL.

If mandatory file locking is on for the file, and corresponding
record of the data to be written file is read or write locked by an
owner that is not associated with the stateid, the server will return
NFS4ERR_LOCKED. If so, the client must check if the owner
corresponding to the stateid used with the WRITE operation has a
conflicting read lock that overlaps with the region that was to be
written. If the stateid's owner has no conflicting read lock, then
the client should try to get the appropriate write byte-range lock
via the LOCK operation before re-attempting the WRITE. When the
WRITE completes, the client should release the byte-range lock via
LOCKU.

   If the stateid's owner had a conflicting read lock, then the client
   has no choice but to return an error to the application that
   attempted the WRITE. The reason is that since the stateid's owner
   had a read lock, the server either attempted to temporarily
   effectively upgrade this read lock to a write lock, or the server has
   no upgrade capability. If the server attempted to upgrade the read
   lock and failed, it is pointless for the client to re-attempt the
   upgrade via the LOCK operation, because there might be another client
   also trying to upgrade. If two clients are blocked trying upgrade
   the same lock, the clients deadlock. If the server has no upgrade
   capability, then it is pointless to try a LOCK operation to upgrade.

15.39. Operation 39: RELEASE_LOCKOWNER - Release Lockowner State

15.39.1. SYNOPSIS

      lock-owner -> ()

15.39.2. ARGUMENT

    struct RELEASE_LOCKOWNER4args {
            lock_owner4   lock_owner;
    };

15.39.3. RESULT

    struct RELEASE_LOCKOWNER4res {
            nfsstat4   status;
    };

15.39.4. DESCRIPTION

   This operation is used to notify the server that the lock_owner is no
   longer in use by the client and that future client requests will not
   reference this lock_owner. This allows the server to release cached
   state related to the specified lock_owner. If file locks, associated
   with the lock_owner, are held at the server, the error
   NFS4ERR_LOCKS_HELD will be returned and no further action will be
   taken.

15.39.5. IMPLEMENTATION

   The client may choose to use this operation to ease the amount of
   server state that is held. Information that can be released when a
   RELEASE_LOCKOWNER is done includes the specified lock-owner string,
   the seqid associated with the lock-owner, any saved reply for the

lock-owner, and any lock stateids associated with that lock-owner.

Depending on the behavior of applications at the client, it may be
important for the client to use this operation since the server has
certain obligations with respect to holding a reference to lock-
owner-associated state as long as an associated file is open.
Therefore, if the client knows for certain that the lock_owner will
no longer be used, either to reference existing lock stateids
associated with the lock-owner to create new ones, it should use
RELEASE_LOCKOWNER.

15.40. Operation 10044: ILLEGAL - Illegal operation

15.40.1. SYNOPSIS

```
<null> -> ()
```

15.40.2. ARGUMENT

```
void;
```

15.40.3. RESULT

```
struct ILLEGAL4res {
        nfsstat4        status;
};
```

15.40.4. DESCRIPTION

This operation is a place holder for encoding a result to handle the
case of the client sending an operation code within COMPOUND that is
not supported. See Section 15.2.4 for more details.

The status field of ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

15.40.5. IMPLEMENTATION

A client will probably not send an operation with code OP_ILLEGAL but
if it does, the response will be ILLEGAL4res just as it would be with
any other invalid operation code. Note that if the server gets an
illegal operation code that is not OP_ILLEGAL, and if the server
checks for legal operation codes during the XDR decode phase, then
the ILLEGAL4res would not be returned.

[16](). NFSv4 Callback Procedures

The procedures used for callbacks are defined in the following
sections. In the interest of clarity, the terms "client" and
"server" refer to NFS clients and servers, despite the fact that for
an individual callback RPC, the sense of these terms would be
precisely the opposite.

[16.1](). Procedure 0: CB_NULL - No Operation

[16.1.1](). SYNOPSIS

   <null>

[16.1.2](). ARGUMENT

      void;

[16.1.3](). RESULT

      void;

[16.1.4](). DESCRIPTION

   Standard NULL procedure. Void argument, void response. Even though
   there is no direct functionality associated with this procedure, the
   server will use CB_NULL to confirm the existence of a path for RPCs
   from server to client.

