NFS Version 4 Working Group
INTERNET-DRAFT
Document: draft-ietf-nfsv4-00.txt

S. Shepler
B. Callaghan
M. Eisler
D. Robinson
R. Thurlow
Sun Microsystems
February 1999

NFS version 4

Status of this Memo

Abstract

NFS version 4 is a distributed file system protocol which owes heritage to NFS versions 2 [RFC1094] and 3 [RFC1813]. Unlike earlier versions, NFS version 4 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, and internationlization have been added. Of course, attention has been applied to making NFS version 4 operate well in an Internet environment.

Copyright

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

NFS version 4 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: stateless 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 NFS version 4 provides a mechanism to allow clients and servers to negotiate security and require clients and servers to support a minimal set of security schemes.

- Good cross-platform interoperability.

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

- Designed for protocol extensions.

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

2. 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 to NFS version 4.

2.1. Ports and Transports

Historically, NFS version 2 and version 3 servers have resided on UDP/TCP port 2049. Port 2049 is a IANA registered port number for NFS and therefore will continue to be used for NFS version 4. Using the well known 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 NFS server SHOULD offer its RPC service via TCP as the primary transport. The server SHOULD also provide UDP for RPC service. The NFS client SHOULD also have a preference for TCP usage but may supply a mechanism to override TCP in favor of UDP as the RPC transport.

2.2. 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. The flavors AUTH_NONE, AUTH_SYS, and AUTH_DH MAY be implemented as well.

2.2.1. 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.

2.2.1.1. Kerberos V5 as security triple

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

columns:

```
 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    rpc_gss_svc_privacy
                                   for integrity,
                                   and 56 bit DES
                                   for privacy.
```

This section will be expanded to include the pertinent details from draft-ietf-nfsv4-nfssec-00.txt.

2.2.1.2. <another security triple>

Another GSS-API mechanism will need to be specified here along with the corresponding security triple(s).

2.3. Security Negotiation

With the NFS version 4 server potentially offering multiple security mechanisms, the client will need a way 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/desired.

2.3.1. Security Error

Based on the assumption that each NFS version 4 client and server must support a minimum set of security (i.e. Kerberos-V5 under RPCSEC_GSS, <ed: add other>), 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.

2.3.2. SECINFO

The new procedure SECINFO (see SECINFO procedure definition) 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 could happen that the server's policies change during the client's interaction therefore forcing the client to negotiate a new security triple.

3. File handles

The file handle in the NFS protocol is an opaque identifier for a file system object. The server is responsible for translating the file handle to its internal representation of the file system object. The file handle is used to uniquely identify a file system object at the NFS server. The client should be able to depend on the fact that a file handle will not be reused once a file system object has been destroyed. If the file handle is reused, the time elapsed before reuse SHOULD be very significant. Note that each NFS procedure is defined in terms of its file handle(s) except for the NULL procedure.

3.1. Obtaining the first file handle

If each of the meaningful operations of the NFS protocol require a file handle, the client must have a mechanism to obtain the first file handle. With NFS version 2 [RFC1094] and NFS version 3 [RFC1813], there exists an ancillary, protocol to obtain the first file handle. The MOUNT protocol, RPC program number 100005, provides the mechanism of translating a string based file system path name to a file handle which can then be used by the NFS protocols.

The MOUNT protocol as currently implemented has deficiencies in the area of security and use via firewalls. This is one reason that the use of the public file handle was introduced [RFC2054] [RFC2055]. The public file handle is a special case file handle that is used in combination with a path name to avoid using the MOUNT protocol for obtaining the first file handle. With the introduction and use of the public file handle in the previous versions of NFS, it has been shown that the MOUNT protocol is unnecessary for viable interaction between the client and server with the use of file handles.

3.2. The persistent and volatile file handle

For the first time in NFS version 4, the file handle constructed by the server can be volatile. In the previous versions of NFS, the server was responsible for ensuring the persistence of the file handle. This meant that as long as a file system object remained in existence at the server the file handle for that object had to be the same each time the client asked for it. This persistent quality eased the implementation at the client in the event of server restart or failure and recovery. For some servers, fulfilling the persistent requirement has been straight forward; for others it has been difficult and affected at best performance and at worst correctness.

The existence of the volatile file handle requires the client to implement a method of recovering from the expiration of a file handle. Most commonly the client will need to store the component

names associated with the file system object in question. With these names, the client will be able to recover by finding a file handle in the name space that is still available or by starting at the root of the server's file system name space.

The use of a volatile file handle provides these advantages:

- Eases the server implementation requirements.
- Server can provide extended services more easily with the use of volatile file handles (HSM software, file system reorganization)
- Allows for server file systems that have difficulty in mapping a stable file handle to a file object. In this case, a server implementation would be able to build a mapping dynamically between a volatile file handle and the file system object.

In some cases a file handle 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 file handle is volatile, it may have expired, requiring the client to get a new file handle). Thus the server needs to be able to return NFS4ERR_STALE in the former case, and NFS4ERR_EXPIRED in the latter case. This can be done by careful construction of the volatile file handle. One implementation that has been suggested is the following. A volatile file handle, while opaque to the client could contain:

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

slot is an index in the server volatile file handle 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_EXPIRED. If slot is out of range, return NFS4ERR_EXPIRED. If the generation number does not match, return NFS4ERR_EXPIRED.

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

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

4. Basic Data Types

Arguments and results from operations will be described in terms of basic XDR types defined in [RFC1832]. The following data types will be defined in terms of basic XDR types:

filehandle: opaque <128>

An NFS version 4 filehandle. A filehandle with zero length is recognized as a "public" filehandle.

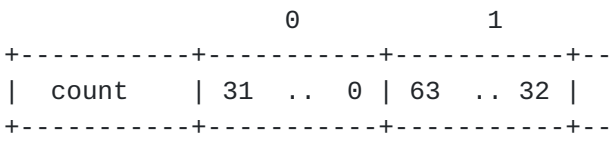
utf8string: opaque <>

A counted array of octets that contains a UTF-8 string.

Note: Section 11, Internationalization, covers the rational of using UTF-8.

bitmap: uint32 <>

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).



createverf: opaque<8>

Verify used for exclusive create semantics

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 local 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.

specdata4

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

This data type represents additional information for the device file types NFCHR and NFBLK.

5. File Attributes

To meet the NFS Version 4 requirements of extensibility and increased interoperability with non-Unix platforms, attributes must be handled in a more flexible manner. The NFS Version 3 fattr3 structure contained a fixed list of attributes that not all clients and servers are able to support or care about, which cannot be extended as new needs crop up, and which provides no way to indicate non-support. With NFS Version 4, 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 extended. Both mandatory and recommended attributes are supported in the NFS V4 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 response. New mandatory or recommended attributes may be added to the NFS protocol between revisions by publishing a standards-track RFC which allocates a new attribute number value and defines the encoding for the attribute.

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

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

Extended attributes are intended primarily for data needed by applications rather than by an NFS client implementation per se; NFS implementors are strongly encouraged to define their new attributes as recommended attributes by bringing them to the working group.

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, though some operations may be impaired or limited in some ways in this case. 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, though 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 be able to deal with not receiving 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 in general it seems that the client has a better ability to fake data or do without.

Most attributes from NFS V3's FSINFO, FSSTAT and PATHCONF procedures have been added as recommended attributes, so that filesystem info may be collected via the filehandle of any object the filesystem. This renders those procedures unnecessary in NFS V4. If a server supports any per-filesystem attributes, it must support the fsid attribute so that the client may always determine when filesystems are crossed so that it can work correctly with these attributes.

5.3. Extended 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 filesystem object. The namespace for these attributes may be accessed by using the OPENATTR operation to get a filehandle for a virtual "attribute directory" and using READDIR and LOOKUP operations on this filehandle. Extended attributes may then be examined or changed by normal READ and WRITE and CREATE operations on the filehandles returned from READDIR and LOOKUP. Attributes may have attributes, for example, a security label may have access control information in its own right.

It is recommended that servers support arbitrary extended attributes. A client should not depend on the ability to store any extended attributes in the server's filesystem. If a server does support extended 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 there should be no attribute names which will be considered illegal by the server.

Names of attributes will not be controlled by a standards body, however vendors and application writers are encouraged to register attribute names and the interpretation and semantics of the stream of bytes via informational RFC so that vendors may interoperate where common interests exist.

The following is a list of mandatory and recommended attributes.

5.4. Mandatory Attributes - Definitions

Name:	supp_attr
Data Type:	nfs_attrvec4
Access:	Read
Description:	the bit vector which would retrieve all mandatory and recommended attributes which may be requested for this object
Justification:	the client must ask this question to request correct attributes

Name: object_type

Data Type: nfs_type4

Access: Read

Description: the type of the object (file/directory/symlink)

Justification: the client cannot handle object correctly without type

Name: object_size

Data Type: uint64

Access: Read Write

Description: the size of the object in bytes

Justification: could be very expensive to derive, likely to be available

Name: change

Data Type: uint64

Description: A value created by the server that the client can use to determine if a file data, directory contents or attributes have been modified. The server can just return the file mtime in this field though if a more precise value exists then it can be substituted, for instance, a checksum or sequence number.

Justification: necessary for any useful caching, likely to be available

Name: persistent_fh

Data Type: boolean

Access: Read

Description: is the filehandle for this object persistent?

Justification: Server should know if the file handles being provided are persistent or not. If the server is not able to make this determination, then it can choose volatile or non-persistent.

