Network Working Group
Request for Comments: 3010
Obsoletes: 1813, 1094
Category: Standards Track

S. Shepler
B. Callaghan
D. Robinson
R. Thurlow
Sun Microsystems Inc.
C. Beame
Hummingbird Ltd.
M. Eisler
Zambeel, Inc.
D. Noveck
Network Appliance, Inc.

December 2000

NFS version 4 Protocol

Status of this Memo

Copyright Notice

Abstract

NFS (Network File System) version 4 is a distributed file system protocol which owes heritage to NFS protocol versions 2 [RFC1094] and 3 [RFC1813]. 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.

Key Words

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.

Table of Contents

1. Introduction

The NFS version 4 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 filesystems, simplicity, and good performance. The NFS version 4 revision has the following goals:

- 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.

- Strong security with negotiation built into the protocol.

  The protocol builds on the work of the ONCRPC working group in supporting the RPCSEC_GSS protocol. 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.

- 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.

- Designed for protocol extensions.

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

1.1. Overview of NFS Version 4 Features

To provide a reasonable context for the reader, the major features of NFS version 4 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 that is new to the NFS protocols. For the reader new to the NFS protocols, there is still a fundamental knowledge that is expected. The reader should be familiar with the XDR and RPC protocols as described in [RFC1831] and [RFC1832]. A basic knowledge of file systems and distributed file systems is expected as well.

1.1.1. RPC and Security

As with previous versions of NFS, the External Data Representation (XDR) and Remote Procedure Call (RPC) mechanisms used for the NFS version 4 protocol are those defined in [RFC1831] and [RFC1832]. To meet end to end security requirements, the RPCSEC_GSS framework [RFC2203] 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 [RFC1964] to provide one security framework. The LIPKEY GSS-API mechanism described in

[RFC2847] will be used to provide for the use of user password and server public key by the NFS version 4 protocol. 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 NFS version 4 protocol has added a new operation which provides the client 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.

1.1.2. Procedure and Operation Structure

A significant departure from the previous versions of the NFS protocol is the introduction of the COMPOUND procedure. For the NFS version 4 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 NFS version 4 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.

1.1.3. File System Model

The general file system model used for the NFS version 4 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 NFS version 4 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.

1.1.3.1. 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 NFS version 4 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.

1.1.3.2. Attribute Types

The NFS version 4 protocol introduces three classes of file system or file attributes. Like the additional filehandle type, the classification of file attributes has been done to ease server implementations along with extending the overall functionality of the NFS protocol. This attribute model is structured to be extensible such that new attributes can be introduced in minor revisions of the protocol without requiring significant rework.

The three classifications are: mandatory, recommended and named attributes. This is a significant departure from the previous attribute model used in the NFS protocol. Previously, the attributes for the file system and file objects were a fixed set of mainly Unix attributes. If the server or client did not support a particular attribute, it would have to simulate the attribute the best it could.

Mandatory attributes are the minimal set of file or file system attributes that must be provided by the server and must be properly represented by the server. Recommended attributes represent different file system types and operating environments. The recommended attributes will allow for better interoperability and the inclusion of more operating environments. The mandatory and recommended attribute sets are traditional file or file system attributes. The third type of attribute is the named attribute. 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.

One significant addition to the recommended set of file attributes is the Access Control List (ACL) attribute. This attribute provides for directory and file access control beyond the model used in previous versions of the NFS protocol. The ACL definition allows for specification of user and group level access control.

1.1.3.3. File System Replication and Migration

With the use of a special file attribute, the ability to migrate or replicate server file systems is enabled within the protocol. The file system locations attribute provides a method for the client to probe the server about the location of a file system. In the event of a migration of a file system, the client will receive an error when operating on the file system and it can then query as to the new file system location. Similar steps are used for replication, the client is able to query the server for the multiple available locations of a particular file system. From this information, the client can use its own policies to access the appropriate file system location.

1.1.4. OPEN and CLOSE

The NFS version 4 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.

1.1.5. File locking

With the NFS version 4 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).

1.1.6. Client Caching and Delegation

The file, attribute, and directory caching for the NFS version 4 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 NFS version 4 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 read or write delegation for the file. If the client is granted a 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 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 can not be granted. The essence of a delegation is that it allows the client to locally service operations such as OPEN, CLOSE, LOCK, LOCKU, READ, WRITE without immediate interaction with the server.

1.2. General Definitions

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

Client	The "client" is the entity that accesses the NFS server’s resources. The client may be an application which contains the logic to access the NFS server directly. The client may also be the traditional operating system client remote file system services for a set of applications. In the case of file locking the client is 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 multiple clients may exist on the same network node.
Clientid	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 Clientid.
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 file (share) locks unless specifically stated otherwise.
Server	The "Server" is the entity responsible for coordinating client access to a set of file systems.
Stable Storage	NFS version 4 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 64-bit quantity returned by a server that uniquely defines the locking state granted by the server for a specific lock owner for a specific file. Stateids composed of all bits 0 or all bits 1 have special meaning and are reserved values.
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.

2. Protocol Data Types

The syntax and semantics to describe the data types of the NFS version 4 protocol are defined in the XDR [RFC1832] and RPC [RFC1831] documents. The next sections build upon the XDR data types to define types and structures specific to this protocol.

2.1. Basic 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 definition of change_info
clientid4	`typedef uint64_t clientid4;`	Shorthand reference to client identification
component4	`typedef utf8string component4;`	Represents path name components
count4	`typedef uint32_t count4;`	Various count parameters (READ, WRITE, COMMIT)
length4	`typedef uint64_t length4;`	Describes LOCK lengths
linktext4	`typedef utf8string linktext4;`	Symbolic link contents
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; NFS4_FHSIZE is defined as 128
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)
pathname4	`typedef component4 pathname4<>;`	Represents path name for LOOKUP, OPEN and others
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 contains an ASN.1 OBJECT IDENTIFIER as used
                 by GSS-API in the mech_type argument to
                 GSS_Init_sec_context.  See [RFC2078] for details.

   seqid4        typedef uint32_t        seqid4;
                 Sequence identifier used for file locking

   stateid4      typedef uint64_t        stateid4;
                 State identifier used for file locking and delegation

   utf8string    typedef opaque          utf8string<>;
                 UTF-8 encoding for strings

   verifier4     typedef opaque        verifier4[NFS4_VERIFIER_SIZE];
                 Verifier used for various operations (COMMIT, CREATE,
                 OPEN, READDIR, SETCLIENTID, WRITE)
                 NFS4_VERIFIER_SIZE is defined as 8
```

2.2. Structured Data Types

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.

time_how4

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

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.

specdata4

```
               struct specdata4 {
                       uint32_t specdata1;
                       uint32_t specdata2;
               };
```

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

fsid4

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

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

fs_location4

```
               struct fs_location4 {
                       utf8string    server<>;
                       pathname4     rootpath;
               };
```

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.

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 |
                +-----------+-----------+-----------+--
```

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 the know value of the change attribute for the directory in which the target file system object resides.

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 SETCLIENT operation to either specify the address of the client that is using a clientid or as part of the call back registration.

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.

nfs_client_id4

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

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

nfs_lockowner4

```
               struct nfs_lockowner4 {
                       clientid4     clientid;
                       opaque        owner<>;
               };
```

This structure is used to identify the owner of a OPEN share or file lock.

3. RPC and Security Flavor

The NFS version 4 protocol is a Remote Procedure Call (RPC) application that uses RPC version 2 and the corresponding eXternal Data Representation (XDR) as defined in [RFC1831] and [RFC1832]. The RPCSEC_GSS security flavor as defined in [RFC2203] MUST be used as the mechanism to deliver stronger security for the NFS version 4 protocol.

3.1. Ports and Transports

Historically, NFS version 2 and version 3 servers have resided on port 2049. The registered port 2049 [RFC1700] 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.

The transport used by the RPC service for the NFS version 4 protocol MUST provide congestion control comparable to that defined for TCP in [RFC2581]. If the operating environment implements TCP, the NFS version 4 protocol SHOULD be supported over TCP. The NFS client and server may use other transports if they support congestion control as defined above and in those cases a mechanism may be provided to override TCP usage in favor of another transport.

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 WAN environment by eliminating the need for SYN handshakes.

Note that for various timers, the client and server should avoid inadvertent synchronization of those timers. For further discussion of the general issue refer to [Floyd].

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 [RFC2078]. This allows for the use of varying security mechanisms by the RPC layer without the additional implementation overhead of adding RPC security flavors. For NFS version 4, 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 NFS version 4

The use of RPCSEC_GSS requires selection of: mechanism, quality of protection, and service (authentication, integrity, privacy). The remainder of this document will refer to these three parameters of the RPCSEC_GSS security as the security triple.

3.2.1.1. Kerberos V5 as security triple

The Kerberos V5 GSS-API mechanism as described in [RFC1964] MUST be implemented and provide the following security triples.

column descriptions:

1 == number of pseudo flavor
2 == name of pseudo flavor
3 == mechanism’s OID
4 == mechanism’s algorithm(s)
5 == RPCSEC_GSS service

1	2	3	4	5
390003	krb5	1.2.840.113554.1.2.2	DES MAC MD5	rpc_gss_svc_none
390004	krb5i	1.2.840.113554.1.2.2	DES MAC MD5	rpc_gss_svc_integrity
390005	krb5p	1.2.840.113554.1.2.2	DES MAC MD5 for integrity, and 56 bit DES for privacy.	rpc_gss_svc_privacy

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 NFS version 3 since the security negotiation is done via the MOUNT protocol.

For a discussion of NFS’ use of RPCSEC_GSS and Kerberos V5, please see [RFC2623].

3.2.1.2. LIPKEY as a security triple

The LIPKEY GSS-API mechanism as described in [RFC2847] MUST be implemented and provide the following security triples. The definition of the columns matches the previous subsection "Kerberos V5 as security triple"

1	2	3	4	5
390006	lipkey	1.3.6.1.5.5.9	negotiated	rpc_gss_svc_none
390007	lipkey-i	1.3.6.1.5.5.9	negotiated	rpc_gss_svc_integrity
390008	lipkey-p	1.3.6.1.5.5.9	negotiated	rpc_gss_svc_privacy

The mechanism algorithm is listed as "negotiated". This is because LIPKEY is layered on SPKM-3 and in SPKM-3 [RFC2847] the confidentiality and integrity algorithms are negotiated. Since SPKM-3 specifies HMAC-MD5 for integrity as MANDATORY, 128 bit cast5CBC for confidentiality for privacy as MANDATORY, and further specifies that HMAC-MD5 and cast5CBC MUST be listed first before weaker algorithms, specifying "negotiated" in column 4 does not impair interoperability. In the event an SPKM-3 peer does not support the mandatory algorithms, the other peer is free to accept or reject the GSS-API context creation.

Because SPKM-3 negotiates the algorithms, subsequent calls to LIPKEY’s GSS_Wrap() and GSS_GetMIC() by RPCSEC_GSS will use a quality of protection value of 0 (zero). See section 5.2 of [RFC2025] for an explanation.

LIPKEY uses SPKM-3 to create a secure channel in which to pass a user name and password from the client to the user. Once the user name and password have been accepted by the server, calls to the LIPKEY context are redirected to the SPKM-3 context. See [RFC2847] for more details.

3.2.1.3. SPKM-3 as a security triple

The SPKM-3 GSS-API mechanism as described in [RFC2847] MUST be implemented and provide the following security triples. The definition of the columns matches the previous subsection "Kerberos V5 as security triple".

1	2	3	4	5
390009	spkm3	1.3.6.1.5.5.1.3	negotiated	rpc_gss_svc_none
390010	spkm3i	1.3.6.1.5.5.1.3	negotiated	rpc_gss_svc_integrity
390011	spkm3p	1.3.6.1.5.5.1.3	negotiated	rpc_gss_svc_privacy

For a discussion as to why the mechanism algorithm is listed as "negotiated", see the previous section "LIPKEY as a security triple."

Because SPKM-3 negotiates the algorithms, subsequent calls to SPKM-3’s GSS_Wrap() and GSS_GetMIC() by RPCSEC_GSS will use a quality of protection value of 0 (zero). See section 5.2 of [RFC2025] for an explanation.

Even though LIPKEY is layered over SPKM-3, SPKM-3 is specified as a mandatory set of triples to handle the situations where the initiator (the client) is anonymous or where the initiator has its own certificate. If the initiator is anonymous, there will not be a user name and password to send to the target (the server). If the initiator has its own certificate, then using passwords is superfluous.

3.3. Security Negotiation

With the NFS version 4 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 must 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.

3.3.1. Security Error

Based on the assumption that each NFS version 4 client and server must support a minimum set of security (i.e. LIPKEY, SPKM-3, and Kerberos-V5 all 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.

3.3.2. SECINFO

The new SECINFO operation will allow the client to determine, on a per filehandle basis, what security triple is to be used for server access. In general, the client will not have to use the SECINFO procedure 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.

3.4. Callback RPC Authentication

The callback RPC (described later) must mutually authenticate the NFS server to the principal that acquired the clientid (also described later), using the same security flavor the original SETCLIENTID operation used. Because LIPKEY is layered over SPKM-3, it is permissible for the server to use SPKM-3 and not LIPKEY for the callback even if the client used LIPKEY for SETCLIENTID.

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

For AUTH_SYS, the server simply uses the AUTH_SYS credential that the user used when it 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 and LIPKEY, the following form is RECOMMENDED:

```
nfs/hostname
```

For Kerberos V5, nfs/hostname would be a server principal in the Kerberos Key Distribution Center database. For LIPKEY, this would be the username passed to the target (the NFS version 4 client that receives the callback).

It should be noted that LIPKEY may not work for callbacks, since the LIPKEY client uses a user id/password. If the NFS client receiving the callback can authenticate the NFS server’s user name/password pair, and if the user that the NFS server is authenticating to has a public key certificate, then it works.

In situations where NFS client uses LIPKEY and uses a per-host principal for the SETCLIENTID operation, instead of using LIPKEY for SETCLIENTID, it is RECOMMENDED that SPKM-3 with mutual authentication be used. This effectively means that the client will use a certificate to authenticate and identify the initiator to the target on the NFS server. Using SPKM-3 and not LIPKEY has the following advantages:

- When the server does a callback, it must authenticate to the principal used in the SETCLIENTID. Even if LIPKEY is used, because LIPKEY is layered over SPKM-3, the NFS client will need to have a certificate that corresponds to the principal used in the SETCLIENTID operation. From an administrative perspective, having a user name, password, and certificate for both the client and server is redundant.

- LIPKEY was intended to minimize additional infrastructure requirements beyond a certificate for the target, and the expectation is that existing password infrastructure can be leveraged for the initiator. In some environments, a per-host password does not exist yet. If certificates are used for any per-host principals, then additional password infrastructure is not needed.

- In cases when a host is both an NFS client and server, it can share the same per-host certificate.

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. Since the filehandle is the client’s reference to an object and the client may cache this reference, the server SHOULD not reuse a filehandle for another file system object. If the server needs to reuse a filehandle value, the time elapsed before reuse SHOULD be large enough such that it is unlikely the client has a cached copy of the reused filehandle value. Note that a client may cache a filehandle for a very long time. For example, a client may cache NFS data to local storage as a method to expand its effective cache size and as a means to survive client restarts. Therefore, the lifetime of a cached filehandle may be extended.

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 NFS version 2 protocol [RFC1094] and the NFS version 3 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 procedure in the NFS version 2 and 3 protocols, it has been demonstrated that the MOUNT protocol is unnecessary for viable interaction between NFS client and server.

Therefore, the NFS version 4 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 procedure. A complete discussion of the server name space is in the section "NFS Server Name Space".

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.

4.2. Filehandle Types

In the NFS version 2 and 3 protocols, there was one type of filehandle with a single set of semantics. The NFS version 4 protocol introduces a new type of filehandle in an attempt to accommodate certain server environments. The first type of filehandle is 'persistent'. The semantics of a persistent filehandle are the same as the filehandles of the NFS version 2 and 3 protocols. The second or new type of filehandle is the "volatile" filehandle.

The volatile filehandle type is being 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. However this increased burden is deemed acceptable based on the overall gains achieved by the protocol.

Since the client will need to handle persistent and volatile filehandle 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. If they are not equal, the client may use information provided by the server, in the form of file attributes, to determine whether they denote the same files or different files. The client would do this as necessary for client side caching. Servers SHOULD try to maintain a one-to-one correspondence between filehandles and files 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 the section "Data Caching and File Identity".

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 file handle 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_NOEXPIRE_WITH_OPEN
The filehandle will not expire while client has the file open. If this bit is set, then the values FH4_VOLATILE_ANY or FH4_VOL_RENAME do not impact expiration while the file is open. Once the file is closed or if the FH4_NOEXPIRE_WITH_OPEN bit is false, the rest of the volatile related bits apply.

FH4_VOLATILE_ANY
The filehandle may expire at any time and will expire during system migration and rename.

FH4_VOL_MIGRATION
The filehandle will expire during file system migration. May only be set if FH4_VOLATILE_ANY is not set.

FH4_VOL_RENAME
The filehandle may expire due to a rename. This includes a rename by the requesting client or a rename by another client. May only be set if FH4_VOLATILE_ANY is not set.

Servers which provide volatile filehandles should deny a RENAME or REMOVE that would affect an OPEN file or any of the components leading to the OPEN file. In addition, the server should deny all RENAME or REMOVE requests during the grace or lease period upon server restart.

The reader may be wondering why there are three FH4_VOL* bits and why FH4_VOLATILE_ANY is exclusive of FH4_VOL_MIGRATION and FH4_VOL_RENAME. If the a filehandle is normally persistent but cannot persist across a file set migration, then the presence of the FH4_VOL_MIGRATION or FH4_VOL_RENAME tells the client that it can treat the file handle as persistent for purposes of maintaining a file name to file handle cache, except for the specific event described by the bit. However, FH4_VOLATILE_ANY tells the client that it should not maintain such a cache for unopened files. A server MUST not present FH4_VOLATILE_ANY with FH4_VOL_MIGRATION or

FH4_VOL_RENAME as this will lead to confusion. FH4_VOLATILE_ANY implies that the file handle will expire upon migration or rename, in addition to other events.

4.2.4. One Method of Constructing a Volatile Filehandle

As mentioned, in some instances a filehandle is stale (no longer valid; perhaps because the file was removed from the server) or it is expired (the underlying file is valid but since the filehandle is volatile, it may have expired). Thus the server needs to be able to return NFS4ERR_STALE in the former case and NFS4ERR_FHEXPIRED in the latter case. This can be done by careful construction of the volatile filehandle. One possible implementation follows.