[16.2](). Procedure 1: CB_COMPOUND - Compound Operations

[16.2.1](). SYNOPSIS

   compoundargs -> compoundres

[16.2.2](). ARGUMENT

```
enum nfs_cb_opnum4 {
        OP_CB_GETATTR          = 3,
        OP_CB_RECALL           = 4,
        OP_CB_ILLEGAL          = 10044
};
```

```
   union nfs_cb_argop4 switch (unsigned argop) {
    case OP_CB_GETATTR:
         CB_GETATTR4args            opcbgetattr;
    case OP_CB_RECALL:
         CB_RECALL4args             opcbrecall;
    case OP_CB_ILLEGAL:            void;
   };


   struct CB_COMPOUND4args {
           utf8str_cs     tag;
           uint32_t       minorversion;
           uint32_t       callback_ident;
           nfs_cb_argop4  argarray<>;
   };
```

16.2.3. RESULT

```
   union nfs_cb_resop4 switch (unsigned resop) {
    case OP_CB_GETATTR:    CB_GETATTR4res  opcbgetattr;
    case OP_CB_RECALL:     CB_RECALL4res   opcbrecall;
    case OP_CB_ILLEGAL:    CB_ILLEGAL4res  opcbillegal;
   };


   struct CB_COMPOUND4res {
           nfsstat4       status;
           utf8str_cs     tag;
           nfs_cb_resop4  resarray<>;
   };
```

16.2.4. DESCRIPTION

The CB_COMPOUND procedure is used to combine one or more of the
callback procedures into a single RPC request. The main callback RPC
program has two main procedures: CB_NULL and CB_COMPOUND. All other
operations use the CB_COMPOUND procedure as a wrapper.

In the processing of the CB_COMPOUND procedure, the client may find
that it does not have the available resources to execute any or all
of the operations within the CB_COMPOUND sequence. In this case, the
error NFS4ERR_RESOURCE will be returned for the particular operation
within the CB_COMPOUND procedure where the resource exhaustion
occurred. This assumes that all previous operations within the
CB_COMPOUND sequence have been evaluated successfully.

Contained within the CB_COMPOUND results is a 'status' field. This
status must be equivalent to the status of the last operation that

was executed within the CB_COMPOUND procedure. Therefore, if an
operation incurred an error then the 'status' value will be the same
error value as is being returned for the operation that failed.

For the definition of the "tag" field, see Section 15.2.

The value of callback_ident is supplied by the client during
SETCLIENTID. The server must use the client supplied callback_ident
during the CB_COMPOUND to allow the client to properly identify the
server.

Illegal operation codes are handled in the same way as they are
handled for the COMPOUND procedure.

16.2.5. IMPLEMENTATION

The CB_COMPOUND procedure is used to combine individual operations
into a single RPC request. The client interprets each of the
operations in turn. If an operation is executed by the client and
the status of that operation is NFS4_OK, then the next operation in
the CB_COMPOUND procedure is executed. The client continues this
process until there are no more operations to be executed or one of
the operations has a status value other than NFS4_OK.

16.2.6. Operation 3: CB_GETATTR - Get Attributes

16.2.6.1. SYNOPSIS

       fh, attr_request -> attrmask, attr_vals

16.2.6.2. ARGUMENT

    struct CB_GETATTR4args {
            nfs_fh4 fh;
            bitmap4 attr_request;
    };

16.2.6.3. **RESULT**

```
struct CB_GETATTR4resok {
        fattr4  obj_attributes;
};

union CB_GETATTR4res switch (nfsstat4 status) {
 case NFS4_OK:
        CB_GETATTR4resok        resok4;
 default:
        void;
};
```

16.2.6.4. **DESCRIPTION**

The CB_GETATTR operation is used by the server to obtain the current
modified state of a file that has been OPEN_DELEGATE_WRITE delegated.
The attributes size and change are the only ones guaranteed to be
serviced by the client. See Section 10.4.3 for a full description of
how the client and server are to interact with the use of CB_GETATTR.

If the filehandle specified is not one for which the client holds a
OPEN_DELEGATE_WRITE delegation, an NFS4ERR_BADHANDLE error is
returned.

16.2.6.5. **IMPLEMENTATION**

The client returns attrmask bits and the associated attribute values
only for the change attribute, and attributes that it may change
(time_modify, and size).

16.2.7. **Operation 4: CB_RECALL - Recall an Open Delegation**

16.2.7.1. **SYNOPSIS**

```
   stateid, truncate, fh -> ()
```

16.2.7.2. **ARGUMENT**

```
struct CB_RECALL4args {
        stateid4        stateid;
        bool            truncate;
        nfs_fh4         fh;
};
```

16.2.7.3. **RESULT**

```
struct CB_RECALL4res {
        nfsstat4        status;
};
```

16.2.7.4. **DESCRIPTION**

   The CB_RECALL operation is used to begin the process of recalling an
   open delegation and returning it to the server.