Name: extended

Data Type: boolean

Access: Read

Description: whether or not this object has extended attributes

Justification:

Name: link_support

Data Type: boolean

Access: Read

Description: whether or not this object's filesystem supports hard links

Justification: Server can easily determine if links are supported

Name: symlink_support

Data Type: boolean

Access: Read

Description: whether or not this object's filesystem supports symbolic links

Justification: Server can easily determine if links are supported

5.5. Recommended Attributes - Definitions

Name: owner

Data Type: utf8<>

Access: Read Write

Description: the string name of the owner of this object; note that the concept of a numeric uid has been dropped

Name: group_owner

Data Type: utf8<>

Access: Read Write

Description: the string name of the group of the owner of this object; note that the concept of a numeric gid has been dropped

Name: file_id

Data Type: fileid4

Access: Read

Description: a number uniquely identifying the file within the filesystem

Name: file_name

Data Type: utf8<>

Access: Read

Description: the name of this object (primarily for readdir requests)

Name: filehandle

Data Type: nfs_fh4

Access: Read

Description: the filehandle of this object (primarily for readdir requests)

Name: ACL

Data Type: nfsacl4

Access: Read Write

Description: the access control list for the object [The nature and format of ACLs is still to be determined.]

Name: mode

Data Type: uint32

Access: Read Write

Description: Unix-style permission bits for this object (deprecated in favor of ACLs)

Name: object_links

Data Type: uint32

Access: Read

Description: number of links to this object

Name: space_used

Data Type: uint64

Access: Read

Description: number of filesystem bytes allocated to this object

Name: fsid.major

Data Type: uint64

Access: Read

Description: unique filesystem identifier for the filesystem holding this object

Name: fsid.minor

Data Type: uint64

Access: Read

Description: unique filesystem identifier within the fsid.major filesystem identifier for the filesystem holding this object

Name: quota_used

Data Type: uint64

Access: Read

Description: number of bytes of disk space occupied by the owner of this object on this filesystem

Name: quota_soft

Data Type: uint64

Access: Read

Description: number of bytes of disk space at which the client may choose to warn the user about limited space

Name: quota_hard

Data Type: uint64

Access: Read

Description: number of bytes of disk space beyond which the server will decline to allocate new space

Name: rawdev

Data Type: specdata4

Access: Read

Description: raw device identifier

Name: access_time

Data Type: nfstime4

Access: Read Write

Description: the time of last access to the object

Name: create_time

Data Type: nfstime4

Access: Read Write

Description: the time of creation of the object. This attribute does not have any relation to the traditional Unix file attribute 'ctime' or 'change time'.

Name: meta-data_time

Data Type: nfstime4

Access: Read Write

Description: the time of last meta-data modification of the object.

Name: mod_time

Data Type: nfstime4

Access: Read Write

Description: the time since the epoch of last modification to the object

Name: backup_time

Data Type: nfstime4

Access: Read Write

Description: the time of last backup of the object

Name: mime_type

Data Type: utf8<>

Access: Read Write

Description: MIME body type/subtype of this object

Name: version

Data Type: utf8<>

Access: Read Write

Description: version number of this document

Name: hidden

Data Type: boolean

Access: Read Write

Description: whether or not this file is considered hidden

Name: archive

Data Type: boolean

Access: Read Write

Description: whether or not this file has been archived since the time of last modification (deprecated in favor of backup_time)

Name: system

Data Type: boolean

Access: Read Write

Description: whether or not this file is a system file

Name: homogeneous

Data Type: boolean

Access: Read

Description: whether or not this object's filesystem is homogeneous, i.e. whether pathconf is the same for all filesystem objects

Name: cansettime

Data Type: boolean

Access: Read

Description: whether or not this object's filesystem can fill in the times on a SETATTR request without an explicit time

Name: no_trunc

Data Type: boolean

Access: Read

Description: if a name longer than name_max is used, will an error be returned or will the name be truncated?

Name: chown_restricted

Data Type: boolean

Access: Read

Description: will a request to change ownership be honored?

Name: case_insensitive

Data Type: boolean

Access: Read

Description: are filename comparisons on this filesystem case insensitive?

Name: case_preserving

Data Type: boolean

Access: Read

Description: is filename case on this filesystem preserved?

Name: name_max

Data Type: uint32

Access: Read

Description: maximum filename size supported for this object

Name: link_max

Data Type: uint32

Access: Read

Description: maximum number of links for this object

Name: read_max

Data Type: uint64

Access: Read

Description: maximum read size supported for this object

Name: write_max

Data Type: uint64

Access: Read

Description: 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.

Name: maxfilesize

Data Type: uint64

Access: Read

Description: maximum supported file size for the filesystem of this object

Name: time_delta

Data Type: nfstime4

Access: Read

Description: smallest useful server time granularity

Name: total_space

Data Type: uint64

Access: Read

Description: total disk space in bytes on the filesystem containing this object

Name: free_space

Data Type: uint64

Access: Read

Description: free disk space in bytes on the filesystem containing this object - this should be the smallest relevant limit

Name: avail_space

Data Type: uint64

Access: Read

Description: disk space in bytes available to this user on the filesystem containing this object - this should be

the smallest relevant limit

Name: total_files

Data Type: uint64

Access: Read

Description: total file slots on the filesystem containing this object

Name: free_files

Data Type: uint64

Access: Read

Description: free file slots on the filesystem containing this object - this should be the smallest relevant limit

Name: avail_files

Data Type: uint64

Access: Read

Description: file slots available to this user on the filesystem containing this object - this should be the smallest relevant limit

Name: volatility

Data Type: nfstime4

Access: Read

Description: approximate time until next expected change on this filesystem, as a measure of volatility

6. NFS Server Namespace

6.1. Server Exports

On a UNIX server the name-space describes all the files reachable by pathnames under the root directory "/". 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. Typically, pieces of the name-space are made available via an "export" feature. 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.

6.2. 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 NFS version 2 and 3 protocols. The client expects all LOOKUP operations to remain within a single server file-system, i.e. 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.

6.3. 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 only of exported directories. The pseudo file-system has a unique fsid and behaves like a normal, read-only file-system.

6.4. Multiple Roots

DOS, Windows 95, 98 and NT 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.

6.5. 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 NFS version 4 is first instantiated. It is expected 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 file-handles are volatile. This can be confirmed by checking the associated "persistent_fh" attribute for those filehandles in question. If the filehandles are volatile, the NFS client must be prepared to recover a filehandle value (i.e. with a v4 multi-component LOOKUP) when receiving an error of NFS4ERR_FHEXPIRED.

6.6. Exported Root

If the server's root file-system is exported, it might be easy to 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.

6.7. Mount Point Crossing

The server file-system environment may 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 NFS, the server would respond with the directory "/a/b/d/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.

6.8. Summary

NFS version 4 provides LOOKUP and READDIR operations for browsing of NFS file-systems. These operations are also used to browse server exports. A v4 server supports export browsing by including exported directories in a pseudo-file-system. A browsing client can cross seamlessly between a pseudo-file-system and a real, exported file-system. Clients must support volatile filehandles and recognize mount point crossing of server file-systems.

7. File Locking

Integrating locking into NFS necessarily causes it to be state-full, with the invasive nature of "share" file locks it 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 Windows "share" locks, it is necessary to atomically open or create files. Having a separate share/unshare operation will not allow correct implementation of the Windows OpenFile API. In order to correctly implement share semantics, the existing mechanisms used when a file is opened or created (LOOKUP, CREATE, ACCESS) need to be replaced. NFS V4 will have an OPEN procedure that subsumes the functionality of LOOKUP, CREATE, and ACCESS. However, because many operations require a file handle, the traditional LOOKUP is preserved to map a file name to file handle without establishing state on the server. Policy of granting access or modifying files is managed by the server based on the client's state. It is believed that these mechanisms can implement policy ranging from advisory only locking to full mandatory locking. While ACCESS is just a subset of OPEN, the ACCESS procedure is maintained as a lighter weight mechanism.

7.1. Definitions

Lock — The term "lock" will be used to refer to both record (byte-range) locks as well as file (share) locks unless specifically stated otherwise.

Client — Throughout this proposal the term "client" is used to indicate the entity that maintains a set of locks on behalf of one or more applications. The client is responsible for crash recovery of those locks it manages. Multiple clients may share the same transport and multiple clients may exist

on the same network node.

Clientid A 64-bit quantity returned by a server that uniquely corresponds to a client supplied Verifier and ID.

Lease An interval of time defined by the server for which the client is irrevokeably 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.

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. A stateid composed of all bits 0 or all bits 1 have special meaning and are reserved.

Verifier A 32-bit quantity generated by the client that the server can use to determine if the client has restarted and lost all previous lock state.

7.2. Locking

It is assumed that manipulating a lock is rare when compared to I/O operations. It is also assumed that crashes and network partitions are relatively rare. Therefore it is important that I/O operations have a light weight mechanism to indicate if they possess a held lock. A lock request contains the heavy weight 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.

7.2.1. 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, and 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. 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.

Two clients might still use the same nfs_client_id.id due to perhaps configuration error (say 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, nfs4err can be a switched union that returns in addition to NFS4ERR_CLID_IN_USE, 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 ia rogue client and the request is returned in error. For more discussion of crash recovery semantics, see the section on "Crash Recovery"

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 short hand reference to the nfs_client_id values presented by the client. From this point forward, the client can use the clientid to refer to itself.

7.2.2. 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 clients use of the SETCLIENTID procedure

- 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 short hand reference to the nfs_lockowner, since the server will be maintaining the correspondence between them.

7.2.3. Use of the stateid

All I/O requests contain a stateid. If the nfs_lockowner performs I/O on a range of bytes within a locked range, the stateid returned by the server must be used to indicate 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 grant the I/O request. If a conflict with an explicit lock occurs, the request is failed (NFS4ERR_LOCKED). This allows "mandatory locking" to be implemented.

A stateid of all bits 1 allows read requests to bypass locking checks at the server. However, write requests with stateid with bits all 1 does not bypass file locking requirements.

An explicit lock may not be granted while an I/O operation with conflicting implicit locking is being performed.