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

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

- slot is an index in the server volatile filehandle table
- generation number is the generation number for the table entry/slot

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
```

5. File Attributes

To meet the requirements of extensibility and increased interoperability with non-Unix platforms, attributes must be handled in a flexible manner. The NFS Version 3 fattr3 structure contains a fixed list of attributes that not all clients and servers are able to support or care about. The fattr3 structure can not be extended as new needs arise and it provides no way to indicate non-support. With the NFS Version 4 protocol, the client will be able to ask what attributes the server supports and will be able to request only those attributes in which it is interested.

To this end, attributes will be divided into three groups: mandatory, recommended, and named. Both mandatory and recommended attributes are supported in the NFS version 4 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 mandatory or recommended attributes may be added to the NFS protocol between major revisions by publishing a standards-track RFC which allocates a new attribute number value and defines the encoding for the attribute. See the section "Minor Versioning" for further discussion.

Named attributes are accessed by the new 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 which are classified as mandatory is deliberately small since servers must do whatever it takes to support them. The recommended attributes may be unsupported; though a server should support as many as it can. Attributes are deemed mandatory 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.

5.1. Mandatory Attributes

These MUST be supported by every NFS Version 4 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 mandatory 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 NFS Version 4 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 which 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. This will not always be comfortable to clients but it seems that the client has a better ability to fabricate or construct an attribute or do without the attribute.

5.3. Named Attributes

These attributes are not supported by direct encoding in the NFS Version 4 protocol but are accessed by string names rather than numbers and correspond to an uninterpreted stream of bytes which 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 "attribute directory" and further perusal of the name space may be done using READDIR and LOOKUP operations on this filehandle. Named attributes may then be examined or changed by normal READ and WRITE and CREATE operations on the filehandles returned from READDIR and LOOKUP. Named attributes may have 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 which is also able to handle them should be able to copy a file's data and meta-data 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.

Names of attributes will not be controlled by this document or other IETF standards track documents. See the section "IANA Considerations" for further discussion.

5.4. Mandatory Attributes - Definitions

Name	#	DataType	Access	Description
supp_attr	0	bitmap	READ	The bit vector which would retrieve all mandatory and recommended attributes that are supported for this object.
type	1	nfs4_ftype	READ	The type of the object (file, directory, symlink)

fh_expire_type	2	uint32	READ	Server uses this to specify filehandle expiration behavior to the client. See the section "Filehandles" for additional description.
change	3	uint64	READ	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_modify attribute for this attribute’s value but only if the file system object can not be updated more frequently than the resolution of time_modify.
size	4	uint64	R/W	The size of the object in bytes.
link_support	5	boolean	READ	Does the object’s file system supports hard links?
symlink_support	6	boolean	READ	Does the object’s file system supports symbolic links?
named_attr	7	boolean	READ	Does this object have named attributes?
fsid	8	fsid4	READ	Unique file system identifier for the file system holding this object. fsid contains major and minor components each of which are uint64.

unique_handles	9	boolean	READ	Are two distinct filehandles guaranteed to refer to two different file system objects?
lease_time	10	nfs_lease4	READ	Duration of leases at server in seconds.
rdattr_error	11	enum	READ	Error returned from getattr during readdir.

5.5. Recommended Attributes - Definitions

Name	#	Data Type	Access	Description
ACL	12	nfsace4<>	R/W	The access control list for the object.
aclsupport	13	uint32	READ	Indicates what types of ACLs are supported on the current file system.
archive	14	boolean	R/W	Whether or not this file has been archived since the time of last modification (deprecated in favor of time_backup).
cansettime	15	boolean	READ	Is the server able to change the times for a file system object as specified in a SETATTR operation?
case_insensitive	16	boolean	READ	Are filename comparisons on this file system case insensitive?
case_preserving	17	boolean	READ	Is filename case on this file system preserved?

chown_restricted	18	boolean	READ	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 NT the "Take Ownership" privilege)
filehandle	19	nfs4_fh	READ	The filehandle of this object (primarily for readdir requests).
fileid	20	uint64	READ	A number uniquely identifying the file within the file system.
files_avail	21	uint64	READ	File slots available to this user on the file system containing this object - this should be the smallest relevant limit.
files_free	22	uint64	READ	Free file slots on the file system containing this object - this should be the smallest relevant limit.
files_total	23	uint64	READ	Total file slots on the file system containing this object.

fs_locations	24	fs_locations	READ	Locations where this file system may be found. If the server returns NFS4ERR_MOVED as an error, this attribute must be supported.
hidden	25	boolean	R/W	Is file considered hidden with respect to the WIN32 API?
homogeneous	26	boolean	READ	Whether or not this object’s file system is homogeneous, i.e. are per file system attributes the same for all file system’s objects.
maxfilesize	27	uint64	READ	Maximum supported file size for the file system of this object.
maxlink	28	uint32	READ	Maximum number of links for this object.
maxname	29	uint32	READ	Maximum filename size supported for this object.
maxread	30	uint64	READ	Maximum read size supported for this object.
maxwrite	31	uint64	READ	Maximum write size supported 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.
mimetype	32	utf8<>	R/W	MIME body type/subtype of this object.
mode	33	mode4	R/W	Unix-style permission bits for this object (deprecated in favor of ACLs)
no_trunc	34	boolean	READ	If a name longer than name_max is used, will an error be returned or will the name be truncated?
numlinks	35	uint32	READ	Number of hard links to this object.
owner	36	utf8<>	R/W	The string name of the owner of this object.
owner_group	37	utf8<>	R/W	The string name of the group ownership of this object.
quota_avail_hard	38	uint64	READ	For definition see "Quota Attributes" section below.
quota_avail_soft	39	uint64	READ	For definition see "Quota Attributes" section below.
quota_used	40	uint64	READ	For definition see "Quota Attributes" section below.
rawdev	41	specdata4	READ	Raw device identifier. Unix device major/minor node information.

space_avail	42	uint64	READ	Disk space in bytes available to this user on the file system containing this object - this should be the smallest relevant limit.
space_free	43	uint64	READ	Free disk space in bytes on the file system containing this object - this should be the smallest relevant limit.
space_total	44	uint64	READ	Total disk space in bytes on the file system containing this object.
space_used	45	uint64	READ	Number of file system bytes allocated to this object.
system	46	boolean	R/W	Is this file a system file with respect to the WIN32 API?
time_access	47	nfstime4	READ	The time of last access to the object.
time_access_set	48	settime4	WRITE	Set the time of last access to the object. SETATTR use only.
time_backup	49	nfstime4	R/W	The time of last backup of the object.
time_create	50	nfstime4	R/W	The time of creation of the object. This attribute does not have any relation to the traditional Unix file attribute "ctime" or "change time".

time_delta	51	nfstime4	READ	Smallest useful server time granularity.
time_metadata	52	nfstime4	R/W	The time of last meta-data modification of the object.
time_modify	53	nfstime4	READ	The time of last modification to the object.
time_modify_set	54	settime4	WRITE	Set the time of last modification to the object. SETATTR use only.

5.6. Interpreting owner and owner_group

The recommended attributes "owner" and "owner_group" 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 [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 that 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.

The translation 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 between a numeric id to the user@dns_domain syntax. A name service may also be used to accomplish the translation. The "dns_domain" portion of the owner string is meant to be a DNS domain name. For example, user@ietf.org.

In the case where there is no translation available to the client or server, the attribute value must be constructed without the "@". Therefore, the absence of the @ from the owner or owner_group attribute signifies that no translation was available and the receiver of the attribute should not place any special meaning with the attribute value. Even though the attribute value can not be translated, it may still be useful. In the case of a client, the attribute string may be used for local display of ownership.

5.7. Character Case Attributes

With respect to the case_insensitive and case_preserving attributes, each UCS-4 character (which UTF-8 encodes) has a "long descriptive name" [RFC1345] which may or may not included 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. For general character handling and internationalization issues, see the section "Internationalization".

5.8. Quota Attributes

For the attributes related to file system quotas, the following definitions apply:

quota_avail_soft
The value in bytes which 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 is a rule as to which other files or directories.

quota_avail_hard
The value in bytes which represent 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.

quota_used
The value in bytes which represent the amount of disc 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 but should do so in a repeatable way. The rule may be configured per-filesystem or may be "choose the set with the smallest quota".

5.9. Access Control Lists

The NFS ACL attribute is an array of access control entries (ACE). There are various access control entry types. The server is able to communicate which ACE types are supported by returning the appropriate value within the aclsupport attribute. The types of ACEs are defined as follows:

Type	Description
ALLOW	Explicitly grants the access defined in acemask4 to the file or directory.
DENY	Explicitly denies the access defined in acemask4 to the file or directory.
AUDIT	LOG (system dependent) any access attempt to a file or directory which uses any of the access methods specified in acemask4.
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 NFS ACE attribute 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;
        utf8string      who;
};
```

To determine if an ACCESS or OPEN request succeeds each nfsace4 entry is processed in order by the server. 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 mode still has unALLOWED bits in common with the "access_mask" of the ACE, the request is denied.

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;
```

5.9.1. ACE type

The semantics of the "type" field follow the descriptions provided above.

The bitmask constants used for the type field 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;
```

5.9.2. ACE flag

The "flag" field contains values based on the following descriptions.

ACE4_FILE_INHERIT_ACE

Can be placed on a directory and indicates that this ACE should be added to each new non-directory file created.

ACE4_DIRECTORY_INHERIT_ACE

Can be placed on a directory and indicates that this ACE should be added to each new directory created.

ACE4_INHERIT_ONLY_ACE

Can be placed on a directory but does not apply to the directory, only to newly created files/directories as specified by the above two flags.

ACE4_NO_PROPAGATE_INHERIT_ACE

Can be placed on a directory. Normally when a new directory is created and an ACE exists on the parent directory which is marked ACL4_DIRECTORY_INHERIT_ACE, two ACEs are placed on the new directory. One for the directory itself and one which is an inheritable ACE for newly created directories. This flag tells the server to not place an ACE on the newly created directory which is inheritable by subdirectories of the created directory.

ACE4_SUCCESSFUL_ACCESS_ACE_FLAG

ACL4_FAILED_ACCESS_ACE_FLAG

Both indicate for AUDIT and ALARM which state to log the event. On every ACCESS or OPEN call which occurs on a file or directory which has an ACL that is of type ACE4_SYSTEM_AUDIT_ACE_TYPE or ACE4_SYSTEM_ALARM_ACE_TYPE, the attempted access is compared to the ace4mask of these ACLs. If the access is a subset of ace4mask and the identifier match, an AUDIT trail or an ALARM is generated. By default this happens regardless of the success or failure of the ACCESS or OPEN call.

The flag ACE4_SUCCESSFUL_ACCESS_ACE_FLAG only produces the AUDIT or ALARM if the ACCESS or OPEN call is successful. The ACE4_FAILED_ACCESS_ACE_FLAG causes the ALARM or AUDIT if the ACCESS or OPEN call fails.

ACE4_IDENTIFIER_GROUP

Indicates that the "who" refers to a GROUP as defined under Unix.

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;
```

5.9.3. ACE Access Mask

The access_mask field contains values based on the following:

Access	Description
READ_DATA	Permission to read the data of the file
LIST_DIRECTORY	Permission to list the contents of a directory
WRITE_DATA	Permission to modify the file's data
ADD_FILE	Permission to add a new file to a directory
APPEND_DATA	Permission to append data to a file
ADD_SUBDIRECTORY	Permission to create a subdirectory to a directory
READ_NAMED_ATTRS	Permission to read the named attributes of a file
WRITE_NAMED_ATTRS	Permission to write the named attributes of a file
EXECUTE	Permission to execute a file
DELETE_CHILD	Permission to delete a file or directory within a directory
READ_ATTRIBUTES	The ability to read basic attributes (non-acls) of a file
WRITE_ATTRIBUTES	Permission to change basic attributes (non-acls) of a file
DELETE	Permission to Delete the file
READ_ACL	Permission to Read the ACL
WRITE_ACL	Permission to Write the ACL
WRITE_OWNER	Permission to change the owner
SYNCHRONIZE	Permission to access file locally at the server with synchronous reads and writes

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;
```

5.9.4. ACE who

There are several special identifiers ("who") which need to be understood universally. 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.

Who	Description
"OWNER"	The owner of the file.
"GROUP"	The group associated with the file.
"EVERYONE"	The world.
"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.

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

6. File System Migration and Replication

With the use of the recommended attribute "fs_locations", the NFS version 4 server has a method of providing file system migration or replication services. For the purposes of migration and replication, a file system will be defined as all files that share a given fsid (both major and minor values are the same).

The fs_locations attribute provides a list of file system locations. These locations are specified by providing the server name (either DNS domain or IP address) and the path name representing the root of the file system. Depending on the type of service being provided, the list will provide a new location or a set of alternate locations for the file system. The client will use this information to redirect its requests to the new server.

6.1. Replication

It is expected that file system replication will be used in the case of read-only data. Typically, the file system will be replicated on two or more servers. The fs_locations attribute will provide the list of these locations to the client. On first access of the file system, the client should obtain the value of the fs_locations attribute. If, in the future, the client finds the server unresponsive, the client may attempt to use another server specified by fs_locations.

If applicable, the client must take the appropriate steps to recover valid filehandles from the new server. This is described in more detail in the following sections.

6.2. Migration

File system migration is used to move a file system from one server to another. Migration is typically used for a file system that is writable and has a single copy. The expected use of migration is for load balancing or general resource reallocation. The protocol does not specify how the file system will be moved between servers. This server-to-server transfer mechanism is left to the server implementor. However, the method used to communicate the migration event between client and server is specified here.

Once the servers participating in the migration have completed the move of the file system, the error NFS4ERR_MOVED will be returned for subsequent requests received by the original server. The NFS4ERR_MOVED error is returned for all operations except GETATTR. Upon receiving the NFS4ERR_MOVED error, the client will obtain the value of the fs_locations attribute. The client will then use the contents of the attribute to redirect its requests to the specified server. To facilitate the use of GETATTR, operations such as PUTFH must also be accepted by the server for the migrated file system’s filehandles. Note that if the server returns NFS4ERR_MOVED, the server MUST support the fs_locations attribute.

If the client requests more attributes than just fs_locations, the server may return fs_locations only. This is to be expected since the server has migrated the file system and may not have a method of obtaining additional attribute data.

The server implementor needs to be careful in developing a migration solution. The server must consider all of the state information clients may have outstanding at the server. This includes but is not limited to locking/share state, delegation state, and asynchronous file writes which are represented by WRITE and COMMIT verifiers. The server should strive to minimize the impact on its clients during and after the migration process.

6.3. Interpretation of the fs_locations Attribute

The fs_location attribute is structured in the following way:

```
struct fs_location {
        utf8string      server<>;
        pathname4       rootpath;
};

struct fs_locations {
        pathname4       fs_root;
        fs_location     locations<>;
};
```

The fs_location struct 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. For a multi-homed server or a set of servers that use the same rootpath, an array of server names may be provided. An entry in the server array is an UTF8 string and represents one of a traditional DNS host name, IPv4 address, or IPv6 address. It is not a requirement that all servers that share the same rootpath be listed in one fs_location struct. The array of server names is provided for convenience. Servers that share the same rootpath may also be listed in separate fs_location entries in the fs_locations attribute.

The fs_locations struct and attribute then contains an array of locations. Since the name space 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 server's name space. Therefore, the fs_root path is only associated with the server from which the fs_locations attribute was obtained. The fs_root path is meant to aid the client in locating the file system at the various servers listed.

As an example, 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". In this example the client accesses the file system first at servA with a multi-component lookup path of "/a/b/c/d". Since the client used a multi-component lookup to obtain the filehandle at "/a/b/c/d", it is unaware that the file system's root is located in servA's name space 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 a fs_root value of "/a/b/c" and two entries in fs_location. One entry in fs_location 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.

6.4. Filehandle Recovery for Migration or Replication

Filehandles for file systems that are replicated or migrated generally have the same semantics as for file systems that are not replicated or migrated. For example, if a file system has persistent filehandles and it is migrated to another server, the filehandle values for the file system will be valid at the new server.

For volatile filehandles, the servers involved likely do not have a mechanism to transfer filehandle format and content between themselves. Therefore, a server may have difficulty in determining if a volatile filehandle from an old server should return an error of NFS4ERR_FHEXPIRED. Therefore, the client is informed, with the use of the fh_expire_type attribute, whether volatile filehandles will expire at the migration or replication event. If the bit FH4_VOL_MIGRATION is set in the fh_expire_type attribute, the client must treat the volatile filehandle as if the server had returned the NFS4ERR_FHEXPIRED error. At the migration or replication event in the presence of the FH4_VOL_MIGRATION bit, the client will not present the original or old volatile file handle to the new server. The client will start its communication with the new server by recovering its filehandles using the saved file names.

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 NFS version 4 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 NFS version 2 and 3 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 File System

NFS version 4 servers avoid this name space inconsistency by presenting all the exports within the framework of a single server name space. An NFS version 4 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 consider 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". File systems are commonly represented as disk letters. MacOS represents file systems as top level names. NFS version 4 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 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.

8. File Locking and Share Reservations

Integrating locking into the NFS protocol necessarily causes it to be state-full. With the inclusion of "share" file locks the protocol becomes substantially more dependent on state than the traditional combination of NFS and NLM [XNFS]. There are three components to making this state manageable:

- Clear division between client and server

- Ability to reliably detect inconsistency in state between client and server

- Simple and robust recovery mechanisms

In this model, the server owns the state information. The client communicates its view of this state to the server as needed. The client is also able to detect inconsistent state before modifying a file.

To support Win32 "share" locks it is necessary to atomically OPEN or CREATE files. 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 NFS version 4 protocol has an OPEN operation that subsumes the functionality 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.

8.1. Locking

It is assumed that manipulating a lock is rare when compared to READ and WRITE operations. It is also assumed that crashes 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 lock request contains the heavyweight information required to establish a lock and uniquely define the lock owner.

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.

8.1.1. 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. Client identification is accomplished with two values.

- A verifier that is used to detect client reboots.

- A variable length opaque array to uniquely define a client.

  For an operating system this may be a fully qualified host name or IP address. For a user level NFS client it may additionally contain a process id or other unique sequence.

The data structure for the Client ID would then appear as:

```
        struct nfs_client_id {
                opaque verifier[4];
                opaque id<>;
        }
```

It is possible through the mis-configuration of a client or the existence of a rogue client that two clients end up using the same nfs_client_id. This situation is avoided by "negotiating" the nfs_client_id between client and server with the use of the SETCLIENTID and SETCLIENTID_CONFIRM operations. The following describes the two scenarios of negotiation.

1 Client has never connected to the server

In this case the client generates an nfs_client_id and unless another client has the same nfs_client_id.id field, the server accepts the request. The server also records the principal (or principal to uid mapping) from the credential in the RPC request that contains the nfs_client_id negotiation request (SETCLIENTID operation).

Two clients might still use the same nfs_client_id.id due to perhaps configuration error. For example, a High Availability configuration where the nfs_client_id.id is derived from the ethernet controller address and both systems have the same address. In this case, the result is a switched union that returns, in addition to NFS4ERR_CLID_INUSE, the network address (the rpcbind netid and universal address) of the client that is using the id.

2 Client is re-connecting to the server after a client reboot

In this case, the client still generates an nfs_client_id but the nfs_client_id.id field will be the same as the nfs_client_id.id generated prior to reboot. If the server finds that the principal/uid is equal to the previously "registered" nfs_client_id.id, then locks associated with the old nfs_client_id are immediately released. If the principal/uid is not equal, then this is a rogue client and the request is returned in error. For more discussion of crash recovery semantics, see the section on "Crash Recovery".

It is possible for a retransmission of request to be received by the server after the server has acted upon and responded to the original client request. Therefore to mitigate effects of the retransmission of the SETCLIENTID operation, the client and server use a confirmation step. The server returns a confirmation verifier that the client then sends to the server in the SETCLIENTID_CONFIRM operation. Once the server receives the confirmation from the client, the locking state for the client is released.

In both cases, upon success, NFS4_OK is returned. To help reduce the amount of data transferred on OPEN and LOCK, the server will also return a unique 64-bit clientid value that is a shorthand reference to the nfs_client_id values presented by the client. From this point forward, the client will use the clientid to refer to itself.

The clientid assigned by the server should be chosen so that it will not conflict with a clientid previously assigned by the server. This applies across server restarts or reboots. When a clientid is presented to a server and that clientid 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 clientid by use of the SETCLIENTID operation and then proceed to any other necessary recovery for the server reboot case (See the section "Server Failure and Recovery").

The client must also employ the SETCLIENTID operation when it receives a NFS4ERR_STALE_STATEID error using a stateid derived from its current clientid, since this also indicates a server reboot which has invalidated the existing clientid (see the next section "nfs_lockowner and stateid Definition" for details).

8.1.2. Server Release of Clientid

If the server determines that the client holds no associated state for its clientid, the server may choose to release the clientid. 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 clientid 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 clientid unless there had been no activity from that client for many minutes.

8.1.3. nfs_lockowner and stateid Definition

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

- A clientid returned by the server as part of the client's use of the SETCLIENTID operation.

- 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 64-bit stateid. The stateid is used as a shorthand reference to the nfs_lockowner, since the server will be maintaining the correspondence between them.

The server is free to form the stateid in any manner that it chooses as long as it is able to recognize invalid and out-of-date stateids. This requirement includes those stateids generated by earlier instances of the server. From this, the client can be properly notified of a server restart. This notification will occur when the client presents a stateid to the server from a previous instantiation.