   The truncate flag is used to optimize recall for a file which is
   about to be truncated to zero. When it is set, the client is freed
   of obligation to propagate modified data for the file to the server,
   since this data is irrelevant.

   If the handle specified is not one for which the client holds an open
   delegation, an NFS4ERR_BADHANDLE error is returned.

   If the stateid specified is not one corresponding to an open
   delegation for the file specified by the filehandle, an
   NFS4ERR_BAD_STATEID is returned.

16.2.7.5. **IMPLEMENTATION**

   The client should reply to the callback immediately. Replying does
   not complete the recall except when an error was returned. The
   recall is not complete until the delegation is returned using a
   DELEGRETURN.

16.2.8. **Operation 10044: CB_ILLEGAL - Illegal Callback Operation**

16.2.8.1. **SYNOPSIS**

   <null> -> ()

16.2.8.2. **ARGUMENT**

   void;

16.2.8.3. **RESULT**

```
/*
 * CB_ILLEGAL: Response for illegal operation numbers
 */
struct CB_ILLEGAL4res {
        nfsstat4        status;
};
```

16.2.8.4. **DESCRIPTION**

This operation is a place-holder for encoding a result to handle the
case of the client sending an operation code within COMPOUND that is
not supported. See Section 15.2.4 for more details.

The status field of CB_ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

16.2.8.5. **IMPLEMENTATION**

A server will probably not send an operation with code OP_CB_ILLEGAL
but if it does, the response will be CB_ILLEGAL4res just as it would
be with any other invalid operation code. Note that if the client
gets an illegal operation code that is not OP_ILLEGAL, and if the
client checks for legal operation codes during the XDR decode phase,
then the CB_ILLEGAL4res would not be returned.

17. **Security Considerations**

NFS has historically used a model where, from an authentication
perspective, the client was the entire machine, or at least the
source IP address of the machine. The NFS server relied on the NFS
client to make the proper authentication of the end-user. The NFS
server in turn shared its files only to specific clients, as
identified by the client's source IP address. Given this model, the
AUTH_SYS RPC security flavor simply identified the end-user using the
client to the NFS server. When processing NFS responses, the client
ensured that the responses came from the same IP address and port
number that the request was sent to. While such a model is easy to
implement and simple to deploy and use, it is certainly not a safe
model. Thus, NFSv4 mandates that implementations support a security
model that uses end to end authentication, where an end-user on a
client mutually authenticates (via cryptographic schemes that do not
expose passwords or keys in the clear on the network) to a principal
on an NFS server. Consideration should also be given to the
integrity and privacy of NFS requests and responses. The issues of
end to end mutual authentication, integrity, and privacy are

discussed as part of Section 3.

When an NFSv4 mandated security model is used and a security
principal or an NFSv4 name in user@dns_domain form needs to be
translated to or from a local representation as described in
Section 5.9, the translation SHOULD be done in a secure manner that
preserves the integrity of the translation. For communication with a
name service such as LDAP ([RFC4511]), this means employing a
security service that uses authentication and data integrity.
Kerberos and Transport Layer Security (TLS) ([RFC5246]) are examples
of such a security service.

Note that being REQUIRED to implement does not mean REQUIRED to use;
AUTH_SYS can be used by NFSv4 clients and servers. However, AUTH_SYS
is merely an OPTIONAL security flavor in NFSv4, and so
interoperability via AUTH_SYS is not assured.

For reasons of reduced administration overhead, better performance
and/or reduction of CPU utilization, users of NFSv4 implementations
may choose to not use security mechanisms that enable integrity
protection on each remote procedure call and response. The use of
mechanisms without integrity leaves the customer vulnerable to an
attacker in between the NFS client and server that modifies the RPC
request and/or the response. While implementations are free to
provide the option to use weaker security mechanisms, there are two
operations in particular that warrant the implementation overriding
user choices.

The first such operation is SECINFO. It is recommended that the
client issue the SECINFO call such that it is protected with a
security flavor that has integrity protection, such as RPCSEC_GSS
with a security triple that uses either rpc_gss_svc_integrity or
rpc_gss_svc_privacy (rpc_gss_svc_privacy includes integrity
protection) service. Without integrity protection encapsulating
SECINFO and therefore its results, an attacker in the middle could
modify results such that the client might select a weaker algorithm
in the set allowed by server, making the client and/or server
vulnerable to further attacks.