The byte range of a lock is indivisible. A range may be locked, unlocked, or changed between read and write but may not have subranges unlocked or changed between read and write. This is the semantics provided by Win32 but only a subset of the semantics provided by Unix. It is expected that Unix clients can more easily simulate modifying subranges than Win32 servers adding this feature.

7.2.4. Sequencing of lock requests

Locking is different than most NFS operations as it requires "at-most-one" semantics that are not provided by ONC RPC. 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 monotonically increasing integer. Different nfs_lockowners have different sequences. The server maintains the last sequence number (L) received and the response that was returned. If a request with a previous sequence number (r < L) is received it is silently ignored as its response 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 silently ignored. Sequences are reinitialized whenever the client verifier changes.

7.3. Blocking locks

Some clients require the support of blocking locks. The current proposal lacks a call-back mechanism, similar to NLM, to notify a client when the lock has been granted. Clients have no choice but to continually poll for the lock, which presents a fairness problem. Two new lock types are added, READW and WRITEW 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 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.

7.4. 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. Any I/O request that has been made with a valid stateid is a positive indication that the client is still alive and locks are being maintained. This becomes an implicit renewal of the lease. In the case no I/O has been performed within the lease interval, a lease can be renewed by having the client issue a zero length READ. Because the nfs_lockowner contains a unique client value, any stateid for a client will renew all leases for locks held with the same client field. This will allow very low overhead lease renewal that scales extremely well. In the typical case, no extra RPC calls are needed and in the worst case one RPC is required every lease period regardless of the number of locks held by the client.

7.5. 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 reboots. I/O operations that may have been queued within the client or network buffers, cannot complete until after the client has successfully recovered the lock protecting the I/O operation.

If a client fails, the server only needs to wait the lease period to allow conflicting locks. If the client reinitializes within the lease period, it may be forced to wait the remainder of the period before resuming service. To minimize this delay, lock requests contain a verifier field in the lock_owner, if the server receives a verifier field that does not match the existing verifier, the server knows that the client has lost all lock state and locks held for that client that do not match the current verifier may be released. In a secure environment, a change in the verifier must only cause the locks held by the authenticated requester to be released in order to prevent a rogue user from freeing otherwise valid locks. The verifier must have the same uniqueness properties of the COMMIT verifier.

If the server fails and loses locking state, the server must wait the lease period before granting any new locks or allowing any I/O. An I/O request during the grace period with an invalid stateid will fail with NFS4ERR_GRACE, the client will reissue the lock request with reclaim set to TRUE, and upon receiving a successful reply, the I/O may be reissued with the new stateid. Any time a client receives an

NFS4ERR_GRACE error it should start recovering all outstanding locks. A lock request during the grace period without reclaim set will also result in a NFS4ERR_GRACE, triggering the client recovery processing. A lock request outside the grace period with reclaim set will succeed only if the server can guarantee that no conflicting lock or I/O request has been granted since reboot.

In the case of a network partition longer than the lease period, the server will have not received an implicit lease renewal and may free all locks held for the client, thus invalidating any stateid held by the client. Subsequent reconnection will cause I/O with invalid stateid to fail with NFS4ERR_EXPIRED, the client will suitably notify the application holding the lock. After the lease period has expired the server may optionally continue to hold the locks for the client. In this case, if a conflicting lock or I/O request is received, the lock must be freed to allow the client to detect possible corruption. When there is a network partition and the lease expires, the server must record on stable storage the client information relating to those leases. This is to prevent the case where another client obtains the conflicting lock, frees the lock, and the server reboots. After the server recovers the original client may recover the network partition and attempt to reclaim the lock. Without any state to indicate that a conflicting may have occurred, the client could get in an inconsistent state. Storing just the client information is the minimal state necessary to detect this condition, but could lead to losing locks unnecessarily. However this is considered to be a very rare event, and a sophisticated server could store more state completely eliminate any unnecessary locks being lost.

7.6. Server revocation of locks

The server can revoke the locks held by a client at any time, when the client detects revocation it must ensure its state matches that of the server. If locks are revoked due to a server reboot, the client will receive a NFS4ERR_GRACE and normal crash recovery described above will be performed.

The server may revoke a lock within the lease period, this is considered a rare event likely to be initiated only by a human (as part of an administration task). The client may assume that only the file that caused the NFS4ERR_EXPIRED to be returned has lost the lock_owner's locks and notifies the holder appropriately. The client can not assume the lease period has been renewed.

The client not being able to renew the lease period is a relatively rare and unusual state. Both sides will detect this state and can recover without data corruption. The client must mark all locks held

as "invalidated" and then must issue an I/O request, either a pending I/O or zero length read to revalidate the lock. If the response is success the lock is upgraded to valid, otherwise it was revoked by the server and the owner is notified.

7.7. 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 that shares opens a file, it issues an OPEN request 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 applications open request.

Pseudo-code definition of the semantics:

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

Old DOS applications specify shares in compatibility mode. Microsoft has indicated in the Win32 specification that it will be deprecated in the future and recommends that deny NONE be used. This specification does not support compatibility mode.

7.8. OPEN/CLOSE procedures

To provide correct semantics for share semantics, a client MUST use the OPEN procedure to obtain the initial file handle 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 procedure. For clients that do not have a deny mode built into their open API, deny equal to NONE should be used.

The OPEN procedure with the CREATE flag, also subsumes the CREATE procedure for regular files as used in previous versions of NFS, allowing a create with a share to be done atomicly.

Will expand on create semantics here.

The CLOSE procedure removes all share locks held by the lock_owner on

that file. If record locks are held they should be explicitly unlocked. Some servers may not support the CLOSE of a file that still has record locks held; if so, CLOSE will fail and return an error.

The LOOKUP procedure is preserved and will return a file handle 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 file handle obtained through LOOKUP.

8. Defined Error Numbers

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_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_NOENT	No such file or directory. The file or directory name specified does not exist.
NFS4ERR_IO	I/O error. A hard error (for example, a disk error) occurred while processing the requested operation.
NFS4ERR_NXIO	I/O error. No such device or address.
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_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 (with exponential backoff of timeout) until the lock is accepted.
NFS4ERR_EXIST	File exists. The file specified already exists.

NFS4ERR_XDEV	Attempt to do a cross-device hard link.
NFS4ERR_NODEV	No such device.
NFS4ERR_NOTDIR	Not a directory. The caller specified a non-directory in a directory operation.
NFS4ERR_ISDIR	Is a directory. The caller specified a directory in a non-directory operation.
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_FBIG	File too large. The operation would have caused a file to grow beyond the server's limit.
NFS4ERR_NOSPC	No space left on device. The operation would have caused the server's file system to exceed its limit.
NFS4ERR_ROFS	Read-only file system. A modifying operation was attempted on a read-only file system.
NFS4ERR_MLINK	Too many hard links.
NFS4ERR_NAMETOOLONG	The filename in an operation was too long.
NFS4ERR_NOTEMPTY	An attempt was made to remove a directory that was not empty.
NFS4ERR_DQUOT	Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS4ERR_LOCKED	A read or write operation was attempted on a locked file.
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_BADHANDLE	Illegal NFS file handle. The file handle failed internal consistency checks.
NFS4ERR_NOT_SYNC	Update synchronization mismatch was detected during a SETATTR operation.
NFS4ERR_BAD_COOKIE	READDIR cookie is stale.
NFS4ERR_NOTSUPP	Operation is not supported.
NFS4ERR_TOOSMALL	Buffer or request is too small.
NFS4ERR_SAME	Returned if an NVERIFY operation shows that no attributes have changed.
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_BADTYPE	An attempt was made to create an object of a type not supported by the server.
NFS4ERR_JUKEBOX	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.

NFS4ERR_FHEXPIRED The file handle provided is volatile and has expired at the server. The client should attempt to recover the new file handle by traversing the server's file system name space. The file handle may have expired because the server has restarted, the file system object has been removed, or the file handle has been flushed from the server's internal mappings.

NOTE: This error definition will need to be crisp and match the section describing the volatile file handles.

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.

9. Compound Requests

NFS version 4 requires a client to combine multiple NFS operations into a single request. Compound requests provide:

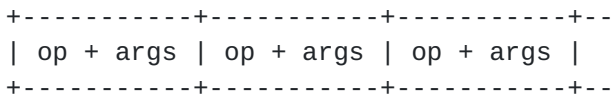
o Good performance on high latency networks

  If a client can combine multiple, dependent operations into a single request then it can avoid the cumulative latency in many request/response round-trips across the network. This is particularly important on the Internet or through geosynchronous satellite connections.

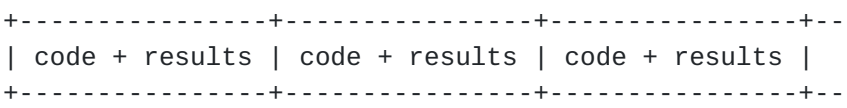
o Protocol simplification

  Clients can build NFS requests of arbitrary complexity from more primitive operations. These requests can be tailored to the unique needs of each client.

A compound request looks like this:



and the reply looks like this:



Where "code" is an indication of the success or failure of the operation including the opcode itself.

10. NFS Version 4 Requests

Nearly all NFS version 4 operations are defined as compound operations - not as RPC procedures. There is a single RPC procedure for all compound requests.

10.1. Evaluation of a Compound Request

The server evaluates the operations in sequence. Each operation consists of a 32 bit operation code, followed by a sequence of arguments 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 fails a non-zero status code will be encoded, evaluation of the compound request will halt, and the reply will be returned.

The client is responsible for recovering from any partially completed compound request.

Each operation assumes a "current" filehandle that is available as part of the execution context of the compound request. Operations may set, change, or return this filehandle.

11. NFS Version 4 Procedures

11.1. Procedure 0: NULL - No operation

SYNOPSIS

(cfh) -> (cfh)

ARGS

(none)

RESULTS

(none)

DESCRIPTION

The server does no work other than to return a NFS_OK result in the reply.