The server must be able to distinguish the following situations and return the error as specified:

- The stateid was generated by an earlier server instance (i.e. before a server reboot). The error NFS4ERR_STALE_STATEID should be returned.

- The stateid was generated by the current server instance but the stateid no longer designates the current locking state for the lockowner-file pair in question (i.e. one or more locking operations has occurred). The error NFS4ERR_OLD_STATEID should be returned.

  This error condition will only occur when the client issues a locking request which changes a stateid while an I/O request that uses that stateid is outstanding.

- The stateid was generated by the current server instance but the stateid does not designate a locking state for any active lockowner-file pair. The error NFS4ERR_BAD_STATEID should be returned.

  This error condition will occur when there has been a logic error on the part of the client or server. This should not happen.

One mechanism that may be used to satisfy these requirements is for the server to divide stateids into three fields:

- A server verifier which uniquely designates a particular server instantiation.

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

- A sequence value which is incremented for each stateid that is associated with the same index into the locking-state table.

By matching the incoming stateid and its field values with the state held at the server, the server is able to easily determine if a stateid is valid for its current instantiation and state. If the stateid is not valid, the appropriate error can be supplied to the client.

8.1.4. Use of the stateid

All READ and WRITE operations contain a stateid. If the nfs_lockowner performs a READ or WRITE on a range of bytes within a locked range, the stateid (previously returned by the server) must be used to indicate that the appropriate lock (record or share) is held. If no state is established by the client, either record lock or share lock, a stateid of all bits 0 is used. If no conflicting locks are held on the file, the server may service the READ or WRITE operation. If a conflict with an explicit lock occurs, an error is returned for the operation (NFS4ERR_LOCKED). This allows "mandatory locking" to be implemented.

A stateid of all bits 1 (one) allows READ operations to bypass record locking checks at the server. However, WRITE operations with stateid with bits all 1 (one) do not bypass record locking checks. File locking checks are handled by the OPEN operation (see the section "OPEN/CLOSE Operations").

An explicit lock may not be granted while a READ or WRITE operation with conflicting implicit locking is being performed.

8.1.5. Sequencing of Lock Requests

Locking is different than most NFS operations as it requires "at-most-one" semantics that are not provided by ONCRPC. ONCRPC 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 nfs_lockowners have different sequences. The server maintains the last sequence number (L) received and the response that was returned.

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

If a request 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 client verifier changes.

Since the sequence number is represented with an unsigned 32-bit integer, the arithmetic involved with the sequence number is mod 2^32.

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 [Juszczak]. 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 nfs_lockowner must be cached as long as the lock state exists on the server.

8.1.6. Recovery from Replayed Requests

As described above, the sequence number is per nfs_lockowner. 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 nfs_lockowner, 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 nfs_lockowner state.

8.1.7. Releasing nfs_lockowner State

When a particular nfs_lockowner no longer holds open or file locking state at the server, the server may choose to release the sequence number state associated with the nfs_lockowner. The server may make this choice based on lease expiration, for the reclamation of server memory, or other implementation specific details. In any event, the server is able to do this safely only when the nfs_lockowner no longer is being utilized by the client. The server may choose to hold the nfs_lockowner state in the event that retransmitted requests are received. However, the period 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 nfs_lockowner state, the server will find that the nfs_lockowner has no files open and an error will be returned to the client. If the nfs_lockowner does have a file open, the stateid will not match and again an error is returned to the client.

In the case that an OPEN is retransmitted and the nfs_lockowner is being used for the first time or the nfs_lockowner 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 nfs_lockowner for the first time, it will direct the client to perform the OPEN_CONFIRM for the corresponding OPEN. This sequence establishes the use of an nfs_lockowner and associated sequence number. See the section "OPEN_CONFIRM - Confirm Open" for further details.

8.2. Lock Ranges

The protocol allows a lock owner to request a lock with one 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 "Server Failure and Recovery" 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.

8.3. 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 used to increase fairness and not correct operation. 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.

8.4. 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 following events cause implicit renewal of all of the leases for a given client (i.e. all those sharing a given clientid). Each of these is a positive indication that the client is still active and that the associated state held at the server, for the client, is still valid.

- An OPEN with a valid clientid.

- Any operation made with a valid stateid (CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, OPEN_CONFIRM, READ, RENEW, SETATTR, WRITE). This does not include the special stateids 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 operation. The use of the SETCLIENTID operation (possibly with the addition of the optional SETCLIENTID_CONFIRM operation) notifies the server to drop the locking state associated with the client.
  >
  > If the server has rebooted, the stateids (NFS4ERR_STALE_STATEID error) or the clientid (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.

8.5. 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.

8.5.1. 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, 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 clientid as a result of the SETCLIENTID operation. The client then confirms the use of the verifier with SETCLIENTID_CONFIRM. The clientid in combination with an opaque owner field is then used by the client to identify the lock 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 clientid which was derived from the old verifier.

For secure environments, a change in the verifier must only cause the release of locks associated with the authenticated requester. This is required to prevent a rogue entity from freeing otherwise valid locks.

Note that the verifier must have the same uniqueness properties of the verifier for the COMMIT operation.

8.5.2. 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 clientid invalidated by reboot or restart. When either of these are received, the client must establish a new clientid (See the section "Client ID") 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 CLAIM_PREVIOUS). 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 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 can not 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.

8.5.3. 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 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.

As a courtesy to the client or as an optimization, the server may continue to hold locks on behalf of a client for which recent communication has extended beyond the lease period. If the server receives a lock or I/O request that conflicts with one of these courtesy locks, the server must free the courtesy lock and grant the new request.

If the server continues to hold locks beyond the expiration of a client's lease, the server MUST employ a method of recording this fact in its stable storage. Conflicting locks requests from another client may be serviced after the lease expiration. There are various scenarios involving server failure after such an event that require the storage of these lease expirations or network partitions. One scenario is as follows:

> A client holds a lock at the server and encounters a network partition and is unable to renew the associated lease. A second client obtains a conflicting lock and then frees the lock. After the unlock request by the second client, the server reboots or reinitializes. Once the server recovers, the network partition heals and the original client attempts to reclaim the original lock.

In this scenario and without any state information, the server will allow the reclaim and the client will be in an inconsistent state because the server or the client has no knowledge of the conflicting lock.

The server may choose to store this lease expiration or network partitioning state in a way that will only identify the client as a whole. Note that this may potentially lead to lock reclaims being denied unnecessarily because of a mix of conflicting and non-conflicting locks. The server may also choose to store information about each lock that has an expired lease with an associated conflicting lock. The choice of the amount and type of state information that is stored is left to the implementor. In any case, the server must have enough state information to enable correct recovery from multiple partitions and multiple server failures.

8.6. 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 nfs_lockowner. This is straightforward to do without a special re-synchronize operation.

Since the server maintains the last lock request and response received on the nfs_lockowner, for each nfs_lockowner, 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 nfs_lockowner, 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 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 nfs_lockowner will re-synchronize and in turn the lock state will re-synchronize.

8.7. 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 period. 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_EXPIRED and the error is received within the lease period for the lock. In this instance the client may assume that only the nfs_lockowner's locks have been lost. The client notifies the lock holder appropriately. The client may not assume the lease period has been renewed as a result of 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 the previous section on crash recovery, 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.

8.8. Share Reservations

A share reservation is a mechanism to control access to a file. It is a separate and independent mechanism from record 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 (deny NONE, READ, WRITE, or BOTH). If the OPEN fails the client will fail the application's open request.

Pseudo-code definition of the semantics:

```
            if ((request.access & file_state.deny)) ||
                  (request.deny & file_state.access))
                          return (NFS4ERR_DENIED)
```

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;
```

8.9. 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 if any access 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. For clients that do not have a deny mode built into their open programming interfaces, deny equal to NONE should be used.

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 locks held by the nfs_lockowner on that file. If record 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 record locks held. The server MUST return failure 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, a file opened with deny READ/WRITE cannot be accessed using a filehandle obtained through LOOKUP because it would not have a valid stateid (i.e. using a stateid of all bits 0 or all bits 1).

8.10. Open Upgrade and Downgrade

When an OPEN is done for a file and the lockowner 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 OPEN’s. 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 OPEN’s result in the OPEN’ed 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 OPEN’s 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 OPEN’s with separate stateid’s and will require separate CLOSE’s 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 open’s 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.

8.11. 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 large internet 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 (server must wait for leases to expire and grace period 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 are not an issue.

8.12. Clocks 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.

8.13. 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, stateid’s, and clientid’s) 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 "File Locking and Share Reservations"

8.13.1. 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 stateid’s assigned by the original server. Therefore the new server must recognize these stateid’s as valid. This holds true for the clientid 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 new 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, 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.

8.13.2. 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, stateid’s and clientid’s 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 filesystems, 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.

8.13.3. 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 (i.e. OPEN, CLOSE, READ, WRITE, RENEW, LOCK, LOCKT, LOCKU), 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.

When a client receives an NFS4ERR_LEASE_MOVED error, it should perform some operation, such as a RENEW, on each file system associated with the server in question. 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, it will receive either NFS4ERR_STALE_CLIENTID or NFS4ERR_STALE_STATEID from the new server, as described above, and can then recover state information as it does in the event of server failure.

9. 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 NFS version 4 protocol uses many techniques similar to those that have been used in previous protocol versions. The NFS version 4 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 NFS version 4 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.

9.1. 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 revalidation 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 the section "Data Caching and File Locking") 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 NFS version 4 protocol provides more aggressive caching strategies with the following design goals:

- Compatibility with a large range of server semantics.

- Provide the same caching benefits as previous versions of the NFS protocol when unable to provide the more aggressive model.

- 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 the section "Open Delegation").

9.2. 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 clientid and may be used on behalf of all the nfs_lockowners 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 receive a response until the recall is complete. The recall is considered complete when the client returns the delegation or the server times out on the recall and revokes the delegation as a result of the timeout. 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 the section "Open Delegation"). 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.

9.2.1. Delegation Recovery

There are three situations that delegation recovery must deal with:

- Client reboot or restart
- Server reboot or restart
- Network partition (full or callback-only)

In the event the client reboots or restarts, the failure to renew leases will result in the revocation of record 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 extend the period for delegation recovery beyond the typical lease expiration period. 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 be prepared for delays that occur because of a conflicting delegation. This longer interval would increase the window for clients to reboot and consult stable storage so that the delegations can be reclaimed. For open delegations, such delegations are reclaimed using OPEN with a claim type of CLAIM_DELEGATE_PREV. (see the sections on "Data Caching and Revocation" and "Operation 18: OPEN" for discussion of open delegation and the details of OPEN respectively).

When the server reboots or restarts, delegations are reclaimed (using the OPEN operation with CLAIM_DELEGATE_PREV) in a similar fashion to record 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 NFS Version 4 protocol:

- Upon reclaim, a client reporting resources assigned to it by an earlier server instance must be granted those resources.

- The server has unquestionable authority to determine whether delegations are to be granted and, once granted, whether they are to be continued.

- The use of callbacks is not to be depended upon until the client has proven its ability to receive them.

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. For delegations, however, the server may extend the period in 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 the same effect. 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 the error NFS4ERR_EXPIRED. It also may find out about delegation revocation after a client reboot when it attempts to reclaim a delegation and receives that same error. Note that in the case of a revoked write open delegation, there are issues because data may have been modified by the client whose delegation is revoked and separately by other clients. See the section "Revocation Recovery for Write Open Delegation" 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 the section "Crash Recovery"). 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.

9.3. 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 record locks are the facilities the NFS version 4 protocol provides to allow applications to coordinate access by providing mutual exclusion facilities. The NFS version 4 protocol's data caching must be implemented such that it does not invalidate the assumptions that those using these facilities depend upon.

9.3.1. Data Caching and OPENs

In order to avoid invalidating the sharing assumptions that applications rely on, NFS version 4 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 the section "Open Delegation") two additional rules apply. Note that these rules are obeyed in practice by many NFS version 2 and version 3 clients.

- First, cached data present on a client must be revalidated after doing an OPEN. 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 NFS version 3 protocol’s practice for the benefit of users assuming this degree of cache revalidation.

- 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 client OPENs as 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.

9.3.2. 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 NFS version 4 protocol unless clients refrain from data caching.

The rules for data caching in the file locking environment are:

- First, when a client obtains a file lock for a particular region, the data cache corresponding to that region (if any cache 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.

- 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 pre-requisite 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 record locking in non-standard ways (e.g. using a record lock as a global semaphore) by flushing to the server more data upon an 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 record lock and those for which there are modifications not covered by a record lock. Any writes done for the former class of files must not include areas not locked and thus not modified on the client.

9.3.3. 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 in the result flags for an OPEN. 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.

9.3.4. Data Caching and File Identity

When clients cache data, the file data needs to organized according to the file system object to which the data belongs. For NFS version 3 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 NFS version 4 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 NFS version 4 protocol alleviates a potential functional regression in comparison with the NFS version 3 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 NFS version 4 client to determine whether two distinct filehandles denote the same server side object:

- If GETATTR directed to two filehandles have different values of the fsid attribute, then the filehandles represent distinct objects.

- 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.

- If GETATTR directed to the two filehandles does not return the fileid attribute for one or both of the handles, then the 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.

- If GETATTR directed to the two filehandles returns different values for the fileid attribute, then they are distinct objects.

- Otherwise they are the same object.

9.4. 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:

- 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.

- The client must have responded properly to previous recalls.

- There must be no current open conflicting with the requested delegation.

- There should be no current delegation that conflicts with the delegation being requested.

- The probability of future conflicting open requests should be low based on the recent history of the file.

- 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, read and write. A read open delegation allows a client to handle, on its own, requests to open a file for reading that do not deny read access to others. Multiple read open delegations may be outstanding simultaneously and do not conflict. A write open delegation allows the client to handle, on its own, all opens. Only one write open delegation may exist for a given file at a given time and it is inconsistent with any read open delegations.

When a client has a read open delegation, it may not make any changes to the contents or attributes of the file but it is assured that no other client may do so. When a client has a write open delegation, it may modify the file data since no other client will be accessing the file’s data. The client holding a write delegation may only affect file attributes which are intimately connected with the file data: object_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 read open 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:

- the type of delegation (read or write)

- space limitation information to control flushing of data on close (write open delegation only, see the section "Open Delegation and Data Caching")

- an nfsace4 specifying read and write permissions

- a stateid to represent the delegation for READ and WRITE

The stateid is separate and distinct from the stateid for the OPEN proper. The standard stateid, unlike the delegation stateid, is associated with a particular nfs_lockowner 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.

- The access and deny bits for the request and the file as described in the section "Share Reservations".

- 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:

- If the nfsace4 indicates that the open may be done, then it should be granted without reference to the server.

- 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.

9.4.1. 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. The continued endurance of the "read open delegation" provides a guarantee that no OPEN for write and thus no write has occurred. Similarly, when closing a file opened for write and if write open 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 write open delegation. A WRITE with a special stateid done by another client will force a recall of read open delegations.

With delegations, a client is able to avoid writing data to the server when the CLOSE of a file is serviced. The CLOSE operation 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 write open 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 of 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.

9.4.2. Open Delegation and File Locks

When a client holds a write open delegation, lock operations are 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.

9.4.3. Recall of Open Delegation

The following events necessitate recall of an open delegation:

- Potentially conflicting OPEN request (or READ/WRITE done with "special" stateid)

- SETATTR issued by another client

- REMOVE request for the file

- 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.

The server needs to employ special handling for a GETATTR where the target is a file that has a write open delegation in effect. In this case, the client holding the delegation needs to be interrogated. The server will use a CB_GETATTR callback, if the GETATTR attribute bits include any of the attributes that a write open delegate may modify (object_size, time_modify, change).

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:

- 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.

- 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 the section "Operation 18: OPEN" for details.)

- If there are granted file locks, the corresponding LOCK operations need to be performed. This applies to the write open delegation case only.

- For a write open 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.

- For a write open 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.

- With the write open 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 UNCHECKED with a object_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 write open delegation, file locking imposes some additional requirements. The flushing of any modified data in any region for which a write lock was released while the write open delegation was in effect is what is required to precisely maintain the associated invariant. However, because the write open 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 write open delegation was in effect.

9.4.4. 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 the section "Revocation Recovery for Write Open Delegation" for additional details.

9.5. Data Caching and Revocation

When locks and delegations are revoked, the assumptions upon which successful caching depend are no longer guaranteed. The owner of the locks or share reservations which have been revoked 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.

9.5.1. Revocation Recovery for Write Open Delegation

Revocation recovery for a write open 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. Unless the client can determine that the file has not modified by any other client, this technique must be limited to situations in which a client has a complete cached copy of the file in question. Use of such a technique may be limited to files under a certain size or may only be used when sufficient disk space is guaranteed to be available within the target file system and when the client has sufficient buffering resources to keep the cached copy available until it is properly stored to the target file system.

9.6. 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 ordinary 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 object_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_RECALL 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 incoherences mentioned above can be reasonably managed. These rules are derived from the practice of previous NFS protocols.

- 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.
- An upper time boundary is maintained on how long a client cache entry can be kept without being refreshed from the server.
- When operations are performed that change 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 only the change attribute and assuming that if the change attribute has the same value as it did when the attributes were cached, then no attributes have changed. The possible exception is the attribute time_access.

9.7. 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, the following rules should be followed:

- The results of unsuccessful LOOKUPs should not be cached, unless they are specifically reverified at the point of use.

- 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.

9.8. 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:

- 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.

- 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.

10. Minor Versioning

To address the requirement of an NFS protocol that can evolve as the need arises, the NFS version 4 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 zero of the NFS version 4 protocol is represented by this RFC. The COMPOUND procedure will 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, NFS version 4 minor versions will not add 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.

2.1 Minor versions may append attributes to GETATTR4args, bitmap4, and GETATTR4res.

This allows for the expansion of the attribute model to allow for future growth or adaptation.

2.2 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 will 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

- adding bits to existing attributes like ACLs that have flag words
- extending enumerated types (including NFS4ERR_*) with new values

4 Minor versions may not modify the structure of existing attributes.

5 Minor versions may not delete operations.

This prevents the potential reuse of a particular operation "slot" in a future minor version.

6 Minor versions may not delete attributes.

7 Minor versions may not delete flag bits or enumeration values.

8 Minor versions may declare an operation as mandatory to NOT implement.

Specifying an operation as "mandatory to not implement" is equivalent to obsoleting an operation. For the client, it means that the operation should 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.

8.1 Minor versions may declare attributes mandatory to NOT implement.

8.2 Minor versions may declare flag bits or enumeration values as mandatory to NOT implement.

9 Minor versions may downgrade features from mandatory to recommended, or recommended to optional.

10 Minor versions may upgrade features from optional to recommended or recommended to mandatory.

11 A client and server that support minor version X must support minor versions 0 (zero) through X-1 as well.

12 No new features may be introduced as mandatory 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 mandatory.

13 A client MUST NOT attempt to use a stateid, file handle, or similar returned object from the COMPOUND procedure with minor version X for another COMPOUND procedure with minor version Y, where X != Y.

11. Internationalization

The primary issue in which NFS needs to deal with internationalization, or I18n, is with respect to file names and other strings as used within the protocol. The choice of string representation must allow reasonable name/string access to clients which use various languages. The UTF-8 encoding of the UCS as defined by [ISO10646] allows for this type of access and follows the policy described in "IETF Policy on Character Sets and Languages", [RFC2277]. This choice is explained further in the following.

11.1. Universal Versus Local Character Sets

[RFC1345] describes a table of 16 bit characters for many different languages (the bit encodings match Unicode, though of course RFC1345 is somewhat out of date with respect to current Unicode assignments). Each character from each language has a unique 16 bit value in the 16 bit character set. Thus this table can be thought of as a universal character set. [RFC1345] then talks about groupings of subsets of the entire 16 bit character set into "Charset Tables". For example one might take all the Greek characters from the 16 bit table (which are consecutively allocated), and normalize their offsets to a table that fits in 7 bits. Thus it is determined that "lower case alpha" is in the same position as "upper case a" in the US-ASCII table, and "upper case alpha" is in the same position as "lower case a" in the US-ASCII table.

These normalized subset character sets can be thought of as "local character sets", suitable for an operating system locale.

Local character sets are not suitable for the NFS protocol. Consider someone who creates a file with a name in a Swedish character set. If someone else later goes to access the file with their locale set to the Swedish language, then there are no problems. But if someone in say the US-ASCII locale goes to access the file, the file name will look very different, because the Swedish characters in the 7 bit table will now be represented in US-ASCII characters on the display. It would be preferable to give the US-ASCII user a way to display the file name using Swedish glyphs. In order to do that, the NFS protocol would have to include the locale with the file name on each operation to create a file.

But then what of the situation when there is a path name on the server like:

```
/component-1/component-2/component-3
```

Each component could have been created with a different locale. If one issues CREATE with multi-component path name, and if some of the leading components already exist, what is to be done with the existing components? Is the current locale attribute replaced with the user's current one? These types of situations quickly become too complex when there is an alternate solution.

If the NFS version 4 protocol used a universal 16 bit or 32 bit character set (or an encoding of a 16 bit or 32 bit character set into octets), then the server and client need not care if the locale of the user accessing the file is different than the locale of the user who created the file. The unique 16 bit or 32 bit encoding of the character allows for determination of what language the character is from and also how to display that character on the client. The server need not know what locales are used.

11.2. Overview of Universal Character Set Standards

The previous section makes a case for using a universal character set. This section makes the case for using UTF-8 as the specific universal character set for the NFS version 4 protocol.

[RFC2279] discusses UTF-* (UTF-8 and other UTF-XXX encodings), Unicode, and UCS-*. There are two standards bodies managing universal code sets:

- ISO/IEC which has the standard 10646-1

- Unicode which has the Unicode standard

Both standards bodies have pledged to track each other's assignments of character codes.

The following is a brief analysis of the various standards.

UCS	Universal Character Set. This is ISO/IEC 10646-1: "a multi-octet character set called the Universal Character Set (UCS), which encompasses most of the world's writing systems."

UCS-2	a two octet per character encoding that addresses the first 2^16 characters of UCS. Currently there are no UCS characters beyond that range.
UCS-4	a four octet per character encoding that permits the encoding of up to 2^31 characters.
UTF	UTF is an abbreviation of the term "UCS transformation format" and is used in the naming of various standards for encoding of UCS characters as described below.
UTF-1	Only historical interest; it has been removed from 10646-1
UTF-7	Encodes the entire "repertoire" of UCS "characters using only octets with the higher order bit clear". [RFC2152] describes UTF-7. UTF-7 accomplishes this by reserving one of the 7bit US-ASCII characters as a "shift" character to indicate non-US-ASCII characters.
UTF-8	Unlike UTF-7, uses all 8 bits of the octets. US-ASCII characters are encoded as before unchanged. Any octet with the high bit cleared can only mean a US-ASCII character. The high bit set means that a UCS character is being encoded.
UTF-16	Encodes UCS-4 characters into UCS-2 characters using a reserved range in UCS-2.
Unicode	Unicode and UCS-2 are the same; [RFC2279] states: Up to the present time, changes in Unicode and amendments to ISO/IEC 10646 have tracked each other, so that the character repertoires and code point assignments have remained in sync. The relevant standardization committees have committed to maintain this very useful synchronism.

11.3. Difficulties with UCS-4, UCS-2, Unicode

Adapting existing applications, and file systems to multi-octet schemes like UCS and Unicode can be difficult. A significant amount of code has been written to process streams of bytes. Also there are many existing stored objects described with 7 bit or 8 bit characters. Doubling or quadrupling the bandwidth and storage requirements seems like an expensive way to accomplish I18N.

UCS-2 and Unicode are "only" 16 bits long. That might seem to be enough but, according to [Unicode1], 49,194 Unicode characters are already assigned. According to [Unicode2] there are still more languages that need to be added.

11.4. UTF-8 and its solutions

UTF-8 solves problems for NFS that exist with the use of UCS and Unicode. UTF-8 will encode 16 bit and 32 bit characters in a way that will be compact for most users. The encoding table from UCS-4 to UTF-8, as copied from [RFC2279]:

UCS-4 range (hex.)	UTF-8 octet sequence (binary)
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-001F FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0020 0000-03FF FFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0400 0000-7FFF FFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

See [RFC2279] for precise encoding and decoding rules. Note because of UTF-16, the algorithm from Unicode/UCS-2 to UTF-8 needs to account for the reserved range between D800 and DFFF.

Note that the 16 bit UCS or Unicode characters require no more than 3 octets to encode into UTF-8

Interestingly, UTF-8 has room to handle characters larger than 31 bits, because the leading octet of form:

    1111111x

is not defined. If needed, ISO could either use that octet to indicate a sequence of an encoded 8 octet character, or perhaps use 11111110 to permit the next octet to indicate an even more expandable character set.

So using UTF-8 to represent character encodings means never having to run out of room.

11.5. 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 the various permutations.

The NFS version 4 protocol does not mandate the use of a particular normalization form at this time. A later revision of this specification may 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).

12. Error Definitions

NFS error numbers are assigned to failed operations within a 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.

A description of each defined error follows:

NFS4_OK	Indicates the operation completed successfully.
NFS4ERR_ACCES	Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with NFS4ERR_PERM, which restricts itself to owner or privileged user permission failures.
NFS4ERR_BADHANDLE	Illegal NFS file handle. The file handle failed internal consistency checks.
NFS4ERR_BADTYPE	An attempt was made to create an object of a type not supported by the server.
NFS4ERR_BAD_COOKIE	READDIR cookie is stale.
NFS4ERR_BAD_SEQID	The sequence number in a locking request is neither the next expected number or the last number processed.

NFS4ERR_BAD_STATEID	A stateid generated by the current server instance, but which does not designate any locking state (either current or superseded) for a current lockowner-file pair, was used.
NFS4ERR_CLID_INUSE	The SETCLIENTID procedure has found that a client id is already in use by another client.
NFS4ERR_DELAY	The server initiated the request, but was not able to complete it in a timely fashion. The client should wait and then try the request with a new RPC transaction ID. For example, this error should be returned from a server that supports hierarchical storage and receives a request to process a file that has been migrated. In this case, the server should start the immigration process and respond to client with this error. This error may also occur when a necessary delegation recall makes processing a request in a timely fashion impossible.
NFS4ERR_DENIED	An attempt to lock a file is denied. Since this may be a temporary condition, the client is encouraged to retry the lock request until the lock is accepted.
NFS4ERR_DQUOT	Resource (quota) hard limit exceeded. The user’s resource limit on the server has been exceeded.
NFS4ERR_EXIST	File exists. The file specified already exists.
NFS4ERR_EXPIRED	A lease has expired that is being used in the current procedure.
NFS4ERR_FBIG	File too large. The operation would have caused a file to grow beyond the server’s limit.
NFS4ERR_FHEXPIRED	The file handle provided is volatile and has expired at the server.
NFS4ERR_GRACE	The server is in its recovery or grace period which should match the lease period of the server.

NFS4ERR_INVAL	Invalid argument or unsupported argument for an operation. Two examples are attempting a READLINK on an object other than a symbolic link or attempting to SETATTR a time field on a server that does not support this operation.
NFS4ERR_IO	I/O error. A hard error (for example, a disk error) occurred while processing the requested operation.
NFS4ERR_ISDIR	Is a directory. The caller specified a directory in a non-directory operation.
NFS4ERR_LEASE_MOVED	A lease being renewed is associated with a file system that has been migrated to a new server.
NFS4ERR_LOCKED	A read or write operation was attempted on a locked file.
NFS4ERR_LOCK_RANGE	A lock request is operating on a sub-range of a current lock for the lock owner and the server does not support this type of request.
NFS4ERR_MINOR_VERS_MISMATCH	The server has received a request that specifies an unsupported minor version. The server must return a COMPOUND4res with a zero length operations result array.
NFS4ERR_MLINK	Too many hard links.
NFS4ERR_MOVED	The filesystem which contains the current filehandle object has been relocated or migrated to another server. The client may obtain the new filesystem location by obtaining the "fs_locations" attribute for the current filehandle. For further discussion, refer to the section "Filesystem Migration or Relocation".
NFS4ERR_NAMETOOLONG	The filename in an operation was too long.
NFS4ERR_NODEV	No such device.
NFS4ERR_NOENT	No such file or directory. The file or directory name specified does not exist.

NFS4ERR_NOFILEHANDLE	The logical current file handle value has not been set properly. This may be a result of a malformed COMPOUND operation (i.e. no PUTFH or PUTROOTFH before an operation that requires the current file handle be set).
NFS4ERR_NOSPC	No space left on device. The operation would have caused the server's file system to exceed its limit.
NFS4ERR_NOTDIR	Not a directory. The caller specified a non-directory in a directory operation.
NFS4ERR_NOTEMPTY	An attempt was made to remove a directory that was not empty.
NFS4ERR_NOTSUPP	Operation is not supported.
NFS4ERR_NOT_SAME	This error is returned by the VERIFY operation to signify that the attributes compared were not the same as provided in the client's request.
NFS4ERR_NXIO	I/O error. No such device or address.
NFS4ERR_OLD_STATEID	A stateid which designates the locking state for a lockowner-file at an earlier time was used.
NFS4ERR_PERM	Not owner. The operation was not allowed because the caller is either not a privileged user (root) or not the owner of the target of the operation.
NFS4ERR_READDIR_NOSPC	The encoded response to a READDIR request exceeds the size limit set by the initial request.
NFS4ERR_RESOURCE	For the processing of the COMPOUND procedure, the server may exhaust available resources and can not continue processing procedures within the COMPOUND operation. This error will be returned from the server in those instances of resource exhaustion related to the processing of the COMPOUND procedure.
NFS4ERR_ROFS	Read-only file system. A modifying operation was attempted on a read-only file system.

NFS4ERR_SAME	This error is returned by the NVERIFY operation to signify that the attributes compared were the same as provided in the client's request.
NFS4ERR_SERVERFAULT	An error occurred on the server which does not map to any of the legal NFS version 4 protocol error values. The client should translate this into an appropriate error. UNIX clients may choose to translate this to EIO.
NFS4ERR_SHARE_DENIED	An attempt to OPEN a file with a share reservation has failed because of a share conflict.
NFS4ERR_STALE	Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.
NFS4ERR_STALE_CLIENTID	A clientid not recognized by the server was used in a locking or SETCLIENTID_CONFIRM request.
NFS4ERR_STALE_STATEID	A stateid generated by an earlier server instance was used.
NFS4ERR_SYMLINK	The current file handle provided for a LOOKUP is not a directory but a symbolic link. Also used if the final component of the OPEN path is a symbolic link. NFS4ERR_TOOSMALL Buffer or request is too small.
NFS4ERR_WRONGSEC	The security mechanism being used by the client for the procedure does not match the server's security policy. The client should change the security mechanism being used and retry the operation.
NFS4ERR_XDEV	Attempt to do a cross-device hard link.

13. NFS Version 4 Requests

For the NFS version 4 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 NFS version 4 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 NFS version 4 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 operation and therefore is fixed for the life of the client instantiation.

13.1. 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 basics of the COMPOUND procedures construction is:

```
                +-----------+-----------+-----------+--
                | op + args | op + args | op + args |
                +-----------+-----------+-----------+--
```

and the reply looks like this:

```
      +------------+-----------------------+-----------------------+--
      |last status | status + op + results | status + op + results |
      +------------+-----------------------+-----------------------+--
```

13.2. 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.

It is the client's responsibility for recovering from any partially completed COMPOUND procedure. Partially completed COMPOUND procedures may occur at any point due to errors such as NFS4ERR_RESOURCE and NFS4ERR_LONG_DELAY. This may occur even given 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.

13.3. Synchronous Modifying Operations

NFS version 4 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 UNSTABLE 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.

13.4. 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.

14. NFS Version 4 Procedures

14.1. Procedure 0: NULL - No Operation

SYNOPSIS

```
<null>
```

ARGUMENT

```
void;
```

RESULT

```
void;
```

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.

ERRORS

None.

14.2. Procedure 1: COMPOUND - Compound Operations

SYNOPSIS

```
compoundargs -> compoundres
```

ARGUMENT

```
union nfs_argop4 switch (nfs_opnum4 argop) {
        case <OPCODE>: <argument>;
        ...
};
```

```
struct COMPOUND4args {
        utf8string      tag;
        uint32_t        minorversion;
        nfs_argop4      argarray<>;
};
```

RESULT

```
union nfs_resop4 switch (nfs_opnum4 resop){
        case <OPCODE>: <result>;
        ...
};

struct COMPOUND4res {
        nfsstat4        status;
        utf8string      tag;
        nfs_resop4      resarray<>;
};
```

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 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. If the server receives an operation array with either of these included, an error of NFS4ERR_NOTSUPP must be returned. 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_NOTSUPP is returned. 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.

IMPLEMENTATION

Note that the definition of the "tag" in both the request and response are 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.

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.

ERRORS

All errors defined in the protocol

14.2.1. Operation 3: ACCESS - Check Access Rights

SYNOPSIS

```
(cfh), accessreq -> supported, accessrights
```

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;
};
```

RESULT

```
struct ACCESS4resok {
        uint32_t        supported;
        uint32_t        access;
};

union ACCESS4res switch (nfsstat4 status) {
 case NFS4_OK:
         ACCESS4resok   resok4;
 default:
         void;
};
```

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 was 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 (no meaning for non-directory objects).
ACCESS4_EXECUTE	Execute file (no meaning for a directory).

On success, the current filehandle retains its value.

IMPLEMENTATION

For the NFS version 4 protocol, the use of the ACCESS procedure when opening a regular file is deprecated in favor of using OPEN.

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 can not reliably perform an access check with only current file attributes.

In the NFS version 2 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 procedure in the NFS version 4 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 will 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 afterwards. 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.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.2. Operation 4: CLOSE - Close File

SYNOPSIS

```
(cfh), seqid, stateid -> stateid
```

ARGUMENT

```
struct CLOSE4args {
        /* CURRENT_FH: object */
        seqid4          seqid
        stateid4        stateid;
};
```

RESULT

```
union CLOSE4res switch (nfsstat4 status) {
 case NFS4_OK:
         stateid4       stateid;
 default:
         void;
};
```

DESCRIPTION

The CLOSE operation releases share reservations for the 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 record 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 record locks held. The server MUST return failure if any locks would exist after the CLOSE.

On success, the current filehandle retains its value.

IMPLEMENTATION

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_BAD_SEQID
NFS4ERR_BAD_STATEID
NFS4ERR_DELAY

```
         NFS4ERR_EXPIRED
         NFS4ERR_FHEXPIRED
         NFS4ERR_GRACE
         NFS4ERR_INVAL
         NFS4ERR_ISDIR
         NFS4ERR_LEASE_MOVED
         NFS4ERR_MOVED
         NFS4ERR_NOFILEHANDLE
         NFS4ERR_OLD_STATEID
         NFS4ERR_RESOURCE
         NFS4ERR_SERVERFAULT
         NFS4ERR_STALE
         NFS4ERR_STALE_STATEID
```

14.2.3. Operation 5: COMMIT - Commit Cached Data

SYNOPSIS

```
         (cfh), offset, count -> verifier
```

ARGUMENT

```
         struct COMMIT4args {
                 /* CURRENT_FH: file */
                 offset4         offset;
                 count4          count;
         };
```

RESULT

```
         struct COMMIT4resok {
                 verifier4       writeverf;
         };

         union COMMIT4res switch (nfsstat4 status) {
          case NFS4_OK:
                  COMMIT4resok   resok4;
          default:
                  void;
         };
```

DESCRIPTION

The COMMIT operation forces or flushes data to stable storage for the file specified by the current file handle. 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 procedure. 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.

IMPLEMENTATION

The COMMIT procedure is similar in operation and semantics to the POSIX fsync(2) 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(2), 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(2) 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.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_FHEXPIRED
NFS4ERR_IO

```
NFS4ERR_ISDIR
NFS4ERR_LOCKED
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_RESOURCE
NFS4ERR_ROFS
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC
```

14.2.4. Operation 6: CREATE - Create a Non-Regular File Object

SYNOPSIS

```
(cfh), name, type -> (cfh), change_info
```

ARGUMENT

```
union createtype4 switch (nfs_ftype4 type) {
 case NF4LNK:
         linktext4      linkdata;
 case NF4BLK:
 case NF4CHR:
         specdata4      devdata;
 case NF4SOCK:
 case NF4FIFO:
 case NF4DIR:
         void;
};

struct CREATE4args {
        /* CURRENT_FH: directory for creation */
        component4      objname;
        createtype4     objtype;
};
```

RESULT

```
struct CREATE4resok {
        change_info4     cinfo;
};

union CREATE4res switch (nfsstat4 status) {
 case NFS4_OK:
         CREATE4resok resok4;
 default:
         void;
};
```

DESCRIPTION

The CREATE operation creates a non-regular file object in a directory with a given name. The OPEN procedure MUST be used to create a regular file.

The objname specifies the name for the new object. If the objname has a length of 0 (zero), the error NFS4ERR_INVAL will be returned. 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 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.

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.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_BADTYPE
NFS4ERR_DQUOT
NFS4ERR_EXIST
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NAMETOOLONG
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOSPC
NFS4ERR_NOTDIR
NFS4ERR_NOTSUPP

NFS4ERR_RESOURCE
NFS4ERR_ROFS
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.5. Operation 7: DELEGPURGE - Purge Delegations Awaiting Recovery

SYNOPSIS

```
clientid ->
```

ARGUMENT

```
 struct DELEGPURGE4args {
         clientid4       clientid;
};
```

RESULT

```
struct DELEGPURGE4res {
        nfsstat4        status;
};
```

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 should be used by clients that record delegation information on stable storage on the client. In this case, DELEGPURGE should be issued immediately after doing delegation recovery on all delegations know 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. The set of delegations known to the server and the client may be different. The reason for this is that 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.

ERRORS

NFS4ERR_RESOURCE

NFS4ERR_SERVERFAULT
NFS4ERR_STALE_CLIENTID

14.2.6. Operation 8: DELEGRETURN - Return Delegation

SYNOPSIS

```
stateid ->
```

ARGUMENT

```
struct DELEGRETURN4args {
        stateid4        stateid;
};
```

RESULT

```
struct DELEGRETURN4res {
        nfsstat4        status;
};
```

DESCRIPTION

Returns the delegation represented by the given stateid.

ERRORS

NFS4ERR_BAD_STATEID
NFS4ERR_OLD_STATEID
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE_STATEID

14.2.7. Operation 9: GETATTR - Get Attributes

SYNOPSIS

```
(cfh), attrbits -> attrbits, attrvals
```

ARGUMENT

```
struct GETATTR4args {
        /* CURRENT_FH: directory or file */
        bitmap4         attr_request;
};
```

RESULT

```
         struct GETATTR4resok {
                 fattr4          obj_attributes;
         };

         union GETATTR4res switch (nfsstat4 status) {
          case NFS4_OK:
                  GETATTR4resok  resok4;
          default:
                  void;
         };
```

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, 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 but cannot obtain its value. In that case no attribute values will be returned.

All servers must support the mandatory attributes as specified in the section "File Attributes".

On success, the current filehandle retains its value.

IMPLEMENTATION

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT

NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.8. Operation 10: GETFH - Get Current Filehandle

SYNOPSIS

```
(cfh) -> filehandle
```

ARGUMENT

```
/* CURRENT_FH: */
void;
```

RESULT

```
struct GETFH4resok {
        nfs_fh4         object;
};

union GETFH4res switch (nfsstat4 status) {
 case NFS4_OK:
        GETFH4resok     resok4;
 default:
        void;
};
```

DESCRIPTION

This operation returns the current filehandle value.

On success, the current filehandle retains its value.