The second operation that SHOULD use integrity protection is any
GETATTR for the fs_locations attribute. The attack has two steps.
First the attacker modifies the unprotected results of some operation
to return NFS4ERR_MOVED. Second, when the client follows up with a
GETATTR for the fs_locations attribute, the attacker modifies the
results to cause the client migrate its traffic to a server
controlled by the attacker.

Because the operations SETCLIENTID/SETCLIENTID_CONFIRM are

responsible for the release of client state, it is imperative that
the principal used for these operations is checked against and match
the previous use of these operations. See Section 9.1.1 for further
discussion.

18. IANA Considerations

This section uses terms that are defined in [RFC5226].

18.1. Named Attribute Definitions

IANA had created a registry called the "NFSv4 Named Attribute
Definitions Registry" for [RFC3530] and [RFC5661]. This section
introduces no new changes, but it does recap the intent.

The NFSv4 protocol supports the association of a file with zero or
more named attributes. The name space identifiers for these
attributes are defined as string names. The protocol does not define
the specific assignment of the name space for these file attributes.
The IANA registry promotes interoperability where common interests
exist. While application developers are allowed to define and use
attributes as needed, they are encouraged to register the attributes
with IANA.

Such registered named attributes are presumed to apply to all minor
versions of NFSv4, including those defined subsequently to the
registration. Where the named attribute is intended to be limited
with regard to the minor versions for which they are not be used, the
assignment in registry will clearly state the applicable limits.

All assignments to the registry are made on a First Come First Served
basis, per section 4.1 of [RFC5226]. The policy for each assignment
is Specification Required, per section 4.1 of [RFC5226].

Under the NFSv4 specification, the name of a named attribute can in
theory be up to 2^32 - 1 bytes in length, but in practice NFSv4
clients and servers will be unable to handle a string that long.
IANA should reject any assignment request with a named attribute that
exceeds 128 UTF-8 characters. To give IESG the flexibility to set up
bases of assignment of Experimental Use and Standards Action, the
prefixes of "EXPE" and "STDS" are Reserved. The zero length named
attribute name is Reserved.

The prefix "PRIV" is allocated for Private Use. A site that wants to
make use of unregistered named attributes without risk of conflicting
with an assignment in IANA's registry should use the prefix "PRIV" in
all of its named attributes.

Because some NFSv4 clients and servers have case insensitive
semantics, the fifteen additional lower case and mixed case
permutations of each of "EXPE", "PRIV", and "STDS", are Reserved
(e.g. "expe", "expE", "exPe", etc. are Reserved). Similarly, IANA
must not allow two assignments that would conflict if both named
attributes were converted to a common case.

The registry of named attributes is a list of assignments, each
containing three fields for each assignment.

1. A US-ASCII string name that is the actual name of the attribute.
    This name must be unique. This string name can be 1 to 128 UTF-8
    characters long.

2. A reference to the specification of the named attribute. The
    reference can consume up to 256 bytes (or more if IANA permits).

3. The point of contact of the registrant. The point of contact can
    consume up to 256 bytes (or more if IANA permits).

18.1.1. Initial Registry

There is no initial registry.

18.1.2. Updating Registrations

The registrant is always permitted to update the point of contact
field. To make any other change will require Expert Review or IESG
Approval.

19. References

19.1. Normative References

[I-D.ietf-nfsv4-rfc3530bis-dot-x]
           Haynes, T. and D. Noveck, "NFSv4 Version 0 XDR
           Description", draft-ietf-nfsv4-rfc3530bis-dot-x-18 (work
           in progress), Aug 2013.

[ISO.10646-1.1993]
           International Organization for Standardization,
           "Information Technology - Universal Multiple-octet coded
           Character Set (UCS) - Part 1: Architecture and Basic
           Multilingual Plane", ISO Standard 10646-1, May 1993.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
           Requirement Levels", March 1997.

   [RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol
              Specification", RFC 2203, September 1997.

   [RFC2277] Alvestrand, H., "IETF Policy on Character Sets and
              Languages", BCP 18, RFC 2277, January 1998.

   [RFC2743] Linn, J., "Generic Security Service Application Program
              Interface Version 2, Update 1", RFC 2743, January 2000.

   [RFC5403] Eisler, M., "RPCSEC_GSS Version 2", RFC 5403,
              February 2009.