ERRORS

(none)

11.2. Procedure 1: ACCESS - Check Access Permission

SYNOPSIS

(cfh), permbits -> permbits

ARGS

permbits: uint32

RESULTS

permbits: uint32

DESCRIPTION

ACCESS determines the access rights that a user, as identified by the credentials in the request, has with respect to a file system object. The client encodes the set of permissions that are to be checked in a bit mask. The server checks the permissions encoded in the bit mask. A status of NFS4_OK is returned along with a bit mask encoded with the permissions that the client is allowed.

The results of this procedure are necessarily advisory in nature. That is, 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, as access rights can be revoked by the server at any time.

The following access permissions may be requested:

ACCESS_READ:	bit 0	Read data from file or read a directory.
ACCESS_LOOKUP:	bit 1	Look up a name in a directory (no meaning for non-directory objects).
ACCESS_MODIFY:	bit 2	Rewrite existing file data or modify existing directory entries.
ACCESS_EXTEND:	bit 3	Write new data or add directory entries.
ACCESS_DELETE:	bit 4	Delete an existing directory entry.
ACCESS_EXECUTE:	bit 5	Execute file (no meaning for a directory).

The server must return an error if the any access permission cannot be determined.

IMPLEMENTATION

In general, it is not sufficient for the client to attempt to deduce access permissions by inspecting the uid, gid, and mode fields in the file attributes, since the server may perform uid or gid mapping or enforce additional access control restrictions. It is also possible that the NFS version 4 protocol server may not be in the same ID space as the NFS version 4 protocol client. In these cases (and perhaps others), the NFS version 4 protocol 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. This is useful in operating systems (such as UNIX) where permission checking is done only when a directory is opened. This procedure is also invoked by NFS client access procedure (called possibly through access(2)). The intent is to make the behavior of opening a remote file more consistent with the behavior of opening a local file.

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

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 NFS version 4 protocol 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 ACCESS_DELETE permission. Operating systems like UNIX will ignore the ACCESS_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. Thus, the bit mask returned for such a request will have the ACCESS_DELETE bit set to 0, indicating that the client does not have this permission.

ERRORS

NFS4ERR_IO

NFS4ERR_SERVERFAULT

SEE

GETATTR.

11.3. Procedure 2: CLOSE - close file

SYNOPSIS

(cfh), stateid -> stateid

ARGS

stateid: uint64

RESULTS

stateid: uint64

DESCRIPTION

The CLOSE procedure notifies the server that all share locks corresponding to the client supplied stateid should be released.

IMPLEMENTATION

Share locks for the matching stateid will be released on successful completion of the CLOSE procedure.

ERRORS

To be determined

SEE

OPEN

11.4. Procedure 3: COMMIT - Commit cached data

SYNOPSIS

(cfh), offset, count -> verifier

Procedure COMMIT forces or flushes data to stable storage that was previously written with a WRITE operation with the stable field set to UNSTABLE.

ARGS

offset: uint64

The position within the file at which the flush is to begin. An offset of 0 means to flush data starting at the beginning of the file.

count: uint32

The number of bytes of data to flush. If count is 0, a flush from offset to the end of file is done.

RESULTS

verifier: uint32

This is a cookie that the client can use to determine whether the server has rebooted between a call to WRITE and a subsequent call to COMMIT. This cookie must be consistent during a single boot session and must be unique between instances of the NFS version 4 protocol server where uncommitted data may be lost.

IMPLEMENTATION

Procedure COMMIT is similar in operation and semantics to the POSIX fsync(2) system call that synchronizes a file's state with the disk, that is it flushes the file's data and metadata to disk. COMMIT performs the same operation for a client, flushing any unsynchronized data and metadata on the server to the server's disk 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 will always 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 verf 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.

This implementation will require some modifications to the buffer cache on the client. After a buffer is written with stable UNSTABLE, it must be considered as dirty by the client system until it is either flushed via a COMMIT operation or written via a WRITE operation with stable set to FILE_SYNC or DATA_SYNC. 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 comes back from either a WRITE or a COMMIT operation that contains an unexpected verf, 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 flag. If there more than one, it might be worthwhile retransmitting all of the buffers in WRITE requests with stable set to UNSTABLE 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_IO NFS4ERR_LOCKED NFS4ERR_SERVERFAULT

SEE

WRITE.

11.5. Procedure 4: CREATE - Create a non-regular file object

SYNOPSIS

(cfh), name, type, how -> (cfh)

ARGS

```
name: utf8string

objtype: filetype

how: union

   UNCHECKED:
   GUARDED:

       attrbits: bitmap
       attrvals

   EXCLUSIVE:

       verifier: createverf
```

RESULTS

(cfh): filehandle

DESCRIPTION

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

The objtype determines the type of object to be created: directory, symlink, etc. The how union may have a value of UNCHECKED, GUARDED, and EXCLUSIVE. UNCHECKED means that the object should be created without checking for the existence of a duplicate object in the same directory. In this case, attrbits and attrvals describe the initial attributes for the file object. GUARDED specifies that the server should check for the presence of a duplicate object before performing the create and should fail the request with NFS4ERR_EXIST if a duplicate object exists. 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. No attributes may be provided in

this case, since the server may use the target object meta-data to store the verifier.

The current filehandle is replaced by that of the new object.

IMPLEMENTATION

The CREATE procedure carries support for EXCLUSIVE create forward from NFS version 3. 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 CREATE 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.

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.

Note:

1. Need to determine an initial set of attributes that must be set, and a set of attributes that can optionally be set, on a per-filetype basis. For instance, if the filetype is a NF4BLK then the device attributes must be set.

2. Need to consider the symbolic link path as an "attribute". No need for a READLINK op if this is so. Similarly, a filehandle could be defined as an attribute for LINK.

3. The presence of a generic create for multiple file types makes the protocol easier to extend to new file types in a minor rev (without defining new ops)

4. The specific exclusive create semantics can be removed if there is guaranteed support for extended attributes. The client could specify the verifier be stored in an extended attribute and then check the attribute value itself instead of relying on the server to do so.

ERRORS

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_EXIST

NFS4ERR_NOTDIR

NFS4ERR_NOSPC

NFS4ERR_ROFS

NFS4ERR_NAMETOOLONG

NFS4ERR_DQUOT

NFS4ERR_NOTSUPP

NFS4ERR_SERVERFAULT

11.6. Procedure 5: GETATTR - Get attributes

SYNOPSIS

(cfh), attrbits -> attrbits, attrvals

ARGS

attrbits: bitmap

RESULTS

attrbits: bitmap

attrvals: sequence of attributes

DESCRIPTION

Obtain attributes from the server. 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 that 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 attribute 0 which is a bitmap of all supported attributes for the filesystem object.

IMPLEMENTATION

?

ERRORS

NFS4ERR_IO

NFS4ERR_SERVERFAULT

11.7. Procedure 6: GETFH - Get current filehandle

SYNOPSIS

(cfh) -> filehandle

ARGS

RESULTS

filehandle: filehandle

DESCRIPTION

Returns the current filehandle. Operations that change the current filehandle like LOOKUP or CREATE to 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 will do it:

```
1: PUTFH  (directory filehandle)
2: LOOKUP (entry name)
3: GETFH
```

IMPLEMENTATION

?

ERRORS

NFS4ERR_SERVERFAULT

11.8. Procedure 7: LINK - Create link to an object

SYNOPSIS

(cfh), dir, newname -> (cfh)

ARGS

dir: filehandle

newname: utf8string

RESULTS

(none)

DESCRIPTION

Procedure LINK creates an additional newname for the file with the current filehandle in the new directory dir file and link.dir must reside on the same file system and server. On entry, the arguments in LINK3args are:

IMPLEMENTATION

Changes to any property of the hard-linked files are reflected in all of the linked files. When a hard link is made to a file, the attributes for the file should have a value for nlink that is one greater than the value before the LINK.

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.

ERRORS

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_EXIST

NFS4ERR_XDEV

NFS4ERR_NOTDIR

NFS4ERR_INVAL

NFS4ERR_NOSPC

NFS4ERR_ROFS

NFS4ERR_MLINK

NFS4ERR_NAMETOOLONG

NFS4ERR_DQUOT

NFS4ERR_NOTSUPP

NFS4ERR_SERVERFAULT

11.9. Procedure 8: LOCK - Create lock

SYNOPSIS

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

ARGS

type: {READ, WRITE, READW, WRITEW}

seqid: uint32

reclaim: boolean

owner: nfs_lockowner

offset: uint64

length: uint64

RESULTS

stateid: uint64

access: int

DESCRIPTION

The LOCK procedure requests that a record lock starting at 'offset' for length 'length' be set on the file represented by 'cfh'. The integer. The 'reclaim' field is used for failure recovery.

IMPLEMENTATION

See locking section for now.

ERRORS

To be determined.

11.10. Procedure 9: LOCKT - test for lock

SYNOPSIS

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

ARGStype: {READ, WRITE, READW, WRITEW}

seqid: uint32

reclaim: boolean

owner: nfs_lockowner

offset: uint64

length: uint64

RESULTS

owner: nfs_lockowner

DESCRIPTION

The LOCKT procedure tests the lock specified by the parameters. The owner of the lock is returned in the event it is currently being held; if no lock is held, nothing other than NFS4_OK is returned.

ERRORS

NFS4ERR_DENIED

11.11. Procedure 10: LOCKU - Unlock file

SYNOPSIS

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

ARGS

type: {READ, WRITE, READW, WRITEW}

seqid: uint32

reclaim: boolean

owner: nfs_lockowner

offset: uint64

length: uint64

RESULTS

stateid: uint64

DESCRIPTION

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

ERRORS

To be determined.