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
```

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_FHEXPIRED

NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.9. Operation 11: LINK - Create Link to a File

SYNOPSIS

```
(sfh), (cfh), newname -> (cfh), change_info
```

ARGUMENT

```
struct LINK4args {
        /* SAVED_FH: source object */
        /* CURRENT_FH: target directory */
        component4      newname;
};
```

RESULT

```
struct LINK4resok {
        change_info4    cinfo;
};

union LINK4res switch (nfsstat4 status) {
 case NFS4_OK:
         LINK4resok resok4;
 default:
         void;
};
```

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.

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.

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 comments under RENAME regarding object and target residing on the same file system apply here as well. The comments regarding the target name applies as well.

Note that symbolic links are created with the CREATE operation.

ERRORS

NFS4ERR_ACCES NFS4ERR_BADHANDLE NFS4ERR_DELAY NFS4ERR_DQUOT NFS4ERR_EXIST NFS4ERR_FHEXPIRED NFS4ERR_INVAL NFS4ERR_IO NFS4ERR_ISDIR NFS4ERR_MLINK NFS4ERR_MOVED NFS4ERR_NAMETOOLONG NFS4ERR_NOFILEHANDLE NFS4ERR_NOSPC NFS4ERR_NOTDIR NFS4ERR_NOTSUPP NFS4ERR_RESOURCE NFS4ERR_ROFS NFS4ERR_SERVERFAULT NFS4ERR_STALE NFS4ERR_WRONGSEC NFS4ERR_XDEV

14.2.10. Operation 12: LOCK - Create Lock

SYNOPSIS

(cfh) type, seqid, reclaim, stateid, offset, length -> stateid, access

ARGUMENT

```
enum nfs4_lock_type {
        READ_LT         = 1,
        WRITE_LT        = 2,
        READW_LT        = 3,    /* blocking read */
        WRITEW_LT       = 4     /* blocking write */ };

struct LOCK4args {
        /* CURRENT_FH: file */
        nfs_lock_type4  locktype;
        seqid4          seqid;
        bool            reclaim;
        stateid4        stateid;
        offset4         offset;
```

```
            length4         length; };
```

RESULT

```
      struct LOCK4denied {
              nfs_lockowner4  owner;
              offset4         offset;
              length4         length; };

      union LOCK4res switch (nfsstat4 status) {
       case NFS4_OK:
               stateid4       stateid;
       case NFS4ERR_DENIED:
               LOCK4denied    denied;
       default:
               void; };
```

DESCRIPTION

The LOCK operation requests a record lock for the byte range specified by the offset and length parameters. The lock type is also specified to be one of the nfs4_lock_types. 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). To lock the entire file, use an offset of 0 (zero) and a length with all bits set to 1. A length of 0 is reserved and should not be used.

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.

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. The File Locking section contains a full description of this and the other file locking operations.

ERRORS

NFS4ERR_ACCES NFS4ERR_BADHANDLE NFS4ERR_BAD_SEQID

NFS4ERR_BAD_STATEID 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_OLD_STATEID NFS4ERR_RESOURCE NFS4ERR_SERVERFAULT NFS4ERR_STALE NFS4ERR_STALE_CLIENTID NFS4ERR_STALE_STATEID NFS4ERR_WRONGSEC

14.2.11. Operation 13: LOCKT - Test For Lock

SYNOPSIS

```
(cfh) type, owner, offset, length -> {void, NFS4ERR_DENIED ->
owner}
```

ARGUMENT

```
struct LOCKT4args {
        /* CURRENT_FH: file */
        nfs_lock_type4  locktype;
        nfs_lockowner4  owner;
        offset4         offset;
        length4         length; };
```

RESULT

```
union LOCKT4res switch (nfsstat4 status) {
 case NFS4ERR_DENIED:
         LOCK4denied    denied;
 case NFS4_OK:
         void;
 default:
         void; };
```

DESCRIPTION

The LOCKT operation tests the lock as specified in the arguments. If a conflicting lock exists, the owner, offset, and length of the conflicting lock are returned; if no lock is held, nothing other than NFS4_OK is returned.

On success, the current filehandle retains its value.

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. The File Locking section contains further discussion of the file locking mechanisms.

LOCKT uses nfs_lockowner4 instead of a stateid4, as LOCK does, to identify the owner so that the client does not have to open the file to test for the existence of a lock.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_DENIED
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
NFS4ERR_WRONGSEC

14.2.12. Operation 14: LOCKU - Unlock File

SYNOPSIS

```
(cfh) type, seqid, stateid, offset, length -> stateid
```

ARGUMENT

```
struct LOCKU4args {
        /* CURRENT_FH: file */
        nfs_lock_type4  locktype;
        seqid4          seqid;
        stateid4        stateid;
        offset4         offset;
        length4         length;
};
```

RESULT

```
union LOCKU4res switch (nfsstat4 status) {
 case   NFS4_OK:
```

```
               stateid4       stateid;
        default:
               void;
       };
```

DESCRIPTION

The LOCKU operation unlocks the record lock specified by the parameters.

On success, the current filehandle retains its value.

IMPLEMENTATION

The File Locking section contains a full description of this and the other file locking procedures.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_BAD_SEQID
NFS4ERR_BAD_STATEID
NFS4ERR_EXPIRED
NFS4ERR_FHEXPIRED
NFS4ERR_GRACE
NFS4ERR_INVAL
NFS4ERR_LOCK_RANGE
NFS4ERR_LEASE_MOVED
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_OLD_STATEID
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_STALE_CLIENTID
NFS4ERR_STALE_STATEID

14.2.13. Operation 15: LOOKUP - Lookup Filename

SYNOPSIS

(cfh), filenames -> (cfh)

ARGUMENT

```
struct LOOKUP4args {
        /* CURRENT_FH: directory */
```

```
           pathname4       path;
   };
```

RESULT

```
   struct LOOKUP4res {
           /* CURRENT_FH: object */
           nfsstat4        status;
   };
```

DESCRIPTION

This operation LOOKUPs or finds a file system object starting from the directory specified by the current filehandle. LOOKUP evaluates the pathname contained in the array of names and obtains a new current filehandle from the final name. All but the final name in the list must be the names of directories.

If the pathname cannot be evaluated either because a component does not exist or because the client does not have permission to evaluate a component of the path, then an error will be returned and the current filehandle will be unchanged.

If the path is a zero length array, if any component does not obey the UTF-8 definition, or if any component in the path is of zero length, the error NFS4ERR_INVAL will be returned.

IMPLEMENTATION

If the client prefers a partial evaluation of the path then a sequence of LOOKUP operations can be substituted e.g.

```
PUTFH  (directory filehandle)
LOOKUP "pub" "foo" "bar"
GETFH
```

or, if the client wishes to obtain the intermediate filehandles

```
PUTFH  (directory filehandle)
LOOKUP "pub"
GETFH
LOOKUP "foo"
GETFH
LOOKUP "bar"
GETFH
```

NFS version 4 servers depart from the semantics of previous NFS versions in allowing LOOKUP requests to cross mountpoints on the server. The client can detect a mountpoint 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 mountpoint. Unix clients that detect a mountpoint crossing will need to mount the server's filesystem. 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" filesystems should provide a pseudo-filesystem into which the exported filesystems can be integrated, so that clients can browse the server's name space. The clients view of a pseudo filesystem will be limited to paths that lead to exported filesystems.

Note: previous versions of the protocol assigned special semantics to the names "." and "..". NFS version 4 assigns no special semantics to these names. The LOOKUPP operator must be used to lookup a parent directory.

Note that this procedure 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 file handle 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.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
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

14.2.14. Operation 16: LOOKUPP - Lookup Parent Directory

SYNOPSIS

```
(cfh) -> (cfh)
```

ARGUMENT

```
/* CURRENT_FH: object */
void;
```

RESULT

```
struct LOOKUPP4res {
        /* CURRENT_FH: directory */
        nfsstat4        status;
};
```

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_ENOENT error must be returned. Therefore, NFS4ERR_ENOENT will be returned by the server when the current filehandle is at the root or top of the server’s file tree.

IMPLEMENTATION

As for LOOKUP, LOOKUPP will also cross mountpoints.

If the current filehandle is not a directory or named attribute directory, the error NFS4ERR_NOTDIR is returned.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOENT
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTDIR
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT

NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.15. Operation 17: NVERIFY - Verify Difference in Attributes

SYNOPSIS

```
(cfh), fattr -> -
```

ARGUMENT

```
struct NVERIFY4args {
        /* CURRENT_FH: object */
        fattr4          obj_attributes;
};
```

RESULT

```
struct NVERIFY4res {
        nfsstat4        status;
};
```

DESCRIPTION

This operation is used to prefix a sequence of operations to be performed if one or more attributes have changed on some filesystem object. If all the attributes match then the error NFS4ERR_SAME must be returned.

On success, the current filehandle retains its value.

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 "pub" "foo" "bar"
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_NOTSUPP is returned to the client.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTSUPP
NFS4ERR_RESOURCE
NFS4ERR_SAME
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.16. Operation 18: OPEN - Open a Regular File

SYNOPSIS

```
(cfh), claim, openhow, owner, seqid, access, deny -> (cfh),
stateid, cinfo, rflags, open_confirm, delegation
```

ARGUMENT

```
struct OPEN4args {
        open_claim4     claim;
        openflag4       openhow;
        nfs_lockowner4  owner;
        seqid4          seqid;
        uint32_t        share_access;
        uint32_t        share_deny;
};

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 {
           pathname4       file;
           stateid4        delegate_stateid;
   };

   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 */
            pathname4      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 */
            uint32_t        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 */
            pathname4      file_delegate_prev;
   };

    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;
   };

   const OPEN4_RESULT_MLOCK        = 0x00000001;
   const OPEN4_RESULT_CONFIRM= 0x00000002;

   struct OPEN4resok {
           stateid4        stateid;        /* Stateid for open */
           change_info4    cinfo;          /* Directory Change Info */
           uint32_t        rflags;         /* Result flags */
           verifier4       open_confirm;   /* OPEN_CONFIRM verifier */
           open_delegation4 delegation;    /* Info on any open
                                              delegation */
   };

   union OPEN4res switch (nfsstat4 status) {
    case NFS4_OK:
           /* CURRENT_FH: opened file */
           OPEN4resok      resok4;
    default:
           void;
   };
```

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.

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: UNCHECKED, GUARDED, or EXCLUSIVE.

UNCHECKED 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 may includes any writable attribute valid for regular files. When an UNCHECKED create encounters an existing file, the attributes specified by createattrs is not used, except that when an object_size of zero is specified, the existing file is truncated. If GUARDED 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 UNCHECKED.

EXCLUSIVE 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.

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 procedure provides for DOS SHARE capability with the use of the access and deny fields of the OPEN arguments. The client specifies at OPEN the required access and 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_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 the section on 'Share Reservations'.

In the case that the client is recovering state from a server failure, the reclaim 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 file handles; 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; used after the client reboots.

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 the section Open Delegation. 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_MLOCK indicates to the caller that mandatory locking is in effect for this file and the client should act appropriately with regard to data cached on the client. OPEN4_RESULT_CONFIRM indicates that the client MUST execute an OPEN_CONFIRM operation before using the open file.

If the file is a zero length array, if any component does not obey the UTF-8 definition, or if any component in the path is of zero length, the error NFS4ERR_INVAL will be returned.

When an OPEN is done and the specified lockowner 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 OPEN’s were completed.

IMPLEMENTATION

The OPEN procedure contains support for EXCLUSIVE create. The mechanism is similar to the support in NFS version 3 [RFC1813]. As in NFS version 3, this mechanism provides reliable exclusive creation. Exclusive create is invoked when the how parameter is EXCLUSIVE. 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 can not 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 GUARDED 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 procedure can fail with NFS4ERR_EXIST, even though the create was performed successfully.

For SHARE reservations, the client must specify a value for access that is one of READ, WRITE, or BOTH. For 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.

If the final component provided to OPEN is a symbolic link, the error NFS4ERR_SYMLINK will be returned to the client. If an intermediate component of the pathname provided to OPEN is a symbolic link, the error NFS4ERR_NOTDIR will be returned to the client.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BAD_SEQID
NFS4ERR_DELAY
NFS4ERR_DQUOT
NFS4ERR_EXIST
NFS4ERR_FHEXPIRED
NFS4ERR_GRACE
NFS4ERR_IO
NFS4ERR_ISDIR
NFS4ERR_LEASE_MOVED
NFS4ERR_MOVED
NFS4ERR_NAMETOOLONG
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOSPC
NFS4ERR_NOTDIR
NFS4ERR_NOTSUPP
NFS4ERR_RESOURCE
NFS4ERR_ROFS
NFS4ERR_SERVERFAULT
NFS4ERR_SHARE_DENIED
NFS4ERR_STALE_CLIENTID
NFS4ERR_SYMLINK

14.2.17. Operation 19: OPENATTR - Open Named Attribute Directory

SYNOPSIS

```
(cfh) -> (cfh)
```

ARGUMENT

```
/* CURRENT_FH: file or directory */
void;
```

RESULT

```
struct OPENATTR4res {
        /* CURRENT_FH: name attr directory*/
        nfsstat4        status;
};
```

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 procedures 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.

IMPLEMENTATION

If the server does not support named attributes for the current filehandle, an error of NFS4ERR_NOTSUPP will be returned to the client.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOENT
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTSUPP
NFS4ERR_RESOURCE

NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.18. Operation 20: OPEN_CONFIRM - Confirm Open

SYNOPSIS

```
(cfh), seqid, open_confirm-> stateid
```

ARGUMENT

```
struct OPEN_CONFIRM4args {
        /* CURRENT_FH: opened file */
        seqid4          seqid;
        verifier4       open_confirm;   /* OPEN_CONFIRM verifier */
};
```

RESULT

```
struct OPEN_CONFIRM4resok {
        stateid4        stateid;
};

union OPEN_CONFIRM4res switch (nfsstat4 status) {
 case NFS4_OK:
         OPEN_CONFIRM4resok     resok4;
 default:
         void;
};
```

DESCRIPTION

This operation is used to confirm the sequence id usage for the first time that a nfs_lockowner is used by a client. The OPEN operation returns a opaque confirmation verifier that is then passed to this operation along with the next sequence id for the nfs_lockowner. The sequence id passed to the OPEN_CONFIRM must be 1 (one) greater than the seqid passed to the OPEN operation from which the open_confirm value was obtained. 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.

IMPLEMENTATION

A given client might generate many nfs_lockowner data structures for a given clientid. The client will periodically either dispose of its nfs_lockowners or stop using them for indefinite periods of time. The latter situation is why the NFS version 4 protocol does not have a an explicit operation to exit an nfs_lockowner: 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 nfs_lockowners that have no current lock, open, or delegation state for any files and have not been used recently. The time period used to determine when to dispose of nfs_lockowners 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 nfs_lockowner data structures.

In the case that a client issues an OPEN operation and the server no longer has a record of the nfs_lockowner, the server needs ensure that this is a new OPEN and not a replay or retransmission.

A lazy server implementation might require confirmation for every nfs_lockowner for which it has no record. However, this is not necessary until the server records the fact that it has disposed of one nfs_lockowner for the given clientid.

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 confirmation verifier within the lease period. In this case, the OPEN state on the server goes to confirmed, and the nfs_lockowner on the server is fully established.

Second, the client sends another OPEN request with a sequence id that is incorrect for the nfs_lockowner (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, LIPKEY, or some other flavor that uses cryptography.

What if the server is in the unconfirmed OPEN state for a given nfs_lockowner, and it receives an operation on the nfs_lockowner that has a stateid but the operation is not OPEN, or it is OPEN_CONFIRM but with the wrong confirmation verifier? 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 nfs_lockowner within the lease period. In this case, the OPEN state is cancelled and disposal of the nfs_lockowner can occur.

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_BAD_SEQID
NFS4ERR_EXPIRED
NFS4ERR_FHEXPIRED
NFS4ERR_GRACE
NFS4ERR_INVAL
NFS4ERR_MOVED
NFS4ERR_NOENT
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTSUPP
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.19. Operation 21: OPEN_DOWNGRADE - Reduce Open File Access

SYNOPSIS

(cfh), stateid, seqid, access, deny -> stateid

ARGUMENT

```
struct OPEN_DOWNGRADE4args {
        /* CURRENT_FH: opened file */
        stateid4        stateid;
        seqid4          seqid;
        uint32_t        share_access;
        uint32_t        share_deny;
};
```

RESULT

```
   struct OPEN_DOWNGRADE4resok {
           stateid4        stateid;
   };

   union OPEN_DOWNGRADE4res switch(nfsstat4 status) {
    case NFS4_OK:
           OPEN_DOWNGRADE4resok    resok4;
    default:
           void;
   };
```

This operation is used to adjust the access and deny bits for a given open. This is necessary when a given lockowner opens the same file multiple times with different access and deny flags. In this situation, a close of one of the open's may change the appropriate access and deny flags to remove bits associated with open's no longer in effect.

The access and deny bits specified in this operation replace the current ones for the specified open file. If either the access or the deny mode specified includes bits not in effect for the open, the error NFS4ERR_INVAL should be returned. Since access and deny bits are subsets of those already granted, it is not possible for this request to be denied because of conflicting share reservations.

On success, the current filehandle retains its value.

ERRORS

NFS4ERR_BADHANDLE NFS4ERR_BAD_SEQID NFS4ERR_BAD_STATEID NFS4ERR_EXPIRED NFS4ERR_FHEXPIRED NFS4ERR_INVAL NFS4ERR_MOVED NFS4ERR_NOFILEHANDLE NFS4ERR_OLD_STATEID NFS4ERR_RESOURCE NFS4ERR_SERVERFAULT NFS4ERR_STALE NFS4ERR_STALE_STATEID

14.2.20. Operation 22: PUTFH - Set Current Filehandle

SYNOPSIS

```
      filehandle -> (cfh)
```

ARGUMENT

```
      struct PUTFH4args {
              nfs4_fh         object; };
```

RESULT

```
      struct PUTFH4res {
          /* CURRENT_FH: */
          nfsstat4        status; };
```

DESCRIPTION

Replaces the current filehandle with the filehandle provided as an argument.

IMPLEMENTATION

Commonly used as the first operator in an NFS request to set the context for following operations.

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_FHEXPIRED
NFS4ERR_MOVED
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.21. Operation 23: PUTPUBFH - Set Public Filehandle

SYNOPSIS

```
- -> (cfh)
```

ARGUMENT

```
void;
```

RESULT

```
struct PUTPUBFH4res {
        /* CURRENT_FH: public fh */
        nfsstat4        status;
};
```

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.

IMPLEMENTATION

Used as the first operator in an NFS request to set the context for following operations.

ERRORS

NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_WRONGSEC

14.2.22. Operation 24: PUTROOTFH - Set Root Filehandle

SYNOPSIS

```
- -> (cfh)
```

ARGUMENT

```
void;
```

RESULT

```
struct PUTROOTFH4res {
        /* CURRENT_FH: root fh */
        nfsstat4        status;
};
```

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.

IMPLEMENTATION

Commonly used as the first operator in an NFS request to set the context for following operations.

ERRORS

NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_WRONGSEC

14.2.23. Operation 25: READ - Read from File

SYNOPSIS

```
(cfh), offset, count, stateid -> eof, data
```

ARGUMENT

```
struct READ4args {
        /* CURRENT_FH: file */
        stateid4        stateid;
        offset4         offset;
        count4          count;
};
```

RESULT

```
struct READ4resok {
        bool            eof;
        opaque          data<>;
};

union READ4res switch (nfsstat4 status) {
 case NFS4_OK:
         READ4resok     resok4;
 default:
         void;
};
```

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 record lock or share reservation request. Used by the server to verify that the associated lock is 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.

On success, the current filehandle retains its value.

IMPLEMENTATION

It is possible for the server to return fewer than count bytes of data. If the server returns less than the count requested and eof set to FALSE, the client should issue 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 may back off the transfer size and reduce the read request return. Server resource exhaustion may also occur necessitating a smaller read return.

If the file is locked the server will return an NFS4ERR_LOCKED error. Since the lock may be of short duration, the client may choose to retransmit the READ request (with exponential backoff) until the operation succeeds.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_BAD_STATEID
NFS4ERR_DELAY
NFS4ERR_DENIED
NFS4ERR_EXPIRED
NFS4ERR_FHEXPIRED
NFS4ERR_GRACE
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_LOCKED
NFS4ERR_LEASE_MOVED
NFS4ERR_MOVED

NFS4ERR_NOFILEHANDLE
NFS4ERR_NXIO
NFS4ERR_OLD_STATEID
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_STALE_STATEID
NFS4ERR_WRONGSEC

14.2.24. Operation 26: READDIR - Read Directory

SYNOPSIS

```
(cfh), cookie, cookieverf, dircount, maxcount, attrbits ->
cookieverf { cookie, filename, attrbits, attributes }
```

ARGUMENT

```
struct READDIR4args {
        /* CURRENT_FH: directory */
        nfs_cookie4     cookie;
        verifier4       cookieverf;
        count4          dircount;
        count4          maxcount;
        bitmap4         attr_request;
};
```

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;
       };
```

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.