   [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol
              Specification Version 2", RFC 5531, May 2009.

   [RFC5665] Eisler, M., Ed., "IANA Considerations for Remote Procedure
              Call (RPC) Network Identifiers and Universal Address
              Formats", RFC 5665, January 2010.

   [RFC5890] Klensin, J., "Internationalized Domain Names in
              Applications (IDNA): Definitions and Document Framework",
              RFC 5890, August 2010.

   [RFC5891] Klensin, J., "Internationalized Domain Names in
              Applications (IDNA): Protocol", RFC 5891, August 2010.

   [RFC6649] Astrand, L. and T. Yu, "Deprecate DES, RC4-HMAC-EXP, and
              Other Weak Cryptographic Algorithms in Kerberos",
              RFC 6649, July 2012.

   [openg_symlink]
              The Open Group, "Section 3.372 of Chapter 3 of Base
              Definitions of The Open Group Base Specifications Issue 6,
              IEEE Std 1003.1, 2004 Edition, HTML Version
              (www.opengroup.org), ISBN 1931624232", 2004.

19.2. Informative References

   [Chet]   Juszczak, C., "Improving the Performance and Correctness
              of an NFS Server", USENIX Conference Proceedings ,
              June 1990.

   [Floyd]   Floyd, S. and V. Jacobson, "The Synchronization of
              Periodic Routing Messages", IEEE/ACM Transactions on
              Networking 2(2), pp. 122-136, April 1994.

   [ISEG_errata]
              IESG, "IESG Processing of RFC Errata for the IETF Stream",

July 2008.

[P1003.1e]
          Institute of Electrical and Electronics Engineers, Inc.,
          "IEEE Draft P1003.1e", 1997.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7,
          RFC 793, September 1981.

[RFC1094] Nowicki, B., "NFS: Network File System Protocol
          specification", RFC 1094, March 1989.

[RFC1345] Simonsen, K., "Character Mnemonics and Character Sets",
          RFC 1345, June 1992.

[RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS
          Version 3 Protocol Specification", RFC 1813, June 1995.

[RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2",
          RFC 1833, August 1995.

[RFC2054] Callaghan, B., "WebNFS Client Specification", RFC 2054,
          October 1996.

[RFC2055] Callaghan, B., "WebNFS Server Specification", RFC 2055,
          October 1996.

[RFC2224] Callaghan, B., "NFS URL Scheme", RFC 2224, October 1997.

[RFC2623] Eisler, M., "NFS Version 2 and Version 3 Security Issues
          and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5",
          RFC 2623, June 1999.

[RFC2624] Shepler, S., "NFS Version 4 Design Considerations",
          RFC 2624, June 1999.

[RFC2755] Chiu, A., Eisler, M., and B. Callaghan, "Security
          Negotiation for WebNFS", RFC 2755, January 2000.

[RFC3010] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
          Beame, C., Eisler, M., and D. Noveck, "Network File System
          (NFS) version 4 Protocol", RFC 3010, December 2000.

[RFC3232] Reynolds, J., "Assigned Numbers: RFC 1700 is Replaced by
          an On-line Database", RFC 3232, January 2002.

[RFC3454] Hoffman, P. and M. Blanchet, "Preparation of
          Internationalized Strings ("stringprep")", RFC 3454,

December 2002.

[RFC3530]   Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
            Beame, C., Eisler, M., and D. Noveck, "Network File System
            (NFS) version 4 Protocol", RFC 3530, April 2003.

[RFC4121]   Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos
            Version 5 Generic Security Service Application Program
            Interface (GSS-API) Mechanism: Version 2", RFC 4121,
            July 2005.

[RFC4178]   Zhu, L., Leach, P., Jaganathan, K., and W. Ingersoll, "The
            Simple and Protected Generic Security Service Application
            Program Interface (GSS-API) Negotiation Mechanism",
            RFC 4178, October 2005.

[RFC4340]   Kohler, E., Handley, M., and S. Floyd, "Datagram
            Congestion Control Protocol (DCCP)", RFC 4340, March 2006.

[RFC4506]   Eisler, M., "XDR: External Data Representation Standard",
            RFC 4506, May 2006.

[RFC4511]   Sermersheim, J., "Lightweight Directory Access Protocol
            (LDAP): The Protocol", RFC 4511, June 2006.

[RFC5226]   Narten, T. and H. Alvestrand, "Guidelines for Writing an
            IANA Considerations Section in RFCs", BCP 26, RFC 5226,
            May 2008.