11.12. Procedure 11: LOOKUP - Lookup filename

SYNOPSIS

(cfh), filenames -> (cfh)

ARGS

filename: utf8string[]

RESULTS

(none)

DESCRIPTION

The current filehandle is assumed to refer to a directory. 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 doesn't exist or because the client doesn't have permission to evaluate a component of the path, then an error will be returned and the current filehandle will be unchanged.

IMPLEMENTATION

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

1. PUTFH (directory filehandle)
2. LOOKUP "pub" "foo" "bar"
3. GETFH

or

1. PUTFH (directory filehandle)
2. LOOKUP "pub"
3. GETFH
4. LOOKUP "foo"
5. GETFH
6. LOOKUP "bar"
7. 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.

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 namespace. 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.

ERRORS

NFS4ERR_IO

NFS4ERR_NOENT

NFS4ERR_ACCES

NFS4ERR_NOTDIR

NFS4ERR_NAMETOOLONG

NFS4ERR_SERVERFAULT

SEE

CREATE

11.13. Procedure 12: LOOKUPP - Lookup parent directory

SYNOPSIS

(cfh) -> (cfh)

ARGS

(none)

RESULTS

(none)

DESCRIPTION

The current filehandle is assumed to refer to a directory. LOOKUPP assigns the filehandle for its parent directory to be the current filehandle. If there is no parent directory an ENOENT error must be returned.

IMPLEMENTATION

As for LOOKUP, LOOKUPP will also cross mountpoints.

ERRORS

NFS4ERR_IO

NFS4ERR_NOENT

NFS4ERR_ACCES

NFS4ERR_SERVERFAULT

SEE

CREATE

11.14. Procedure 13: NVERIFY - Verify attributes different

SYNOPSIS

(cfh), attrbits, attrvals -> -

ARGS

attrbits: bitmap

attrvals: sequence of attributes

RESULTS

(none)

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.

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:

1. PUTFH (public)
2. LOOKUP "pub" "foo" "bar"
3. NVERIFY attrbits attrs
4. READ 0 32767

ERRORS

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_SERVERFAULT

NFS4ERR_SAME

11.15. Procedure 14: OPEN - Open a regular file

SYNOPSIS

(cfh) filename, flag, owner, seqid, reclaim, access, deny -> stateid, access

ARGS

filename: utf8string

flag: openflag (union (createhow4, void))

owner: nfs_lockowner

seqid: uint32

reclaim: boolean

access: int (flag)

deny: int (flag)

RESULTS

stateid: uint64

access: int

DESCRIPTION

OPEN

Procedure OPEN creates and/or opens a regular file in a directory with a given name. The flag determines if the file should be created if it does not exist and the how union contains a value of UNCHECKED, GUARDED, or EXCLUSIVE. UNCHECKED means that the file should be created without checking for the existence of a duplicate object in the same directory. In this case, attrbits and attrvals describe the initial attributes for the file. GUARDED specifies that the server should check for the presence of a duplicate object before performing the create and should fail the request with NFS4ERR_EXIST if a duplicate object exists. 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. No attributes may be provided in

this case, since the server may use the target object meta-data to store the verifier.

The current filehandle is replaced by that of the new object.

IMPLEMENTATION

The OPEN procedure carries support for EXCLUSIVE create forward from NFS version 3. 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.

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.

Note: Need to determine an initial set of attributes that must be set, and a set of attributes that can optionally be set.

ERRORS

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_EXIST

NFS4ERR_NOTDIR

NFS4ERR_NOSPC

NFS4ERR_ROFS

NFS4ERR_NAMETOOLONG

NFS4ERR_DQUOT

NFS4ERR_NOTSUPP

NFS4ERR_SERVERFAULT

11.16. Procedure 15: PUTFH - Set current filehandle

SYNOPSIS

filehandle -> (cfh)

ARGS

filehandle: filehandle

RESULTS

(none)

DESCRIPTION

Replaces the current filehandle with the filehandle provided as an argument. If no filehandle has previously been installed as the current filehandle then root filehandle is assumed. If the length of the filehandle is zero, it is recognized by the server as a "public" filehandle.

IMPLEMENTATION

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

ERRORS

NFS4ERR_BADHANDLE

NFS4ERR_SERVERFAULT

11.17. Procedure 16: PUTROOTFH - Set root filehandle

SYNOPSIS

- -> (cfh)

ARGS

(none)

RESULTS

(none)

DESCRIPTION

Replaces the current filehandle with the filehandle that represents the root of the server's namespace. 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 any NFS request to set the context for following operations.

ERRORS

NFS4ERR_SERVERFAULT

11.18. Procedure 17: READ - Read from file

SYNOPSIS

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

ARGS

offset: uint64

count: uint32

stateid: uint64

RESULTS

eof: bool

data: opaque <>

DESCRIPTION

READ reads data from the file identified by the current filehandle.

offset

The position within the file at which the read is to begin. An offset of 0 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 count set to 0 and eof set to TRUE, subject to access permissions checking.

count

The number of bytes of data that are to be read. If count is 0, the READ will succeed and return 0 bytes of data, subject to access permissions checking. count must be less than or equal to the value of the rtmax for the file system that contains file. If greater, the server may return only rtmax bytes, resulting in a short read.

stateid

The stateid returned from a previous record or share lock request. Used by the server to verify that the associated

lock is still valid and to update lease timeouts for the client.

If the operation is successful the results are:

eof

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), eof is returned as TRUE; otherwise it is FALSE. A successful READ of an empty file will always return eof as TRUE.

data

The counted data read from the file.

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_IO

NFS4ERR_NXIO

NFS4ERR_ACCES

NFS4ERR_INVAL

NFS4ERR_LOCKED

NFS4ERR_SERVERFAULT

11.19. Procedure 18: READDIR - Read directory

SYNOPSIS

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

ARGS

cookie: uint64

This should be set to 0 in the first request to read the directory. On subsequent requests, it should be a cookie as returned by the server.

dircount: uint32

The maximum number of bytes of directory information returned. This number should not include the size of the attributes and file handle portions of the result.

maxcount: uint32

The maximum size of the result in bytes. The size must include all XDR overhead. The server is free to return less than count bytes of data.

attrbits: bitmap

The attributes to be returned for each directory entry.

RESULTS

A list of directory entries. Each entry contains:

cookie: uint64

A value recognized by the server as a "bookmark" into the directory. It may be an offset or an index into a table. Ideally, the cookie value should not change if the directory is modified.

filename: utf8string;

The name of the directory entry.

attrbits: bitmap

A bitmap that indicates which attributes follow. Ideally this bitmap will be identical to the attribute bitmap in the arguments, i.e. the server returns everything the client asked for. However, the returned bitmap may be different if the server does not support the attribute or if the attribute is not valid for the filetype.

Note: need to consider the file handle as an "attribute" that may be optionally returned. The concept of file handle as attribute might also be useful for the CREATE of a hard link.

DESCRIPTION

Procedure READDIR retrieves a variable number of entries from a file system directory and returns complete information about each entry along with information to allow the client to request additional directory entries in a subsequent READDIR.

IMPLEMENTATION

Issues that need to be understood for this procedure include increased cache flushing activity on the client (as new file handles are returned with names which are entered into caches) and over-the-wire overhead versus expected subsequent LOOKUP and GETATTR elimination.

The dircount and maxcount fields are included as an optimization. Consider a READDIR call on a UNIX operating system implementation for 1048 bytes; the reply does not contain many entries because of the overhead due to attributes and file handles. An alternative is to issue a READDIR call for 8192 bytes and then only use the first 1048 bytes of directory information. However, the server doesn't know that all that is needed is 1048 bytes of directory information (as would be returned by READDIR). It sees the 8192 byte request and issues a VOP_READDIR for 8192 bytes. It then steps through all of those directory entries, obtaining attributes and file handles for each entry. When it encodes the result, the server only encodes until it gets 8192 bytes of results which include the attributes and file handles. Thus, it has done a larger VOP_READDIR and many more attribute fetches than it needed to. The ratio of the directory entry size to the size of the attributes plus the size of the file handle is usually at least 8 to 1. The server has done much more work than it needed to.

The solution to this problem is for the client to provide two counts to the server. The first is the number of bytes of

directory information that the client really wants, dircount. The second is the maximum number of bytes in the result, including the attributes and file handles, maxcount. Thus, the server will issue a VOP_READDIR for only the number of bytes that the client really wants to get, not an inflated number. This should help to reduce the size of VOP_READDIR requests on the server, thus reducing the amount of work done there, and to reduce the number of VOP_LOOKUP, VOP_GETATTR, and other calls done by the server to construct attributes and file handles.

ERRORS

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_NOTDIR

NFS4ERR_BAD_COOKIE

NFS4ERR_TOOSMALL

NFS4ERR_NOTSUPP

NFS4ERR_SERVERFAULT

11.20. Procedure 19: READLINK - Read symbolic link

SYNOPSIS

(cfh) -> linktext

ARGS

(none)

RESULTS

linktext: utf8string

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.

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, NFLNK. The server should return the error, NFS4ERR_INVAL, if the object is not of type, NFLNK.

ERRORS

NFS4ERR_IO

NFS4ERR_INVAL

NFS4ERR_ACCES

NFS4ERR_NOTSUPP

NFS4ERR_SERVERFAULT

11.21. Procedure 20: REMOVE - Remove filesystem object

SYNOPSIS

(cfh), filename -> -

ARGS

entryname: utf8string

RESULTS

(none)

DESCRIPTION

REMOVE 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.

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 filetype.

The concept of last reference is server specific. However, if the nlink 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 use RENAME to rename the file from its old name to a new hidden name.

ERRORS

NFS4ERR_NOENT

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_NOTDIR

NFS4ERR_NAMETOOLONG

NFS4ERR_ROFS

NFS4ERR_NOTEMPTY

NFS4ERR_SERVERFAULT

11.22. Procedure 21: RENAME - Rename directory entry

SYNOPSIS

(cfh), oldname, newdir, newname -> -

ARGS

oldname: utf8string

newdir: filehandle

newname: utf8string

RESULTS

status: uint32

DESCRIPTION

RENAME renames the directory identified by oldname in the directory corresponding to the current filehandle to newname in directory newdir. The operation is required to be atomic to the client. Source and target directories must reside on the same file system on the server.