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 server may return less data.

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 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_READDIR_NOSPC will be returned to the client.

Finally, attrbits 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 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 should not returned.

On success, the current filehandle retains its value.

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.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_BAD_COOKIE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTDIR
NFS4ERR_NOTSUPP
NFS4ERR_READDIR_NOSPC
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_TOOSMALL
NFS4ERR_WRONGSEC

14.2.25. Operation 27: READLINK - Read Symbolic Link

SYNOPSIS

```
(cfh) -> linktext
```

ARGUMENT

```
/* CURRENT_FH: symlink */
void;
```

RESULT

```
struct READLINK4resok {
        linktext4       link;
};

union READLINK4res switch (nfsstat4 status) {
 case NFS4_OK:
         READLINK4resok resok4;
 default:
         void;
};
```

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.

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.

ERRORS

```
NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTSUPP
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC
```

14.2.26. Operation 28: REMOVE - Remove Filesystem Object

SYNOPSIS

```
(cfh), filename -> change_info
```

ARGUMENT

```
struct REMOVE4args {
        /* CURRENT_FH: directory */
        component4       target;
};
```

RESULT

```
struct REMOVE4resok {
        change_info4    cinfo;
}

union REMOVE4res switch (nfsstat4 status) {
 case NFS4_OK:
         REMOVE4resok   resok4;
 default:
         void;
}
```

DESCRIPTION

The REMOVE operation removes (deletes) a directory entry 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 has a length of 0 (zero), or if target does not obey the UTF-8 definition, the error NFS4ERR_INVAL will be returned.

On success, the current filehandle retains its value.

IMPLEMENTATION

NFS versions 2 and 3 required a different operator RMDIR for directory removal. NFS version 4 REMOVE can be used to delete any directory entry independent of its file type.

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 file handle. 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.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NAMETOOLONG
NFS4ERR_NOENT
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTDIR
NFS4ERR_NOTEMPTY
NFS4ERR_NOTSUPP
NFS4ERR_RESOURCE
NFS4ERR_ROFS
NFS4ERR_SERVERFAULT

NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.27. Operation 29: RENAME - Rename Directory Entry

SYNOPSIS

```
(sfh), oldname (cfh), newname -> source_change_info,
target_change_info
```

ARGUMENT

```
struct RENAME4args {
        /* SAVED_FH: source directory */
        component4      oldname;
        /* CURRENT_FH: target directory */
        component4      newname;
};
```

RESULT

```
struct RENAME4resok {
        change_info4    source_cinfo;
        change_info4    target_cinfo;
};

union RENAME4res switch (nfsstat4 status) {
 case NFS4_OK:
         RENAME4resok   resok4;
 default:
         void;
};
```

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. 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 or newname has a length of 0 (zero), or if oldname or newname does not obey the UTF-8 definition, the error NFS4ERR_INVAL will be returned.

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.

A filehandle may or may not become stale or expire on a rename. However, server implementors are strongly encouraged to attempt to keep file handles from becoming stale or expiring in this fashion.

On some servers, the filenames, "." and "..", are illegal as either oldname or newname. In addition, neither oldname nor newname can be an alias for the source directory. These servers will return the error, NFS4ERR_INVAL, in these cases.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_DQUOT
NFS4ERR_EXIST
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_ISDIR
NFS4ERR_MOVED
NFS4ERR_NAMETOOLONG

NFS4ERR_NOENT
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOSPC
NFS4ERR_NOTDIR
NFS4ERR_NOTEMPTY
NFS4ERR_NOTSUPP
NFS4ERR_RESOURCE
NFS4ERR_ROFS
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC
NFS4ERR_XDEV

14.2.28. Operation 30: RENEW - Renew a Lease

SYNOPSIS

```
stateid -> ()
```

ARGUMENT

```
struct RENEW4args {
        stateid4        stateid;
};
```

RESULT

```
struct RENEW4res {
        nfsstat4        status;
};
```

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 client id provided via the SETCLIENTID procedure.

The stateid for RENEW may not be one of the special stateids consisting of all bits 0 (zero) or all bits 1.

IMPLEMENTATION

ERRORS

NFS4ERR_BAD_STATEID
NFS4ERR_EXPIRED

NFS4ERR_GRACE
NFS4ERR_INVAL
NFS4ERR_LEASE_MOVED
NFS4ERR_MOVED
NFS4ERR_OLD_STATEID
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE_STATEID
NFS4ERR_WRONGSEC

14.2.29. Operation 31: RESTOREFH - Restore Saved Filehandle

SYNOPSIS

```
(sfh) -> (cfh)
```

ARGUMENT

```
/* SAVED_FH: */
void;
```

RESULT

```
struct RESTOREFH4res {
        /* CURRENT_FH: value of saved fh */
        nfsstat4        status;
};
```

DESCRIPTION

Set the current filehandle to the value in the saved filehandle. If there is no saved filehandle then return an error NFS4ERR_NOFILEHANDLE.

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)
```

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_FHEXPIRED
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.30. Operation 32: SAVEFH - Save Current Filehandle

SYNOPSIS

```
(cfh) -> (sfh)
```

ARGUMENT

```
/* CURRENT_FH: */
void;
```

RESULT

```
struct SAVEFH4res {
        /* SAVED_FH: value of current fh */
        nfsstat4        status;
};
```

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.

IMPLEMENTATION

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_FHEXPIRED
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE

NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.31. Operation 33: SECINFO - Obtain Available Security

SYNOPSIS

```
(cfh), name -> { secinfo }
```

ARGUMENT

```
struct SECINFO4args {
        /* CURRENT_FH: */
        component4     name;
};
```

RESULT

```
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;
};

struct secinfo4 {
        uint32_t flavor;
        opaque flavor_info<>;   /* null for AUTH_SYS, AUTH_NONE;
                                   contains rpcsec_gss_info for
                                   RPCSEC_GSS. */
};

typedef secinfo4 SECINFO4resok<>;

union SECINFO4res switch (nfsstat4 status) {
 case NFS4_OK:
         SECINFO4resok resok4;
 default:
         void;
};
```

DESCRIPTION

The SECINFO operation is used by the client to obtain a list of valid RPC authentication flavors for a specific file handle, file name pair. The result will contain an array which represents the security mechanisms available. The array entries are represented by the secinfo4 structure. The field 'flavor' will contain a value of AUTH_NONE, AUTH_SYS (as defined in [RFC1831]), 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 [RFC2078]), the quality of protection (as defined in [RFC2078]) 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.

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.

It is recommended that the client issue the SECINFO call protected by a security triple that uses either rpc_gss_svc_integrity or rpc_gss_svc_privacy service. The use of rpc_gss_svc_none would allow an attacker in the middle to modify the SECINFO 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.

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_FHEXPIRED
NFS4ERR_MOVED
NFS4ERR_NAMETOOLONG
NFS4ERR_NOENT
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTDIR
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT

NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.32. Operation 34: SETATTR - Set Attributes

SYNOPSIS

(cfh), attrbits, attrvals -> -

ARGUMENT

```
struct SETATTR4args {
        /* CURRENT_FH: target object */
        stateid4        stateid;
        fattr4          obj_attributes;
};
```

RESULT

```
struct SETATTR4res {
        nfsstat4        status;
        bitmap4         attrsset;
};
```

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 is necessary for SETATTRs that change the size of a file (modify the attribute object_size). This stateid represents a record lock, share reservation, or delegation which must be valid for the SETATTR to modify the file data. A valid stateid would always be specified. When the file size is not changed, the special stateid consisting of all bits 0 (zero) should be used.

On either success or failure of the operation, the server will return the attrsset bitmask to represent what (if any) attributes were successfully set.

On success, the current filehandle retains its value.

IMPLEMENTATION

The file size attribute is used to request changes to the size of a file. A value of 0 (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.

Changing the size of a file with SETATTR indirectly changes the time_modify. A client must account for this as size changes can result in data deletion.

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.

If the server cannot successfully set all the attributes it must return an NFS4ERR_INVAL error. If the server can only support 32 bit offsets and sizes, a SETATTR request to set the size of a file to larger than can be represented in 32 bits will be rejected with this same error.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_BAD_STATEID
NFS4ERR_DELAY
NFS4ERR_DENIED
NFS4ERR_DQUOT
NFS4ERR_EXPIRED
NFS4ERR_FBIG
NFS4ERR_FHEXPIRED
NFS4ERR_GRACE
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOSPC
NFS4ERR_NOTSUPP
NFS4ERR_OLD_STATEID
NFS4ERR_PERM
NFS4ERR_RESOURCE
NFS4ERR_ROFS

```
      NFS4ERR_SERVERFAULT
      NFS4ERR_STALE
      NFS4ERR_STALE_STATEID
      NFS4ERR_WRONGSEC
```

14.2.33. Operation 35: SETCLIENTID - Negotiate Clientid

SYNOPSIS

```
client, callback -> clientid, setclientid_confirm
```

ARGUMENT

```
struct SETCLIENTID4args {
        nfs_client_id4  client;
        cb_client4      callback;
};
```

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;
};
```

DESCRIPTION

The SETCLIENTID operation introduces the ability of the client to notify the server of its intention to use a particular client identifier and verifier pair. Upon successful completion the server will return a clientid which is used in subsequent file locking requests and a confirmation verifier. The client will use the SETCLIENTID_CONFIRM operation to return the verifier to the server. At that point, the client may use the clientid in subsequent operations that require an nfs_lockowner.

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.

IMPLEMENTATION

The server takes the verifier and client identification supplied in the nfs_client_id4 and searches for a match of the client identification. If no match is found the server saves the principal/uid information along with the verifier and client identification and returns a unique clientid that is used as a shorthand reference to the supplied information.

If the server finds matching client identification and a corresponding match in principal/uid, the server releases all locking state for the client and returns a new clientid.

The principal, or principal to user-identifier mapping is taken from the credential presented in the RPC. As mentioned, the server will use the credential and associated principal for the matching with existing clientids. If the client is a traditional host-based client like a Unix NFS client, then the credential presented may be the host credential. If the client is a user level client or lightweight client, the credential used may be the end user’s credential. The client should take care in choosing an appropriate credential since denial of service attacks could be attempted by a rogue client that has access to the credential.

ERRORS

NFS4ERR_CLID_INUSE
NFS4ERR_INVAL
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT

14.2.34. Operation 36: SETCLIENTID_CONFIRM - Confirm Clientid

SYNOPSIS

```
setclientid_confirm -> -
```

ARGUMENT

```
struct SETCLIENTID_CONFIRM4args {
        verifier4       setclientid_confirm;
};
```

RESULT

```
struct SETCLIENTID_CONFIRM4res {
        nfsstat4        status;
};
```

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) opaque confirmation verifier. The server responds with a simple status of success or failure.

IMPLEMENTATION

The client must use the SETCLIENTID_CONFIRM operation to confirm its use of client identifier. If the server is holding state for a client which has presented a new verifier via SETCLIENTID, then the state will not be released, as described in the section "Client Failure and Recovery", until a valid SETCLIENTID_CONFIRM is received. Upon successful confirmation the server will release the previous state held on behalf of the client. The server should choose a confirmation cookie value that is reasonably unique for the client.

ERRORS

NFS4ERR_CLID_INUSE
NFS4ERR_INVAL
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE_CLIENTID

14.2.35. Operation 37: VERIFY - Verify Same Attributes

SYNOPSIS

(cfh), fattr -> -

ARGUMENT

```
struct VERIFY4args {
        /* CURRENT_FH: object */
        fattr4          obj_attributes;
};
```

RESULT

```
struct VERIFY4res {
        nfsstat4        status;
};
```

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.

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_NOTSUPP is returned to the client.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_DELAY
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTSUPP

NFS4ERR_NOT_SAME
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONGSEC

14.2.36. Operation 38: WRITE - Write to File

SYNOPSIS

(cfh), offset, count, stability, stateid, data -> count, committed, verifier

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<>;
};
```

RESULT

```
struct WRITE4resok {
        count4          count;
        stable_how4     committed;
        verifier4       writeverf;
};

union WRITE4res switch (nfsstat4 status) {
 case NFS4_OK:
         WRITE4resok    resok4;
 default:
         void;
};
```

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 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 returned from a previous record lock or share reservation request is provided as part of the argument. The stateid is used by the server to verify that the associated lock is 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, verf. The write verifier is a cookie that the client can use to determine whether the server has changed 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 NFS version 4 protocol service and must be unique between instances of the NFS version 4 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.

On success, the current filehandle retains its value.

IMPLEMENTATION

It is possible for the server to write fewer than count bytes of data. In this case, the server should not return an error unless no data was written at all. If the server writes less than count bytes, 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 NFS version 4 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.

ERRORS

NFS4ERR_ACCES
NFS4ERR_BADHANDLE
NFS4ERR_BAD_STATEID
NFS4ERR_DELAY
NFS4ERR_DENIED
NFS4ERR_DQUOT
NFS4ERR_EXPIRED
NFS4ERR_FBIG
NFS4ERR_FHEXPIRED
NFS4ERR_GRACE

NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_LEASE_MOVED
NFS4ERR_LOCKED
NFS4ERR_MOVED
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOSPC
NFS4ERR_OLD_STATEID
NFS4ERR_RESOURCE
NFS4ERR_ROFS
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_STALE_STATEID
NFS4ERR_WRONGSEC

15. NFS Version 4 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.

15.1. Procedure 0: CB_NULL - No Operation

SYNOPSIS

```
<null>
```

ARGUMENT

```
void;
```

RESULT

```
void;
```

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.

ERRORS

None.

15.2. Procedure 1: CB_COMPOUND - Compound Operations

SYNOPSIS

```
compoundargs -> compoundres
```

ARGUMENT

```
enum nfs_cb_opnum4 {
        OP_CB_GETATTR           = 3,
        OP_CB_RECALL            = 4 };

union nfs_cb_argop4 switch (unsigned argop) {
 case OP_CB_GETATTR:    CB_GETATTR4args opcbgetattr;
 case OP_CB_RECALL:     CB_RECALL4args  opcbrecall; };

struct CB_COMPOUND4args {
        utf8string      tag;
        uint32_t        minorversion;
        nfs_cb_argop4   argarray<>; };
```

RESULT

```
union nfs_cb_resop4 switch (unsigned resop){
 case OP_CB_GETATTR:    CB_GETATTR4res  opcbgetattr;
 case OP_CB_RECALL:     CB_RECALL4res   opcbrecall; };

struct CB_COMPOUND4res {
        nfsstat4 status;
        utf8string      tag;
        nfs_cb_resop4   resarray<>; };
```

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.

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.

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_BAD_STATEID
NFS4ERR_RESOURCE

15.2.1. Operation 3: CB_GETATTR - Get Attributes

SYNOPSIS

fh, attrbits -> attrbits, attrvals

ARGUMENT

```
struct CB_GETATTR4args {
        nfs_fh4 fh;
        bitmap4 attr_request;
};
```

RESULT

```
struct CB_GETATTR4resok {
        fattr4  obj_attributes;
};

union CB_GETATTR4res switch (nfsstat4 status) {
 case NFS4_OK:
         CB_GETATTR4resok       resok4;
 default:
         void;
};
```

DESCRIPTION

The CB_GETATTR operation is used to obtain the attributes modified by an open delegate to allow the server to respond to GETATTR requests for a file which is the subject of an open delegation.

If the handle specified is not one for which the client holds a write open delegation, an NFS4ERR_BADHANDLE error is returned.

IMPLEMENTATION

The client returns attrbits and the associated attribute values only for attributes that it may change (change, time_modify, object_size).

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_RESOURCE

15.2.2. Operation 4: CB_RECALL - Recall an Open Delegation

SYNOPSIS

stateid, truncate, fh -> status

ARGUMENT

```
struct CB_RECALL4args {
        stateid4        stateid;
        bool            truncate;
        nfs_fh4         fh;
};
```

RESULT

```
struct CB_RECALL4res {
        nfsstat4        status;
};
```

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.

IMPLEMENTATION

The client should reply to the callback immediately. Replying does not complete the recall. The recall is not complete until the delegation is returned using a DELEGRETURN.

ERRORS

NFS4ERR_BADHANDLE
NFS4ERR_BAD_STATEID
NFS4ERR_RESOURCE

16. Security Considerations

The major security feature to consider is the authentication of the user making the request of NFS service. Consideration should also be given to the integrity and privacy of this NFS request. These specific issues are discussed as part of the section on "RPC and Security Flavor".

17. IANA Considerations

17.1. Named Attribute Definition

The NFS version 4 protocol provides for the association of named attributes to files. 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 application developer or system vendor is allowed to define the attribute, its semantics, and the associated name. Even though this name space will not be specifically controlled to prevent collisions, the application developer or system vendor is strongly encouraged to provide the name assignment and associated semantics for attributes via an Informational RFC. This will provide for interoperability where common interests exist.

18. RPC definition file

```
   /*
    *  Copyright (C) The Internet Society (1998,1999,2000).
    *  All Rights Reserved.
    */

   /*
    *      nfs4_prot.x
    *
    */

   %#pragma ident  "@(#)nfs4_prot.x        1.97    00/06/12"

   /*
    * Basic typedefs for RFC 1832 data type definitions
    */
   typedef int             int32_t;
   typedef unsigned int    uint32_t;
   typedef hyper           int64_t;
   typedef unsigned hyper  uint64_t;

   /*
    * Sizes
    */
   const NFS4_FHSIZE               = 128;
   const NFS4_VERIFIER_SIZE        = 8;

   /*
    * File types
    */
   enum nfs_ftype4 {
           NF4REG          = 1,    /* Regular File */
           NF4DIR          = 2,    /* Directory */
           NF4BLK          = 3,    /* Special File - block device */
           NF4CHR          = 4,    /* Special File - character device */
           NF4LNK          = 5,    /* Symbolic Link */
           NF4SOCK         = 6,    /* Special File - socket */
           NF4FIFO         = 7,    /* Special File - fifo */
           NF4ATTRDIR      = 8,    /* Attribute Directory */
           NF4NAMEDATTR    = 9     /* Named Attribute */
   };