[RFC5246]   Dierks, T. and E. Rescorla, "The Transport Layer Security
            (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5661]   Shepler, S., Eisler, M., and D. Noveck, "Network File
            System (NFS) Version 4 Minor Version 1 Protocol",
            RFC 5661, January 2010.

[RFC5740]   Adamson, B., Bormann, C., Handley, M., and J. Macker,
            "Negative-acknowledgment (NACK)-Oriented Reliable
            Multicast (NORM) Transport Protocol", RFC 5740,
            November 2009.

[fcntl]   The Open Group, "Section 'fcntl()' of System Interfaces of
            The Open Group Base Specifications Issue 6 IEEE Std
            1003.1, 2004 Edition, HTML Version (www.opengroup.org),
            ISBN 1931624232", 2004.

[fsync]   The Open Group, "Section 'fsync()' of System Interfaces of
            The Open Group Base Specifications Issue 6 IEEE Std

                    1003.1, 2004 Edition, HTML Version (www.opengroup.org),
                    ISBN 1931624232", 2004.

     [getpwnam]
                    The Open Group, "Section 'getpwnam()' of System Interfaces
                    of The Open Group Base Specifications Issue 6 IEEE Std
                    1003.1, 2004 Edition, HTML Version (www.opengroup.org),
                    ISBN 1931624232", 2004.

     [read_api]
                    The Open Group, "Section 'read()' of System Interfaces of
                    The Open Group Base Specifications Issue 6, IEEE Std
                    1003.1, 2004 Edition", 2004.

     [readdir_api]
                    The Open Group, "Section 'readdir()' of System Interfaces
                    of The Open Group Base Specifications Issue 6, IEEE Std
                    1003.1, 2004 Edition", 2004.

     [unlink]   The Open Group, "Section 'unlink()' of System Interfaces
                    of The Open Group Base Specifications Issue 6 IEEE Std
                    1003.1, 2004 Edition, HTML Version (www.opengroup.org),
                    ISBN 1931624232", 2004.

     [write_api]
                    The Open Group, "Section 'write()' of System Interfaces of
                    The Open Group Base Specifications Issue 6, IEEE Std
                    1003.1, 2004 Edition", 2004.

     [xnfs]   The Open Group, "Protocols for Interworking: XNFS, Version
                    3W, ISBN 1-85912-184-5", February 1998.

Appendix A. Acknowledgments

   A bis is certainly built on the shoulders of the first attempt.
   Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow,
   Carl Beame, Mike Eisler, and David Noveck are responsible for a great
   deal of the effort in this work.

   Rob Thurlow clarified how a client should contact a new server if a
   migration has occurred.

   David Black, Nico Williams, Mike Eisler, Trond Myklebust, and James
   Lentini read many drafts of Section 12 and contributed numerous
   useful suggestions, without which the necessary revision of that
   section for this document would not have been possible.

Peter Staubach read almost all of the drafts of Section 12 leading to the published result and his numerous comments were always useful and contributed substantially to improving the quality of the final result.

James Lentini graciously read the rewrite of Section 8 and his comments were vital in improving the quality of that effort.

Rob Thurlow, Sorin Faibish, James Lentini, Bruce Fields, and Trond Myklebust were faithful attendants of the biweekly triage meeting and accepted many an action item.

Bruce Fields was a good sounding board for both the Third Edge Condition and Courtesy Locks in general. He was also the leading advocate of stamping out backport issues from [RFC5661].

Marcel Telka was a champion of straightening out the difference between a lock-owner and an open-owner. He has also been diligent in reviewing the final document.

Benjamin Kaduk reminded us that DES is dead and Nico Williams helped us close the lid on the coffin.

Appendix B. RFC Editor Notes

[RFC Editor: please remove this section prior to publishing this document as an RFC]

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCNFSv4XDR with RFCxxxx where xxxx is the RFC number assigned to the XDR document.]

[RFC Editor: Please note that there is also a reference entry that needs to be modified for the companion document.]

Authors' Addresses

Thomas Haynes (editor)
NetApp
495 E Java Dr
Sunnyvale, CA 95054
USA

Phone: +1 408 419 3018
Email: thomas@netapp.com

   David Noveck (editor)
   EMC Corporation
   228 South Street
   Hopkinton, MA 01748
   US

   Phone: +1 508 249 5748
   Email: david.noveck@emc.com