If the directory, newdir, 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 should return the error, NFS4ERR_EXIST.

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. Even though the operation is atomic, the status, NFS4ERR_MLINK, may be returned if the server used a "unlink/link/unlink" sequence internally.

A file handle may or may not become stale on a rename. However, server implementors are strongly encouraged to attempt to keep file handles from becoming stale 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.

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.

ERRORS

NFS4ERR_NOENT

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_EXIST

NFS4ERR_XDEV

NFS4ERR_NOTDIR

NFS4ERR_ISDIR

NFS4ERR_INVAL

NFS4ERR_NOSPC

NFS4ERR_ROFS

NFS4ERR_MLINK

NFS4ERR_NAMETOOLONG

NFS4ERR_NOTEMPTY

NFS4ERR_DQUOT

NFS4ERR_NOTSUPP

NFS4ERR_SERVERFAULT

11.23. Procedure 22: RENEW - renew a lease

SYNOPSIS

stateid -> ()

ARGS

stateid: uint64 length: uint64

RESULTS

none

DESCRIPTION

Renews all leases for the client associated with the stateid.

ERRORS

TDB

11.24. Procedure 23: RESTOREFH - Restore saved filehandle

SYNOPSIS

(sfh) -> (cfh)

ARGS

(none)

RESULTS

(none)

DESCRIPTION

Make the saved filehandle the current filehandle. If there is no saved filehandle then return an error NFS4ERR_INVAL.

IMPLEMENTATION

Operators like CREATE 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.

1. PUTFH (directory filehandle)
2. SAVEFH
3. GETATTR attrbits (pre-op dir attrs)
4. CREATE optbits "foo" attrs
5. GETATTR attrbits (file attributes)
6. RESTOREFH
7. GETATTR attrbits (post-op dir attrs)

ERRORS

NFS4ERR_SERVERFAULT

11.25. Procedure 24: SAVEFH - Save current filehandle

SYNOPSIS

(cfh) -> (sfh)

ARGS

(none)

RESULTS

(none)

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.

IMPLEMENTATION

(see RESTOREFH)

ERRORS

NFS4ERR_SERVERFAULT

11.26. Procedure 25: SECINFO - Obtain Available Security

SYNOPSIS

(cfh), filename -> { secinfo }

ARGS

filename: utf8string

RESULTS

secinfo: secinfo
This is a link list of security flavors available for the supplied file handle and filename.

DESCRIPTION

This procedure is used by the client to obtain a list of valid RPC authentication flavors for a specific file handle, file name pair. For the flavors, AUTH_NONE, AUTH_SYS, AUTH_DH, and AUTH_KRB4 no additional security information is returned. For a return value of AUTH_RPCSEC_GSS, a security triple is returned that contains the mechanism object id (as defined in [RFC2078]), the quality of protection (as defined in [RFC 2078]) and the service type (as defined in [RFC2203]). It is possible for SECINFO to return multiple entries with flavor equal to AUTH_RPCSEC_GSS with different security triple values.

IMPLEMENTATION

This procedure is expected to be used by the NFS client when the error value of NFS4ERR_WRONGSEC is returned from another NFS procedure. 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.

ERRORS

NFS4ERR_NOENT

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_NAMETOOLONG

NFS4ERR_STALE

NFS4ERR_SERVERFAULT

NFS4ERR_FHEXPIRED

NFS4ERR_WRONGSEC

11.27. Procedure 26: SETATTR - Set attributes

SYNOPSIS

(cfh), attrbits, attrvals -> -

ARGS

attrbits: bitmap

attrvals

DESCRIPTION

Procedure SETATTR changes one or more of the attributes of a file system object on the server. The new attributes are specified with a bitmap and the attributes that follow the bitmap in bit order.

IMPLEMENTATION

The file size attribute is used to request changes to the size of a file. A value of 0 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 mtime. 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. An error may be returned if the server can not store a uid or gid in its own representation of uids or gids, respectively. 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_PERM

NFS4ERR_IO

NFS4ERR_ACCES

NFS4ERR_INVAL

NFS4ERR_NOSPC

NFS4ERR_ROFS

NFS4ERR_DQUOT

NFS4ERR_SERVERFAULT

11.28. Procedure 27: SETCLIENTID - negotiated clientid

SYNOPSIS

verifier, client -> clientid

ARGS

verifier: uint32

client: opaque <>

RESULTS

clientid: uint64

DESCRIPTION

Procedure SETCLIENTID 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.

IMPLEMENTATION

The server takes the verifier and client identification supplied and search 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 short hand reference to the supplied information.

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

ERRORS

TBD

11.29. Procedure 28: VERIFY - Verify attributes same

SYNOPSIS

(cfh), attrbits, attrvals -> -

ARGS

attrbits: bitmap

attrvals

RESULTS

(none)

DESCRIPTION

This operation is used to verify that attributes have a value assumed by the client before proceeding with following operations in the compound request. For instance, a VERIFY can be used to make sure that the file size has not changed for an append-mode write:

1. PUTFH 0x0123456
2. VERIFY attrbits attrs
3. WRITE 450328 4096

If the attributes are not as expected, then the request fails and the data is not appended to the file.

IMPLEMENTATION

ERRORS

11.30. Procedure 29: WRITE - Write to file

SYNOPSIS

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

ARGS

offset: uint64

count: uint32

stability: uint32

stateid: uint64

data: opaque

RESULTS

count: uint32

committed: uint32

verifier: uint32

DESCRIPTION

Write data to the file identified by the current filehandle. Arguments are as follows:

offset

The position within the file at which the write is to begin. An offset of 0 means to write data starting at the beginning of the file.

count

The number of bytes of data to be written. If count is 0, the WRITE will succeed and return a count of 0, barring errors due to permissions checking. The size of data must be less than or equal to the value of the wtmax attribute for the filesystem that contains file. If greater, the server may write only wtmax bytes, resulting in a short write.

stability

If stable is FILE_SYNC, 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_SYNC, 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_SYNC in the same fashion as FILE_SYNC, but with a possible performance drop. If stable is UNSTABLE, 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.

stateid

The stateid returned from a previous record or share lock request. Used by the server to verify that the associated lock is still valid and to update lease timeouts for the client.

data

The data to be written to the file.

If the operation is successful the following results are returned:

count

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.

committed

The server should return 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_SYNC. If the level of commitment was at least as strong as DATA_SYNC, then

committed should be set to DATA_SYNC. Otherwise, committed must be returned as UNSTABLE. If stable was FILE_SYNC, then committed must also be FILE_SYNC: anything else constitutes a protocol violation. If stable was DATA_SYNC, then committed may be FILE_SYNC or DATA_SYNC: anything else constitutes a protocol violation. If stable was UNSTABLE, then committed may be either FILE_SYNC, DATA_SYNC, or UNSTABLE.

verifier

This 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 UNSTABLE and the reply yields a committed response of DATA_SYNC or UNSTABLE, 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.

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 mtime of the file to be updated. However, the mtime 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 mtime 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 UNSTABLE and in which the result committed was returned as UNSTABLE 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_SYNC, 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_IO

NFS4ERR_ACCES

NFS4ERR_FBIG

NFS4ERR_DQUOT

NFS4ERR_NOSPC

NFS4ERR_ROFS

NFS4ERR_INVAL

NFS4ERR_LOCKED

NFS4ERR_SERVERFAULT

12. Locking notes

12.1. Short and long leases

The usual lease trade-offs 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 huge numbers of clients. RENEW requests drop in direct proportion to the lease time. The disadvantages of long leases are slower server recover after crash (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 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.

12.2. Clocks and leases

To avoid the need for synchronized clocks, lease times are granted by the server as a time delta, though 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 across the Internet 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 a 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'll take another 200 msec to get a response back to the server. So the client must send a lock renewal or write data back to the server 400 msec before the lease would expire.

The client could measure propagation delay with reasonable accuracy by measuring the round-trip time for lock extensions assuming that there's not much server processing overhead in an extension.

12.3. Locks and lease times

Lock requests do not contain desired lease times. The server

allocates leases with no information from the client. The assumption here is that the client really has no idea of just how long the lock will be required. If a scenario can be found where a hint from the client as to the maximum lease time desired would be useful, then this feature could be added to lock requests.

12.4. Locking of directories and other meta-files

A question: should directories and/or other file-system objects like symbolic links be lockable ? Clients will want to cache whole directories. It would be nice to have consistent directory caches, but it would require that any client creating a new file get a write lock on the directory and be prepared to handle lock denial. Is the weak cache consistency that we currently have for directories acceptable ? I think perhaps it is - given the expense of doing full consistency on an Internet scale.

12.5. Proxy servers and leases

Proxy servers. There is some interest in having NFS V4 support caching proxies. Support for proxy caching is a requirement if servers are to handle large numbers of clients - clients that may have little or no ability to cache on their own. How could proxy servers use lease-based locking ?

12.6. Locking and the new latency

Latency caused by locking. If a client wants to update a file then it will have to wait until the leases on read locks have expired. If the leases are of the order of 60 seconds or several minutes then the client (and end-user) may be blocked for a while. This is unfamiliar for current NFS users who are not bothered by mandatory locking - but it could be an issue if we decide we like the caching benefits. A similar problem exists for clients that wish to read a file that is write locked. The read-lock case is likely to be more common if read-locking is used to protect cached data on the client.

13. 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. NFS' choice of string representation must allow reasonable name/string access to clients which use various languages. The UTF-8 encoding allows for this type of access and this choice is explained in the following.