   /*
    * Error status
    */
   enum nfsstat4 {
           NFS4_OK                 = 0,
           NFS4ERR_PERM            = 1,
           NFS4ERR_NOENT           = 2,
           NFS4ERR_IO              = 5,
           NFS4ERR_NXIO            = 6,
           NFS4ERR_ACCES           = 13,
           NFS4ERR_EXIST           = 17,
           NFS4ERR_XDEV            = 18,
           NFS4ERR_NODEV           = 19,
           NFS4ERR_NOTDIR          = 20,
           NFS4ERR_ISDIR           = 21,
           NFS4ERR_INVAL           = 22,
           NFS4ERR_FBIG            = 27,
           NFS4ERR_NOSPC           = 28,
           NFS4ERR_ROFS            = 30,
           NFS4ERR_MLINK           = 31,
           NFS4ERR_NAMETOOLONG     = 63,
           NFS4ERR_NOTEMPTY        = 66,
           NFS4ERR_DQUOT           = 69,
           NFS4ERR_STALE           = 70,
           NFS4ERR_BADHANDLE       = 10001,
           NFS4ERR_BAD_COOKIE      = 10003,
           NFS4ERR_NOTSUPP         = 10004,
           NFS4ERR_TOOSMALL        = 10005,
           NFS4ERR_SERVERFAULT     = 10006,
           NFS4ERR_BADTYPE         = 10007,
           NFS4ERR_DELAY           = 10008,
           NFS4ERR_SAME            = 10009,/* nverify says attrs same */
           NFS4ERR_DENIED          = 10010,/* lock unavailable        */
           NFS4ERR_EXPIRED         = 10011,/* lock lease expired      */
           NFS4ERR_LOCKED          = 10012,/* I/O failed due to lock  */
           NFS4ERR_GRACE           = 10013,/* in grace period         */
           NFS4ERR_FHEXPIRED       = 10014,/* file handle expired     */
           NFS4ERR_SHARE_DENIED    = 10015,/* share reserve denied    */
           NFS4ERR_WRONGSEC        = 10016,/* wrong security flavor   */
           NFS4ERR_CLID_INUSE      = 10017,/* clientid in use         */
           NFS4ERR_RESOURCE        = 10018,/* resource exhaustion     */
           NFS4ERR_MOVED           = 10019,/* filesystem relocated    */
           NFS4ERR_NOFILEHANDLE    = 10020,/* current FH is not set   */
           NFS4ERR_MINOR_VERS_MISMATCH = 10021,/* minor vers not supp */
           NFS4ERR_STALE_CLIENTID  = 10022,
           NFS4ERR_STALE_STATEID   = 10023,
           NFS4ERR_OLD_STATEID     = 10024,
           NFS4ERR_BAD_STATEID     = 10025,
           NFS4ERR_BAD_SEQID       = 10026,
           NFS4ERR_NOT_SAME        = 10027,/* verify - attrs not same */
           NFS4ERR_LOCK_RANGE      = 10028,
           NFS4ERR_SYMLINK         = 10029,
           NFS4ERR_READDIR_NOSPC   = 10030,
           NFS4ERR_LEASE_MOVED     = 10031
   };

   /*
    * Basic data types
    */
   typedef uint32_t        bitmap4<>;
   typedef uint64_t        offset4;
   typedef uint32_t        count4;
   typedef uint64_t        length4;
   typedef uint64_t        clientid4;
   typedef uint64_t        stateid4;
   typedef uint32_t        seqid4;
   typedef opaque          utf8string<>;
   typedef utf8string      component4;
   typedef component4      pathname4<>;
   typedef uint64_t        nfs_lockid4;
   typedef uint64_t        nfs_cookie4;
   typedef utf8string      linktext4;
   typedef opaque          sec_oid4<>;
   typedef uint32_t        qop4;
   typedef uint32_t        mode4;
   typedef uint64_t        changeid4;
   typedef opaque          verifier4[NFS4_VERIFIER_SIZE];

   /*
    * Timeval
    */
   struct nfstime4 {
           int64_t         seconds;
           uint32_t        nseconds;
   };

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

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

   /*
    * File access handle
    */
   typedef opaque  nfs_fh4<NFS4_FHSIZE>;


   /*
    * File attribute definitions
    */

   /*
    * FSID structure for major/minor
    */
   struct fsid4 {
           uint64_t        major;
           uint64_t        minor;
   };

   /*
    * Filesystem locations attribute for relocation/migration
    */
   struct fs_location4 {
           utf8string      server<>;
           pathname4       rootpath;
   };

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

   /*
    * Various Access Control Entry definitions
    */

   /*
    * Mask that indicates which Access Control Entries are supported.
    * Values for the fattr4_aclsupport attribute.
    */
   const ACL4_SUPPORT_ALLOW_ACL    = 0x00000001;
   const ACL4_SUPPORT_DENY_ACL     = 0x00000002;
   const ACL4_SUPPORT_AUDIT_ACL    = 0x00000004;
   const ACL4_SUPPORT_ALARM_ACL    = 0x00000008;


   typedef uint32_t        acetype4;

   /*
    * acetype4 values, others can be added as needed.
    */
   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;


   /*
    * ACE flag
    */
   typedef uint32_t aceflag4;

   /*
    * ACE flag values
    */
   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;


   /*
    * ACE mask
    */
   typedef uint32_t        acemask4;

   /*
    * ACE mask values
    */
   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;
   /*
    * ACE4_GENERIC_READ -- defined as combination of
    *      ACE4_READ_ACL |
    *      ACE4_READ_DATA |
    *      ACE4_READ_ATTRIBUTES |
    *      ACE4_SYNCHRONIZE
    */

   const ACE4_GENERIC_READ = 0x00120081;

   /*
    * ACE4_GENERIC_WRITE -- defined as combination of
    *      ACE4_READ_ACL |
    *      ACE4_WRITE_DATA |
    *      ACE4_WRITE_ATTRIBUTES |
    *      ACE4_WRITE_ACL |
    *      ACE4_APPEND_DATA |
    *      ACE4_SYNCHRONIZE
    */

   const ACE4_GENERIC_WRITE = 0x00160106;


   /*
    * ACE4_GENERIC_EXECUTE -- defined as combination of
    *      ACE4_READ_ACL
    *      ACE4_READ_ATTRIBUTES
    *      ACE4_EXECUTE
    *      ACE4_SYNCHRONIZE
    */
   const ACE4_GENERIC_EXECUTE = 0x001200A0;


   /*
    * Access Control Entry definition
    */
   struct nfsace4 {
           acetype4        type;
           aceflag4        flag;
           acemask4        access_mask;
           utf8string      who;
   };

   /*
    * Special data/attribute associated with
    * file types NF4BLK and NF4CHR.
    */
   struct specdata4 {
           uint32_t        specdata1;
           uint32_t        specdata2;
   };

   /*
    * Values for fattr4_fh_expire_type
    */
   const   FH4_PERSISTENT          = 0x00000000;
   const   FH4_NOEXPIRE_WITH_OPEN  = 0x00000001;
   const   FH4_VOLATILE_ANY        = 0x00000002;
   const   FH4_VOL_MIGRATION       = 0x00000004;
   const   FH4_VOL_RENAME          = 0x00000008;


   typedef bitmap4         fattr4_supported_attrs;
   typedef nfs_ftype4      fattr4_type;
   typedef uint32_t        fattr4_fh_expire_type;
   typedef changeid4       fattr4_change;
   typedef uint64_t        fattr4_size;
   typedef bool            fattr4_link_support;
   typedef bool            fattr4_symlink_support;
   typedef bool            fattr4_named_attr;
   typedef fsid4           fattr4_fsid;
   typedef bool            fattr4_unique_handles;
   typedef uint32_t        fattr4_lease_time;
   typedef nfsstat4        fattr4_rdattr_error;

   typedef nfsace4         fattr4_acl<>;
   typedef uint32_t        fattr4_aclsupport;
   typedef bool            fattr4_archive;
   typedef bool            fattr4_cansettime;
   typedef bool            fattr4_case_insensitive;
   typedef bool            fattr4_case_preserving;
   typedef bool            fattr4_chown_restricted;
   typedef uint64_t        fattr4_fileid;
   typedef uint64_t        fattr4_files_avail;
   typedef nfs_fh4         fattr4_filehandle;
   typedef uint64_t        fattr4_files_free;
   typedef uint64_t        fattr4_files_total;
   typedef fs_locations4   fattr4_fs_locations;
   typedef bool            fattr4_hidden;
   typedef bool            fattr4_homogeneous;
   typedef uint64_t        fattr4_maxfilesize;
   typedef uint32_t        fattr4_maxlink;
   typedef uint32_t        fattr4_maxname;
   typedef uint64_t        fattr4_maxread;
   typedef uint64_t        fattr4_maxwrite;
   typedef utf8string      fattr4_mimetype;
   typedef mode4           fattr4_mode;
   typedef bool            fattr4_no_trunc;
   typedef uint32_t        fattr4_numlinks;
   typedef utf8string      fattr4_owner;
   typedef utf8string      fattr4_owner_group;
   typedef uint64_t        fattr4_quota_avail_hard;
   typedef uint64_t        fattr4_quota_avail_soft;
   typedef uint64_t        fattr4_quota_used;
   typedef specdata4       fattr4_rawdev;
   typedef uint64_t        fattr4_space_avail;
   typedef uint64_t        fattr4_space_free;
   typedef uint64_t        fattr4_space_total;
   typedef uint64_t        fattr4_space_used;
   typedef bool            fattr4_system;
   typedef nfstime4        fattr4_time_access;
   typedef settime4        fattr4_time_access_set;
   typedef nfstime4        fattr4_time_backup;
   typedef nfstime4        fattr4_time_create;
   typedef nfstime4        fattr4_time_delta;
   typedef nfstime4        fattr4_time_metadata;
   typedef nfstime4        fattr4_time_modify;
   typedef settime4        fattr4_time_modify_set;


   /*
    * Mandatory Attributes
    */
   const FATTR4_SUPPORTED_ATTRS    = 0;
   const FATTR4_TYPE               = 1;
   const FATTR4_FH_EXPIRE_TYPE     = 2;
   const FATTR4_CHANGE             = 3;
   const FATTR4_SIZE               = 4;
   const FATTR4_LINK_SUPPORT       = 5;
   const FATTR4_SYMLINK_SUPPORT    = 6;
   const FATTR4_NAMED_ATTR         = 7;
   const FATTR4_FSID               = 8;
   const FATTR4_UNIQUE_HANDLES     = 9;
   const FATTR4_LEASE_TIME         = 10;
   const FATTR4_RDATTR_ERROR       = 11;

   /*
    * Recommended Attributes
    */
   const FATTR4_ACL                = 12;
   const FATTR4_ACLSUPPORT         = 13;
   const FATTR4_ARCHIVE            = 14;
   const FATTR4_CANSETTIME         = 15;
   const FATTR4_CASE_INSENSITIVE   = 16;
   const FATTR4_CASE_PRESERVING    = 17;
   const FATTR4_CHOWN_RESTRICTED   = 18;
   const FATTR4_FILEHANDLE         = 19;
   const FATTR4_FILEID             = 20;
   const FATTR4_FILES_AVAIL        = 21;
   const FATTR4_FILES_FREE         = 22;
   const FATTR4_FILES_TOTAL        = 23;
   const FATTR4_FS_LOCATIONS       = 24;
   const FATTR4_HIDDEN             = 25;
   const FATTR4_HOMOGENEOUS        = 26;
   const FATTR4_MAXFILESIZE        = 27;
   const FATTR4_MAXLINK            = 28;
   const FATTR4_MAXNAME            = 29;
   const FATTR4_MAXREAD            = 30;
   const FATTR4_MAXWRITE           = 31;
   const FATTR4_MIMETYPE           = 32;
   const FATTR4_MODE               = 33;
   const FATTR4_NO_TRUNC           = 34;
   const FATTR4_NUMLINKS           = 35;
   const FATTR4_OWNER              = 36;
   const FATTR4_OWNER_GROUP        = 37;
   const FATTR4_QUOTA_AVAIL_HARD   = 38;
   const FATTR4_QUOTA_AVAIL_SOFT   = 39;
   const FATTR4_QUOTA_USED         = 40;
   const FATTR4_RAWDEV             = 41;
   const FATTR4_SPACE_AVAIL        = 42;
   const FATTR4_SPACE_FREE         = 43;
   const FATTR4_SPACE_TOTAL        = 44;
   const FATTR4_SPACE_USED         = 45;
   const FATTR4_SYSTEM             = 46;
   const FATTR4_TIME_ACCESS        = 47;
   const FATTR4_TIME_ACCESS_SET    = 48;
   const FATTR4_TIME_BACKUP        = 49;
   const FATTR4_TIME_CREATE        = 50;
   const FATTR4_TIME_DELTA         = 51;
   const FATTR4_TIME_METADATA      = 52;
   const FATTR4_TIME_MODIFY        = 53;
   const FATTR4_TIME_MODIFY_SET    = 54;


   typedef opaque  attrlist4<>;

   /*
    * File attribute container
    */
   struct fattr4 {
           bitmap4         attrmask;
           attrlist4       attr_vals;
   };

   /*
    * Change info for the client
    */
   struct change_info4 {
           bool            atomic;
           changeid4       before;
           changeid4       after;
   };

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

   /*
    * Callback program info as provided by the client
    */
   struct cb_client4 {
           unsigned int    cb_program;
           clientaddr4     cb_location;
   };

   /*
    * Client ID
    */
   struct nfs_client_id4 {
           verifier4       verifier;
           opaque          id<>;
   };

   struct nfs_lockowner4 {
           clientid4       clientid;
           opaque          owner<>;
   };

   enum nfs_lock_type4 {
           READ_LT         = 1,
           WRITE_LT        = 2,
           READW_LT        = 3,    /* blocking read */
           WRITEW_LT       = 4     /* blocking write */
   };

   /*
    * ACCESS: Check access permission
    */
   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;
   };

   struct ACCESS4resok {
           uint32_t        supported;
           uint32_t        access;
   };

   union ACCESS4res switch (nfsstat4 status) {
    case NFS4_OK:
            ACCESS4resok   resok4;
    default:
            void;
   };

   /*
    * CLOSE: Close a file and release share locks
    */
   struct CLOSE4args {
           /* CURRENT_FH: object */
           seqid4          seqid;
           stateid4        stateid;
   };

   union CLOSE4res switch (nfsstat4 status) {
    case NFS4_OK:
            stateid4       stateid;
    default:
            void;
   };

   /*
    * COMMIT: Commit cached data on server to stable storage
    */
   struct COMMIT4args {
           /* CURRENT_FH: file */
           offset4         offset;
           count4          count;
   };
   struct COMMIT4resok {
           verifier4       writeverf;
   };


   union COMMIT4res switch (nfsstat4 status) {
    case NFS4_OK:
            COMMIT4resok   resok4;
    default:
            void;
   };

   /*
    * CREATE: Create a file
    */
   union createtype4 switch (nfs_ftype4 type) {
    case NF4LNK:
            linktext4      linkdata;
    case NF4BLK:
    case NF4CHR:
            specdata4      devdata;
    case NF4SOCK:
    case NF4FIFO:
    case NF4DIR:
            void;
   };

   struct CREATE4args {
           /* CURRENT_FH: directory for creation */
           component4      objname;
           createtype4     objtype;
   };

   struct CREATE4resok {
           change_info4     cinfo;
   };

   union CREATE4res switch (nfsstat4 status) {
    case NFS4_OK:
            CREATE4resok resok4;
    default:
            void;
   };

   /*
    * DELEGPURGE: Purge Delegations Awaiting Recovery
    */
   struct DELEGPURGE4args {
           clientid4       clientid;
   };

   struct DELEGPURGE4res {
           nfsstat4        status;
   };

   /*
    * DELEGRETURN: Return a delegation
    */
   struct DELEGRETURN4args {
           stateid4        stateid;
   };

   struct DELEGRETURN4res {
           nfsstat4        status;
   };

   /*
    * GETATTR: Get file attributes
    */
   struct GETATTR4args {
           /* CURRENT_FH: directory or file */
           bitmap4         attr_request;
   };

   struct GETATTR4resok {
           fattr4          obj_attributes;
   };

   union GETATTR4res switch (nfsstat4 status) {
    case NFS4_OK:
            GETATTR4resok  resok4;
    default:
            void;
   };

   /*
    * GETFH: Get current filehandle
    */
   struct GETFH4resok {
           nfs_fh4         object;
   };

   union GETFH4res switch (nfsstat4 status) {
    case NFS4_OK:
           GETFH4resok     resok4;
    default:
           void;
   };

   /*
    * LINK: Create link to an object
    */
   struct LINK4args {
           /* SAVED_FH: source object */
           /* CURRENT_FH: target directory */
           component4      newname;
   };

   struct LINK4resok {
           change_info4    cinfo;
   };

   union LINK4res switch (nfsstat4 status) {
    case NFS4_OK:
            LINK4resok resok4;
    default:
            void;
   };

   /*
    * LOCK/LOCKT/LOCKU: Record lock management
    */
   struct LOCK4args {
           /* CURRENT_FH: file */
           nfs_lock_type4  locktype;
           seqid4          seqid;
           bool            reclaim;
           stateid4        stateid;
           offset4         offset;
           length4         length;
   };

   struct LOCK4denied {
           nfs_lockowner4  owner;
           offset4         offset;
           length4         length;
   };

   union LOCK4res switch (nfsstat4 status) {
    case NFS4_OK:
            stateid4       stateid;
    case NFS4ERR_DENIED:
            LOCK4denied    denied;
    default:
```

```
                void;
        };

        struct LOCKT4args {
                /* CURRENT_FH: file */
                nfs_lock_type4  locktype;
                nfs_lockowner4  owner;
                offset4         offset;
                length4         length;
        };

        union LOCKT4res switch (nfsstat4 status) {
         case NFS4ERR_DENIED:
                 LOCK4denied    denied;
         case NFS4_OK:
                 void;
         default:
                 void;
        };

        struct LOCKU4args {
                /* CURRENT_FH: file */
                nfs_lock_type4  locktype;
                seqid4          seqid;
                stateid4        stateid;
                offset4         offset;
                length4         length;
        };

        union LOCKU4res switch (nfsstat4 status) {
         case   NFS4_OK:
                 stateid4       stateid;
         default:
                 void;
        };

        /*
         * LOOKUP: Lookup filename
         */
        struct LOOKUP4args {
                /* CURRENT_FH: directory */
                pathname4       path;
        };

        struct LOOKUP4res {
                /* CURRENT_FH: object */
                nfsstat4        status;
        };
```

```
   /*
    * LOOKUPP: Lookup parent directory
    */
   struct LOOKUPP4res {
           /* CURRENT_FH: directory */
           nfsstat4        status;
   };

   /*
    * NVERIFY: Verify attributes different
    */
   struct NVERIFY4args {
           /* CURRENT_FH: object */
           fattr4          obj_attributes;
   };

   struct NVERIFY4res {
           nfsstat4        status;
   };

   /*
    * 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;
   } ;

   /*
    * Share Access and Deny constants for open argument
    */
   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;

   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 {
           pathname4       file;
           stateid4        delegate_stateid;
   };

   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 */
           pathname4       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 */
           uint32_t        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 */
           pathname4       file_delegate_prev;
   };

   /*
    * OPEN: Open a file, potentially receiving an open delegation
    */
   struct OPEN4args {
           open_claim4     claim;
           openflag4       openhow;
           nfs_lockowner4  owner;
           seqid4          seqid;
           uint32_t        share_access;
           uint32_t        share_deny;
   };
   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
    */
   /* Mandatory locking is in effect for this file. */
   const OPEN4_RESULT_MLOCK        = 0x00000001;
   /* Client must confirm open */
   const OPEN4_RESULT_CONFIRM      = 0x00000002;

   struct OPEN4resok {
           stateid4        stateid;        /* Stateid for open */
           change_info4    cinfo;          /* Directory Change Info */
           uint32_t        rflags;         /* Result flags */
           verifier4       open_confirm;   /* OPEN_CONFIRM verifier */
           open_delegation4 delegation;    /* Info on any open
                                              delegation */
   };

   union OPEN4res switch (nfsstat4 status) {
    case NFS4_OK:
           /* CURRENT_FH: opened file */
           OPEN4resok      resok4;
    default:
           void;
   };

   /*
    * OPENATTR: open named attributes directory
    */
   struct OPENATTR4res {
           /* CURRENT_FH: name attr directory*/
           nfsstat4        status;
   };

   /*
    * OPEN_CONFIRM: confirm the open
    */
   struct OPEN_CONFIRM4args {
           /* CURRENT_FH: opened file */
           seqid4          seqid;
           verifier4       open_confirm;   /* OPEN_CONFIRM verifier */
   };

   struct OPEN_CONFIRM4resok {
           stateid4        stateid;
   };

   union OPEN_CONFIRM4res switch (nfsstat4 status) {
    case NFS4_OK:
            OPEN_CONFIRM4resok     resok4;
    default:
            void;
   };

   /*
    * OPEN_DOWNGRADE: downgrade the access/deny for a file
    */
   struct OPEN_DOWNGRADE4args {
           /* CURRENT_FH: opened file */
           stateid4        stateid;
           seqid4          seqid;
           uint32_t        share_access;
           uint32_t        share_deny;
   };

   struct OPEN_DOWNGRADE4resok {
              stateid4        stateid;
   };

   union OPEN_DOWNGRADE4res switch(nfsstat4 status) {
    case NFS4_OK:
           OPEN_DOWNGRADE4resok    resok4;
    default:
            void;
   };

   /*
    * PUTFH: Set current filehandle
    */
   struct PUTFH4args {
           nfs_fh4         object;
   };

   struct PUTFH4res {
           /* CURRENT_FH: */
           nfsstat4        status;
   };

   /*
    * PUTPUBFH: Set public filehandle
    */
   struct PUTPUBFH4res {
           /* CURRENT_FH: public fh */
           nfsstat4        status;
   };

   /*
    * PUTROOTFH: Set root filehandle
    */
   struct PUTROOTFH4res {
           /* CURRENT_FH: root fh */
           nfsstat4        status;
   };

   /*
    * READ: Read from file
    */
   struct READ4args {
           /* CURRENT_FH: file */
           stateid4        stateid;
           offset4         offset;
           count4          count;
   };

   struct READ4resok {
           bool            eof;
           opaque          data<>;
   };

   union READ4res switch (nfsstat4 status) {
    case NFS4_OK:
            READ4resok     resok4;
    default:
            void;
   };

   /*
    * READDIR: Read directory
    */
   struct READDIR4args {
           /* CURRENT_FH: directory */
           nfs_cookie4     cookie;
           verifier4       cookieverf;
           count4          dircount;
           count4          maxcount;
           bitmap4         attr_request;
   };

   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;
   };


   /*
    * READLINK: Read symbolic link
    */
   struct READLINK4resok {
           linktext4       link;
   };

   union READLINK4res switch (nfsstat4 status) {
    case NFS4_OK:
            READLINK4resok resok4;
    default:
            void;
   };

   /*
    * REMOVE: Remove filesystem object
    */
   struct REMOVE4args {
           /* CURRENT_FH: directory */
           component4      target;
   };

   struct REMOVE4resok {
           change_info4    cinfo;
   };

   union REMOVE4res switch (nfsstat4 status) {
    case NFS4_OK:
            REMOVE4resok   resok4;
    default:
            void;
   };

   /*
    * RENAME: Rename directory entry
    */
   struct RENAME4args {
           /* SAVED_FH: source directory */
           component4      oldname;
           /* CURRENT_FH: target directory */
           component4      newname;
   };

   struct RENAME4resok {
           change_info4    source_cinfo;
           change_info4    target_cinfo;
   };

   union RENAME4res switch (nfsstat4 status) {
    case NFS4_OK:
           RENAME4resok    resok4;
    default:
           void;
   };

   /*
    * RENEW: Renew a Lease
    */
   struct RENEW4args {
           stateid4        stateid;
   };

   struct RENEW4res {
           nfsstat4        status;
   };

   /*
    * RESTOREFH: Restore saved filehandle
    */

   struct RESTOREFH4res {
           /* CURRENT_FH: value of saved fh */
           nfsstat4        status;
   };

   /*
    * SAVEFH: Save current filehandle
    */

   struct SAVEFH4res {
           /* SAVED_FH: value of current fh */
           nfsstat4        status;
   };

   /*
    * SECINFO: Obtain Available Security Mechanisms
    */
   struct SECINFO4args {
           /* CURRENT_FH: */
           component4      name;
   };

   /*
    * 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;
   };

   struct secinfo4 {
           uint32_t        flavor;
           /* null for AUTH_SYS, AUTH_NONE;
              contains rpcsec_gss_info for
              RPCSEC_GSS. */
           opaque          flavor_info<>;
   };

   typedef secinfo4 SECINFO4resok<>;

   union SECINFO4res switch (nfsstat4 status) {
    case NFS4_OK:
            SECINFO4resok resok4;
    default:
            void;
   };

   /*
    * SETATTR: Set attributes
    */
   struct SETATTR4args {
           /* CURRENT_FH: target object */
           stateid4        stateid;
           fattr4          obj_attributes;

   };

   struct SETATTR4res {
           nfsstat4        status;
           bitmap4         attrsset;
   };

   /*
    * SETCLIENTID
    */
   struct SETCLIENTID4args {
           nfs_client_id4  client;
           cb_client4      callback;
   };

   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;
   };

   struct SETCLIENTID_CONFIRM4args {
           verifier4       setclientid_confirm;
   };

   struct SETCLIENTID_CONFIRM4res {
           nfsstat4        status;
   };

   /*
    * VERIFY: Verify attributes same
    */
   struct VERIFY4args {
           /* CURRENT_FH: object */
           fattr4          obj_attributes;
   };

   struct VERIFY4res {
           nfsstat4        status;
   };

   /*
    * WRITE: Write to file
    */
   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<>;
   };

   struct WRITE4resok {
           count4          count;
           stable_how4     committed;
           verifier4       writeverf;
   };

   union WRITE4res switch (nfsstat4 status) {
    case NFS4_OK:
            WRITE4resok    resok4;
    default:
            void;
   };

   /*
    * Operation arrays
    */

   enum nfs_opnum4 {
           OP_ACCESS               = 3,
           OP_CLOSE                = 4,
           OP_COMMIT               = 5,
           OP_CREATE               = 6,
           OP_DELEGPURGE           = 7,
           OP_DELEGRETURN          = 8,
           OP_GETATTR              = 9,
           OP_GETFH                = 10,
           OP_LINK                 = 11,
           OP_LOCK                 = 12,
           OP_LOCKT                = 13,
           OP_LOCKU                = 14,
           OP_LOOKUP               = 15,
           OP_LOOKUPP              = 16,
           OP_NVERIFY              = 17,
           OP_OPEN                 = 18,
           OP_OPENATTR             = 19,
           OP_OPEN_CONFIRM         = 20,
           OP_OPEN_DOWNGRADE       = 21,
           OP_PUTFH                = 22,
           OP_PUTPUBFH             = 23,
           OP_PUTROOTFH            = 24,
           OP_READ                 = 25,
           OP_READDIR              = 26,
           OP_READLINK             = 27,
           OP_REMOVE               = 28,
           OP_RENAME               = 29,
           OP_RENEW                = 30,
           OP_RESTOREFH            = 31,
           OP_SAVEFH               = 32,
           OP_SECINFO              = 33,
           OP_SETATTR              = 34,
           OP_SETCLIENTID          = 35,
           OP_SETCLIENTID_CONFIRM  = 36,
           OP_VERIFY               = 37,
           OP_WRITE                = 38
   };

   union nfs_argop4 switch (nfs_opnum4 argop) {
    case OP_ACCESS:        ACCESS4args opaccess;
    case OP_CLOSE:         CLOSE4args opclose;
    case OP_COMMIT:        COMMIT4args opcommit;
    case OP_CREATE:        CREATE4args opcreate;
    case OP_DELEGPURGE:    DELEGPURGE4args opdelegpurge;
    case OP_DELEGRETURN:   DELEGRETURN4args opdelegreturn;
    case OP_GETATTR:       GETATTR4args opgetattr;
    case OP_GETFH:         void;
    case OP_LINK:          LINK4args oplink;
    case OP_LOCK:          LOCK4args oplock;
    case OP_LOCKT:         LOCKT4args oplockt;
    case OP_LOCKU:         LOCKU4args oplocku;
    case OP_LOOKUP:        LOOKUP4args oplookup;
    case OP_LOOKUPP:       void;
    case OP_NVERIFY:       NVERIFY4args opnverify;
    case OP_OPEN:          OPEN4args opopen;
    case OP_OPENATTR:      void;
    case OP_OPEN_CONFIRM:  OPEN_CONFIRM4args opopen_confirm;
    case OP_OPEN_DOWNGRADE:        OPEN_DOWNGRADE4args opopen_downgrade;
    case OP_PUTFH:         PUTFH4args opputfh;
    case OP_PUTPUBFH:      void;
    case OP_PUTROOTFH:     void;
    case OP_READ:          READ4args opread;
    case OP_READDIR:       READDIR4args opreaddir;
    case OP_READLINK:      void;
```

```
     case OP_REMOVE:        REMOVE4args opremove;
     case OP_RENAME:        RENAME4args oprename;
     case OP_RENEW:         RENEW4args oprenew;
     case OP_RESTOREFH:     void;
     case OP_SAVEFH:        void;
     case OP_SECINFO:       SECINFO4args opsecinfo;
     case OP_SETATTR:       SETATTR4args opsetattr;
     case OP_SETCLIENTID:   SETCLIENTID4args opsetclientid;
     case OP_SETCLIENTID_CONFIRM:   SETCLIENTID_CONFIRM4args
                                            opsetclientid_confirm;
     case OP_VERIFY:        VERIFY4args opverify;
     case OP_WRITE:         WRITE4args opwrite;
    };

    union nfs_resop4 switch (nfs_opnum4 resop){
     case OP_ACCESS:        ACCESS4res opaccess;
     case OP_CLOSE:         CLOSE4res opclose;
     case OP_COMMIT:        COMMIT4res opcommit;
     case OP_CREATE:        CREATE4res opcreate;
     case OP_DELEGPURGE:    DELEGPURGE4res opdelegpurge;
     case OP_DELEGRETURN:   DELEGRETURN4res opdelegreturn;
     case OP_GETATTR:       GETATTR4res opgetattr;
     case OP_GETFH:         GETFH4res opgetfh;
     case OP_LINK:          LINK4res oplink;
     case OP_LOCK:          LOCK4res oplock;
     case OP_LOCKT:         LOCKT4res oplockt;
     case OP_LOCKU:         LOCKU4res oplocku;
     case OP_LOOKUP:        LOOKUP4res oplookup;
     case OP_LOOKUPP:       LOOKUPP4res oplookupp;
     case OP_NVERIFY:       NVERIFY4res opnverify;
     case OP_OPEN:          OPEN4res opopen;
     case OP_OPENATTR:      OPENATTR4res opopenattr;
     case OP_OPEN_CONFIRM:  OPEN_CONFIRM4res opopen_confirm;
     case OP_OPEN_DOWNGRADE:        OPEN_DOWNGRADE4res opopen_downgrade;
     case OP_PUTFH:         PUTFH4res opputfh;
     case OP_PUTPUBFH:      PUTPUBFH4res opputpubfh;
     case OP_PUTROOTFH:     PUTROOTFH4res opputrootfh;
     case OP_READ:          READ4res opread;
     case OP_READDIR:       READDIR4res opreaddir;
     case OP_READLINK:      READLINK4res opreadlink;
     case OP_REMOVE:        REMOVE4res opremove;
     case OP_RENAME:        RENAME4res oprename;
     case OP_RENEW:         RENEW4res oprenew;
     case OP_RESTOREFH:     RESTOREFH4res oprestorefh;
     case OP_SAVEFH:        SAVEFH4res opsavefh;
     case OP_SECINFO:       SECINFO4res opsecinfo;
     case OP_SETATTR:       SETATTR4res opsetattr;
     case OP_SETCLIENTID:   SETCLIENTID4res opsetclientid;
     case OP_SETCLIENTID_CONFIRM:   SETCLIENTID_CONFIRM4res
                                            opsetclientid_confirm;
     case OP_VERIFY:        VERIFY4res opverify;
     case OP_WRITE:         WRITE4res opwrite;
    };

    struct COMPOUND4args {
            utf8string      tag;
            uint32_t        minorversion;
            nfs_argop4      argarray<>;
    };

    struct COMPOUND4res {
            nfsstat4 status;
            utf8string      tag;
            nfs_resop4      resarray<>;
    };

    /*
     * Remote file service routines
     */
    program NFS4_PROGRAM {
            version NFS_V4 {
                    void
                            NFSPROC4_NULL(void) = 0;

                    COMPOUND4res
                            NFSPROC4_COMPOUND(COMPOUND4args) = 1;

            } = 4;
    } = 100003;



    /*
     * NFS4 Callback Procedure Definitions and Program
     */

    /*
     * CB_GETATTR: Get Current Attributes
     */
    struct CB_GETATTR4args {
            nfs_fh4 fh;
            bitmap4 attr_request;
    };

    struct CB_GETATTR4resok {
            fattr4  obj_attributes;
    };

    union CB_GETATTR4res switch (nfsstat4 status) {
     case NFS4_OK:
             CB_GETATTR4resok       resok4;
     default:
             void;
    };

    /*
     * CB_RECALL: Recall an Open Delegation
     */
    struct CB_RECALL4args {
            stateid4        stateid;
            bool            truncate;
            nfs_fh4         fh;
    };

    struct CB_RECALL4res {
            nfsstat4        status;
    };

    /*
     * Various definitions for CB_COMPOUND
     */
    enum nfs_cb_opnum4 {
            OP_CB_GETATTR            = 3,
            OP_CB_RECALL             = 4
    };

    union nfs_cb_argop4 switch (unsigned argop) {
     case OP_CB_GETATTR:    CB_GETATTR4args opcbgetattr;
     case OP_CB_RECALL:     CB_RECALL4args  opcbrecall;
    };

    union nfs_cb_resop4 switch (unsigned resop){
     case OP_CB_GETATTR:    CB_GETATTR4res  opcbgetattr;
     case OP_CB_RECALL:     CB_RECALL4res   opcbrecall;
    };

    struct CB_COMPOUND4args {
            utf8string      tag;
            uint32_t        minorversion;
            nfs_cb_argop4   argarray<>;
    };

    struct CB_COMPOUND4res {
            nfsstat4 status;
            utf8string      tag;
            nfs_cb_resop4   resarray<>;
    };


    /*
     * Program number is in the transient range since the client
     * will assign the exact transient program number and provide
     * that to the server via the SETCLIENTID operation.
     */
    program NFS4_CALLBACK {
            version NFS_CB {
                    void
                            CB_NULL(void) = 0;
                    CB_COMPOUND4res
                            CB_COMPOUND(CB_COMPOUND4args) = 1;
            } = 1;
    } = 40000000;
```

19. Bibliography

[Floyd] S. Floyd, V. Jacobson, "The Synchronization of Periodic Routing Messages," IEEE/ACM Transactions on Networking, 2(2), pp. 122-136, April 1994.

[Gray] C. Gray, D. Cheriton, "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," Proceedings of the Twelfth Symposium on Operating Systems Principles, p. 202-210, December 1989.

[ISO10646] "ISO/IEC 10646-1:1993. International Standard -- Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane."

[Juszczak] Juszczak, Chet, "Improving the Performance and Correctness of an NFS Server," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, June 1990, pages 53-63. Describes reply cache implementation that avoids work in the server by handling duplicate requests. More important, though listed as a side-effect, the reply cache aids in the avoidance of destructive non-idempotent operation re-application -- improving correctness.

[Kazar] Kazar, Michael Leon, "Synchronization and Caching Issues in the Andrew File System," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Dallas Winter 1988, pages 27-36. A description of the cache consistency scheme in AFS. Contrasted with other distributed file systems.

[Macklem] Macklem, Rick, "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol," Winter USENIX Conference Proceedings, USENIX Association, Berkeley, CA, January 1991. Describes performance work in tuning the 4.3BSD Reno NFS implementation. Describes performance improvement (reduced CPU loading) through elimination of data copies.

[Mogul] Mogul, Jeffrey C., "A Recovery Protocol for Spritely NFS," USENIX File System Workshop Proceedings, Ann Arbor, MI, USENIX Association, Berkeley, CA, May 1992. Second paper on Spritely NFS proposes a lease-based scheme for recovering state of consistency protocol.

[Nowicki] Nowicki, Bill, "Transport Issues in the Network File System," ACM SIGCOMM newsletter Computer Communication Review, April 1989. A brief description of the basis for the dynamic retransmission work.

[Pawlowski] Pawlowski, Brian, Ron Hixon, Mark Stein, Joseph Tumminaro, "Network Computing in the UNIX and IBM Mainframe Environment," Uniforum ‘89 Conf. Proc., (1989) Description of an NFS server implementation for IBM’s MVS operating system.

[RFC1094] Sun Microsystems, Inc., "NFS: Network File System Protocol Specification", RFC 1094, March 1989.

[RFC1345] Simonsen, K., "Character Mnemonics & Character Sets", RFC 1345, June 1992.

[RFC1700] Reynolds, J. and J. Postel, "Assigned Numbers", STD 2, RFC 1700, October 1994.

[RFC1813] Callaghan, B., Pawlowski, B. and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, June 1995.

[RFC1831] Srinivasan, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 1831, August 1995.

[RFC1832] Srinivasan, R., "XDR: External Data Representation Standard", RFC 1832, August 1995.

[RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, August 1995.

[RFC2025] Adams, C., "The Simple Public-Key GSS-API Mechanism (SPKM)", RFC 2025, October 1996.

[RFC2054] Callaghan, B., "WebNFS Client Specification", RFC 2054, October 1996.

[RFC2055] Callaghan, B., "WebNFS Server Specification", RFC 2055, October 1996.

[RFC2078] Linn, J., "Generic Security Service Application Program Interface, Version 2", RFC 2078, January 1997.

[RFC2152] Goldsmith, D., "UTF-7 A Mail-Safe Transformation Format of Unicode", RFC 2152, May 1997.

[RFC2203] Eisler, M., Chiu, A. and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, August 1995.

[RFC2277] Alvestrand, H., "IETF Policy on Character Sets and Languages", BCP 18, RFC 2277, January 1998.

[RFC2279] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

[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.

[RFC2847] Eisler, M., "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM", RFC 2847, June 2000.

[Sandberg] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Summer 1985. The basic paper describing the SunOS implementation of the NFS version 2 protocol, and discusses the goals, protocol specification and trade-offs.

[Srinivasan] Srinivasan, V., Jeffrey C. Mogul, "Spritely NFS: Implementation and Performance of Cache Consistency Protocols", WRL Research Report 89/5, Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Ave., Palo Alto, CA, 94301, May 1989. This paper analyzes the effect of applying a Sprite-like consistency protocol applied to standard NFS. The issues of recovery in a stateful environment are covered in [Mogul].

[Unicode1] The Unicode Consortium, "The Unicode Standard, Version 3.0", Addison-Wesley Developers Press, Reading, MA, 2000. ISBN 0-201-61633-5.
More information available at: http://www.unicode.org/

[Unicode2] "Unsupported Scripts" Unicode, Inc., The Unicode Consortium, P.O. Box 700519, San Jose, CA 95710-0519 USA, September 1999
http://www.unicode.org/unicode/standard/unsupported.html

[XNFS] The Open Group, Protocols for Interworking: XNFS, Version 3W, The Open Group, 1010 El Camino Real Suite 380, Menlo Park, CA 94025, ISBN 1-85912-184-5, February 1998.
HTML version available: http://www.opengroup.org

20. Authors

20.1. Editor’s Address

Spencer Shepler
Sun Microsystems, Inc.
7808 Moonflower Drive
Austin, Texas 78750

Phone: +1 512-349-9376
EMail: spencer.shepler@sun.com

20.2. Authors’ Addresses

Carl Beame
Hummingbird Ltd.

EMail: beame@bws.com

Brent Callaghan
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

Phone: +1 650-786-5067
EMail: brent.callaghan@sun.com

Mike Eisler
5565 Wilson Road
Colorado Springs, CO 80919

Phone: +1 719-599-9026
EMail: mike@eisler.com

David Noveck
Network Appliance
375 Totten Pond Road
Waltham, MA 02451

Phone: +1 781-895-4949
E-mail: dnoveck@netapp.com

David Robinson
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

Phone: +1 650-786-5088
EMail: david.robinson@sun.com

Robert Thurlow
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

Phone: +1 650-786-5096
EMail: robert.thurlow@sun.com

20.3. Acknowledgements

The author thanks and acknowledges:

Neil Brown for his extensive review and comments of various drafts.

21. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.