13.1. 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 are consecutively allocated), and normalize their offsets to a table that fits in 7 bits. Thus we find 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 we have 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 NFS V4 used a universal 16 bit or 32 bit character set (or a encoding of a 16 bit or 32 bit character set into octets), then 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.

13.2. Overview of Universal Character Set Standards

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

[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 — UCS transformation format.

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.

13.3. 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], 38,887 Unicode characters are already assigned. And according to [Unicode2] there are still more languages that need to be added.

13.4. 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.

14. 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".

As this document progresses, other issues of denial of service and other typical security issues will be addressed here along with those issues specific to NFS service.

15. NFS Version 4 RPC definition file

```
/*
 *      nfs_prot.x
 *
 */

%#pragma ident  "@(#)nfs_prot.x 1.28    99/02/26"

/*
 * Sizes
 */
const NFS4_FHSIZE         = 128;
const NFS4_CREATEVERFSIZE = 8;

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

struct specdata4 {
        uint32_t        specdata1;
        uint32_t        specdata2;
};

/*
 * Basic data types
 */
typedef opaque          utf8string<>;
typedef uint64_t        offset4;
typedef uint32_t        count4;
typedef uint32_t        length4;
typedef uint64_t        clientid4;
typedef uint64_t        stateid4;
typedef uint32_t        seqid4;
typedef uint32_t        writeverf4;
typedef opaque          createverf4[NFS4_CREATEVERFSIZE];
typedef utf8string      filename4;
typedef uint64_t        nfs_lockid4;
typedef uint32_t        nfs_lease4;
typedef uint32_t        nfs_lockstate4;
typedef uint64_t        nfs_cookie4;
typedef utf8string      linktext4;
typedef opaque          sec_oid4<>;
typedef uint32_t        qop4;
```

```
   typedef uint32_t        fattr4_type;
   typedef uint32_t        fattr4_mode;
   typedef uint32_t        fattr4_accessbits;
   typedef uint32_t        fattr4_nlink;
   typedef utf8string      fattr4_uid;
   typedef utf8string      fattr4_gid;
   typedef uint64_t        fattr4_size;
   typedef uint64_t        fattr4_used;
   typedef specdata4       fattr4_rdev;
   typedef uint64_t        fattr4_fsid;
   typedef uint64_t        fattr4_fileid;
   typedef nfstime4        fattr4_atime;
   typedef nfstime4        fattr4_mtime;
   typedef nfstime4        fattr4_ctime;
   typedef uint32_t        fattr4_rtmax;
   typedef uint32_t        fattr4_rtpref;
   typedef uint32_t        fattr4_rtmult;
   typedef uint32_t        fattr4_wtmax;
   typedef uint32_t        fattr4_wtpref;
   typedef uint32_t        fattr4_wtmult;
   typedef uint32_t        fattr4_dtpref;
   typedef uint64_t        fattr4_maxfilesize;
   typedef uint64_t        fattr4_change;
   typedef nfstime4        fattr4_time_delta;
   typedef uint32_t        fattr4_properties;
   typedef uint32_t        fattr4_linkmax;
   typedef uint32_t        fattr4_name_max;

   /*
    * 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_NOT_SYNC        = 10002,
        NFS4ERR_BAD_COOKIE      = 10003,
        NFS4ERR_NOTSUPP         = 10004,
        NFS4ERR_TOOSMALL        = 10005,
        NFS4ERR_SERVERFAULT     = 10006,
        NFS4ERR_BADTYPE         = 10007,
        NFS4ERR_JUKEBOX         = 10008,
        NFS4ERR_SAME            = 10009,
        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 */
};

enum rpc_flavor4 {
        AUTH_NONE       = 0,
        AUTH_SYS        = 1,
        AUTH_DH         = 2,
        AUTH_KRB4       = 3,
        AUTH_RPCSEC_GSS = 4
};

/*
 * From RFC 2203
 */
enum rpc_gss_svc_t {
        RPC_GSS_SVC_NONE        = 1,
        RPC_GSS_SVC_INTEGRITY   = 2,
        RPC_GSS_SVC_PRIVACY     = 3
};

/*
 * File access handle
 */
struct nfs_fh4 {
        opaque          data<NFS4_FHSIZE>;
};

/*
 * File types
 */
enum ftype4 {
```

```
        NF4REG          = 1,
        NF4DIR          = 2,
        NF4BLK          = 3,
        NF4CHR          = 4,
        NF4LNK          = 5,
        NF4SOCK         = 6,
        NF4FIFO         = 7
};

const FATTR4_TYPE               = 1;
const FATTR4_MODE               = 2;
const FATTR4_ACCESSBITS         = 3;
const FATTR4_NLINK              = 4;
const FATTR4_UID                = 5;
const FATTR4_GID                = 6;
const FATTR4_SIZE               = 7;
const FATTR4_USED               = 8;
const FATTR4_RDEV               = 9;
const FATTR4_FSID               = 10;
const FATTR4_FILEID             = 11;
const FATTR4_ATIME              = 12;
const FATTR4_MTIME              = 13;
const FATTR4_CTIME              = 14;
const FATTR4_RTMAX              = 15;
const FATTR4_RTPREF             = 16;
const FATTR4_RTMULT             = 17;
const FATTR4_WTMAX              = 18;
const FATTR4_WTPREF             = 19;
const FATTR4_WTMULT             = 20;
const FATTR4_DTPREF             = 21;
const FATTR4_MAXFILESIZE        = 22;
const FATTR4_TIME_DELTA         = 23;
const FATTR4_PROPERTIES         = 24;
const FATTR4_LINKMAX            = 25;
const FATTR4_NAME_MAX           = 26;
const FATTR4_NO_TRUNC           = 27;
const FATTR4_CHOWN_RESTRICTED   = 28;
const FATTR4_CASE_INSENSITIVE   = 29;
const FATTR4_CASE_PRESERVING    = 30;

/*
 * fattr4_properties bits
 */
const FSF_LINK                  = 0x00000001;
const FSF_SYMLINK               = 0x00000002;
const FSF_HOMOGENEOUS           = 0x00000004;
const FSF_CANSETTIME            = 0x00000008;
const FSF_NOTRUNC               = 0x00000010;
```

```
   const FSF_CHOWN_RESTRICTED      = 0x00000020;
   const FSF_CASE_INSENSITIVE      = 0x00000040;
   const FSF_CASE_PRESERVING       = 0x00000080;

   struct bitmap4 {
           uint32_t        bits<>;
   };

   struct attrlist {
           opaque          attrs<>;
   };

   struct fattr4 {
           bitmap4         attrmask;
           attrlist        attr_vals;
   };

   struct cid {
           opaque          verifier<4>;
           opaque          id<>;
   };

   union nfs_client_id switch (clientid4 clientid) {
    case 0:
            cid            ident;
    default:
            void;
   };

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

   union nfs_lockowner switch (stateid4 stateid) {
    case 0:
            lockown        ident;
    default:
            void;
   };

   enum lock_type {
           READ    = 1,
           WRITE   = 2,
           READW   = 3,    /* blocking read */
           WRITEW  = 4     /* blocking write */
   };
```

```
   /*
    * ACCESS: Check access permission
    */
   const ACCESS4_READ      = 0x0001;
   const ACCESS4_LOOKUP    = 0x0002;
   const ACCESS4_MODIFY    = 0x0004;
   const ACCESS4_EXTEND    = 0x0008;
   const ACCESS4_DELETE    = 0x0010;
   const ACCESS4_EXECUTE   = 0x0020;

   struct ACCESS4args {
           uint32_t        access;
   };

   struct ACCESS4resok {
           uint32_t        access;
   };

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

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

   struct COMMIT4resok {
           writeverf4      verf;
   };


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

   /*
    * CREATE: Create a file
    */
```

```
   enum createmode4 {
           UNCHECKED       = 0,
           GUARDED         = 1,
           EXCLUSIVE       = 2
   };

   union createhow4 switch (createmode4 mode) {
    case UNCHECKED:
    case GUARDED:
            fattr4         createattrs;
    case EXCLUSIVE:
            createverf4    verf;
   };

   const ACCESS4_READ      = 0x0001;
   const ACCESS4_MODIFY    = 0x0002;
   const ACCESS4_LOOKUP    = 0x0004;
   const ACCESS4_EXTEND    = 0x0008;
   const ACCESS4_DELETE    = 0x0010;
   const ACCESS4_EXECUTE   = 0x0020;

   const DENY4_NONE        = 0x0000;
   const DENY4_READ        = 0x0001;
   const DENY4_WRITE       = 0x0002;

   union openflag switch (uint32_t flag) {
    case CREATE:
            createhow4     how;
    default:
            void;
   };

   /*
    * LOCK/LOCKT/LOCKU: Record lock management
    */
   struct LOCK4args {
           lock_type       type;
           seqid4          seqid;
           bool            reclaim;
           nfs_lockowner   owner;
           offset4         offset;
           length4         length;
   };

   struct lockres {
           stateid4        stateid;
           int32_t         access;
   };
```

```
   union LOCK4res switch (nfsstat4 status) {
    case NFS4_OK:
            lockres        result;
    default:
            void;
   };

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

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

   /*
    * SETCLIENTID
    */
   struct SETCLIENTID4args {
           seqid4          seqid;
           nfs_client_id   client;
   };

   union SETCLIENTID4res switch (nfsstat4 status) {
    case NFS4_OK:
            clientid4      clientid;
    default:
            void;
   };

   /*
    * OPEN: Open a file, potentially with a share lock
    */
   struct OPEN4args {
           filename4       filenames<>;
           openflag        flag;
           nfs_lockowner   owner;
           seqid4          seqid;
           bool            reclaim;
           int32_t         access;
```

```
        int32_t         deny;
};

union OPEN4res switch (nfsstat4 status) {
 case NFS4_OK:
         LOCK4resok     resok;
 default:
         void;
};

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

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

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

struct GETATTR4resok {
        fattr4          obj_attributes;
};

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

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

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

   /*
    * LINK: Create link to an object
    */
   struct LINK4args {
           nfs_fh4         dir;
           filename4       newname;
   };

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

   /*
    * LOOKUP: Lookup filename
    */
   struct LOOKUP4args {
           filename4       filenames<>;
   };

   union LOOKUP4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

   /*
    * LOOKUPP: Lookup parent directory
    */
   union LOOKUPP4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

   /*
    * NVERIFY: Verify attributes different
```

```
    */
   struct NVERIFY4args {
           bitmap4         attr_request;
           fattr4          obj_attributes;
   };

   union NVERIFY4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

   /*
    * RESTOREFH: Restore saved filehandle
    */

   union RESTOREFH4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

   /*
    * SAVEFH: Save current filehandle
    */
   union SAVEFH4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

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

   union PUTFH4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };
```

```
   /*
    * PUTROOTFH: Set root filehandle
    */
   union PUTROOTFH4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

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

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

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

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

   };

   struct entry4 {
           cookie4         cookie;
           filename4       name;
           fattr4          attrs;
           entry4          *nextentry;
   };
```

```
   struct dirlist4 {
           entry4          *entries;
           bool            eof;
   };

   struct READDIR4resok {
           dirlist4        reply;
   };


   union READDIR4res switch (nfsstat4 status) {
    case NFS4_OK:
            READDIR4resok  resok;
    default:
            void;
   };


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

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

   /*
    * REMOVE: Remove filesystem object
    */
   struct REMOVE4args {
           filename4       target;
   };

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

   /*
    * RENAME: Rename directory entry
```

```
    */
   struct RENAME4args {
           filename4       oldname;
           nfs_fh4         newdir;
           filename4       newname;
   };

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

   struct RENEW4args {
           stateid4        stateid;
   };

   union RENEW4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

   /*
    * SETATTR: Set attributes
    */
   struct SETATTR4args {
           fattr4          obj_attributes;
   };

   union SETATTR4res switch (nfsstat4 status) {
    case NFS4_OK:
            void;
    default:
            void;
   };

   /*
    * VERIFY: Verify attributes same
    */
   struct VERIFY4args {
           bitmap4         attr_request;
           fattr4          obj_attributes;
   };

   union VERIFY4res switch (nfsstat4 status) {
```

```
    case NFS4_OK:
            void;
    default:
            void;
   };

   /*
    * WRITE: Write to file
    */
   enum stable_how4 {
           UNSTABLE        = 0,
           DATA_SYNC       = 1,
           FILE_SYNC       = 2
   };

   struct WRITE4args {
           stateid4        stateid;
           offset4         offset;
           count4          count;
           stable_how4     stable;
           opaque          data<>;
   };

   struct WRITE4resok {
           count4          count;
           stable_how4     committed;
           writeverf4      verf;
   };

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

   /*
    * SECINFO: Obtain Available Security Mechanisms
    */
   struct SECINFO4args {
           filename4       name;
   };

   struct rpc_flavor_info {
           secoid4         oid;
           qop4            qop;
           rpc_gss_svc_t   service;
   };
```

```
   struct secinfo4 {
           rpc_flavor4     flavor;
           rpc_flavor_info *flavor_info;
           secinfo4        *nextentry;
   };

   struct SECINFO4resok {
           secinfo4        reply;
   };


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

   enum opcode {
           OP_NULL                 = 0,
           OP_ACCESS               = 1,
           OP_CLOSE                = 2,
           OP_COMMIT               = 3,
           OP_GETATTR              = 4,
           OP_GETFH                = 5,
           OP_LINK                 = 6,
           OP_LOCK                 = 7,
           OP_LOCKT                = 8,
           OP_LOCKU                = 9,
           OP_LOOKUP               = 10,
           OP_LOOKUPP              = 11,
           OP_NVERIFY              = 12,
           OP_OPEN                 = 13,
           OP_PUTFH                = 14,
           OP_PUTROOTFH            = 15,
           OP_READ                 = 16,
           OP_READDIR              = 17,
           OP_READLINK             = 18,
           OP_REMOVE               = 19,
           OP_RENAME               = 20,
           OP_RENEW                = 21,
           OP_RESTOREFH            = 22,
           OP_SAVEFH               = 23,
           OP_SECINFO              = 24,
           OP_SETATTR              = 25,
           OP_SETCLIENTID          = 26,
           OP_VERIFY               = 27,
           OP_WRITE                = 28
```

```
   };

   union opunion switch (unsigned opcode) {
    case OP_NULL:          void;
    case OP_ACCESS:        ACCESS4args opaccess;
    case OP_CLOSE:         CLOSE4args opclose;
    case OP_COMMIT:        COMMIT4args opcommit;
    case OP_GETATTR:       GETATTR4args opgettattr;
    case OP_GETFH:         void;
    case OP_LINK:          LINK4args oplink;
    case OP_LOCK:          LOCK4args oplock;
    case OP_LOCKT:         LOCK4args oplockt;
    case OP_LOCKU:         LOCK4args oplocku;
    case OP_LOOKUP:        LOOKUP4args oplookup;
    case OP_LOOKUPP:       void;
    case OP_NVERIFY:       NVERIFY4args opnverify;
    case OP_OPEN:          OPEN4args opopen;
    case OP_PUTFH:         PUTFH4args opputfh;
    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_VERIFY:        VERIFY4args opverify;
    case OP_WRITE:         WRITE4args opwrite;
   };

   struct op {
           opunion         ops;
   };

   union resultdata switch (unsigned resop){
    case OP_NULL:          void;
    case OP_ACCESS:        ACCESS4res op;
    case OP_CLOSE:         CLOSE4res opclose;
    case OP_COMMIT:        COMMIT4res opcommit;
    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_PUTFH:          PUTFH4res opputfh;
    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_VERIFY:         VERIFY4res opverify;
    case OP_WRITE:          WRITE4res opwrite;
   };

   struct COMPOUND4args {
           utf8string      tag;
           op              oplist<>;
   };

   struct COMPOUND4resok {
           utf8string      tag;
           resultdata      data<>;
   };

   union COMPOUND4res switch (nfsstat4 status){
    case NFS4_OK:
            COMPOUND4resok resok;
    default:
            void;
   };


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

```
                    COMPOUND4res
                            NFSPROC4_COMPOUND(COMPOUND4args) = 1;

            } = 4;
    } = 100003;
```

16. Bibliography

[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.

[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", RFC1094, March 1989.

http://www.ietf.org/rfc/rfc1094.txt

[RFC1345]
Simonsen, K., "Character Mnemonics & Character Sets", RFC1345, Rationel Almen Planlaegning, June 1992.

http://www.ietf.org/rfc/rfc1345.txt

[RFC1813]
Callaghan, B., Pawlowski, B., Staubach, P., "NFS Version 3 Protocol Specification", RFC1813, Sun Microsystems, Inc., June 1995.

http://www.ietf.org/rfc/rfc1813.txt

[RFC1831]
Srinivasan, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC1831, Sun Microsystems, Inc., August 1995.

http://www.ietf.org/rfc/rfc1831.txt

[RFC1832]
Srinivasan, R., "XDR: External Data Representation Standard", RFC1832, Sun Microsystems, Inc., August 1995.

http://www.ietf.org/rfc/rfc1832.txt

[RFC1833]
Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC1833, Sun Microsystems, Inc., August 1995.

http://www.ietf.org/rfc/rfc1833.txt

[RFC2054]
Callaghan, B., "WebNFS Client Specification", RFC2054, Sun Microsystems, Inc., October 1996

http://www.ietf.org/rfc/rfc2054.txt

[RFC2055]
Callaghan, B., "WebNFS Server Specification", RFC2054, Sun Microsystems, Inc., October 1996

http://www.ietf.org/rfc/rfc2055.txt

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

http://www.ietf.org/rfc/rfc2078.txt

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

http://www.ietf.org/rfc/rfc2152.txt

[RFC2203]
Eisler, M., Chiu, A., Ling, L., "RPCSEC_GSS Protocol Specification", RFC2203, Sun Microsystems, Inc., August 1995.

http://www.ietf.org/rfc/rfc2203.txt

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

http://www.ietf.org/rfc/rfc2279.txt

[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.

[SPNEGO]
Baize, E., Pinkas, D., "The Simple and Protected GSS-API Negotiation Mechanism", draft-ietf-cat-snego-09.txt, Bull, April 1998.

ftp://ftp.isi.edu/internet-drafts/draft-ietf-cat-snego-09.txt

[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]
"Unicode Technical Report #8 - The Unicode Standard, Version 2.1", Unicode, Inc., The Unicode Consortium, P.O. Box 700519, San Jose, CA 95710-0519 USA, September 1998

http://www.unicode.org/unicode/reports/tr8.html

[Unicode2]
"Unsupported Scripts" Unicode, Inc., The Unicode Consortium, P.O. Box 700519, San Jose, CA 95710-0519 USA, October 1998

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

17. Authors and Contributors

General feedback related to this document should be directed to:

nfsv4-wg@sunroof.eng.sun.com

or the editor.

17.1. Contributors

The following individuals have contributed to the document:

Carl Beame, beame@bws.com, of Hummingbird Communications Ltd.

17.2. Editor's Address

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

Phone: +1 512-349-9376
E-mail: shepler@eng.sun.com

17.3. Authors' Addresses

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

Phone: +1 650-786-5067
E-mail: brent.callaghan@eng.sun.com

Mike Eisler
Sun Microsystems, Inc.
5565 Wilson Road
Colorado Springs, CO 80919

Phone: +1 719-599-9026
E-mail: mre@eng.sun.com

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

Phone: +1 650-786-5088
E-mail: david.robinson@eng.sun.com

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

Phone: +1 650-786-5096
E-mail: robert.thurlow@eng.sun.com

18. Full Copyright Statement