NFSv4 Multi-Domain Access
draft-adamson-nfsv4-multi-domain-access-04

Abstract

   The Network File System, version 4 (NFSv4) uses a representation of
   identity that allows the use of users and groups from multiple,
   distinct administrative domains, and NFSv4 allows the use of security
   mechanisms that authenticate principals from multiple, distinct
   administrative domains. This document describes methods by which
   NFSv4 clients and servers can handle principals, users, groups from
   multiple administrative domains.

Status of this Memo

Copyright Notice

carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Requirements notation

    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 [RFC2119].

2. Introduction

   The NFS Version 4 [RFC3530] protocol enables the construction of a
   distributed file system which can join NFSv4.0 or NFSv4.1
   [I-D.ietf-nfsv4-minorversion1] servers from multiple administrative
   domains, each potentially using separate name resolution services and
   separate security services, into a common multi-domain name space.

   NFSv4 deals with two kinds of identities: authentication identities
   (referred to here as "principals") and authorization identities
   ("users" and "groups" of users). NFSv4 supports multiple
   authentication methods, each authenticating an "initiator principal"
   (typically representing a user) to an "acceptor principals" (always
   corresponding to the server). NFSv4 does not prescribe how to
   represent authorization identities on file systems. All file access
   decisions constitute "authorization" and are made by servers using
   information about client principals (such as username, group
   memberships, and so on) and file metadata related to authorization,
   such as a file's access control list (ACL).

   Authentication in NFSv4 occurs at the the RPCSEC_GSS [RFC2203] layer
   where GSS-API mechanisms [RFC2743] are used to authenticate users on
   NFSv4 clients to NFSv4 servers, and to provide security services such
   as confidentiality and integrity protection for the protocol's
   messages. The NFSv4 protocol specifies no particular representation
   for authentication identities as these are entirely GSS-API
   mechanism-specific.

   Authorization for file object access is done at the NFSv4 protocol
   layer (i.e., above the RPCSEC_GSS layer), on the server side, based
   on an authenticated client principal's authorization context and the
   authorization meta-data of the file system objects that the client
   wishes to access. File authorization meta-data is set and retrieved
   in the NFSv4 RPC [RFC1831] layer, specifically via the object's
   owner, owner_group and acl, dacl and sacl attributes (the last three
   being ACLs). ACLs are lists of ACL entries (ACEs). Each ACE has a
   "who" field identifying a subject to whom some permission is granted
   or denied. The owner and owner_group attributes and the who ACE
   field, all reference users and groups. On the wire, the protocol
   represents users and groups as strings of characters with this form:
   name@domain, where <name> is a user or group name, and <domain> is a
   the name of an administrative domain, more specifically a DNS
   [RFC1034] domainname.

   NFSv4 server implementations usually do not, and really ought not,
   store authorization identities on disk in the same form as is used on
   the wire. The reason is that users' and groups' names change all too
   often, while searching for and updating file authorization meta-data

after a user/group name change is not trivial, particularly in a global namespace spanning multiple administrative domains.

NFSv4 servers therefore must perform two kinds of mappings:

1. Between the authentication identity and the authorization context (a principal's user ID, group memberships, etcetera)

2. Between the on-the-wire authorization identity representation and the on-disk authorization identity representation.

Many aspects of these mappings are entirely implementation-specific, but some require name resolution services, and in order to interoperate servers must use such services in compatible ways. Many implementations are limited to being able to represent users and groups from a single domain.

In this document we address both of those kind of mappings, describing possible implementation strategies, and specifying a name service for interoperation in a global namespace [I-D.ietf-nfsv4-federated-fs-reqts].

3. Background

   NFSv4 uses a syntax of the form "name@domain" to represent, on the
   wire, the NFSv4 ACL name for users and groups. This design provides
   a level of indirection that allows a client and server with different
   internal representations of authorization identity to interoperate
   even when referring to authorization identities from different
   administrative domains.

   Multi-domain capable sites need to meet certain requirements in order
   to ensure that clients and servers can map name@domain to internal
   representations reliably:

   o The name portion of name@domain MUST be unique within the
      specified DNS domain.

   o Every local representation of a user and of a group MUST have a
      canonical name@domain, and it must be possible to return the
      canonical name@domain for any identity stored on disk, at least
      when required infrastructure servers (such as name services) are
      online.

   As described in [I-D.ietf-nfsv4-minorversion1] section 2.2.1.1 "RPC
   Security Flavors":

         NFSv4.1 clients and servers MUST implement RPCSEC_GSS. (This
         requirement to implement is not a requirement to use.) Other
         flavors, such as AUTH_NONE, and AUTH_SYS, MAY be implemented
         as well.

   The AUTH_NONE security flavor can be useful to the multi-domain NFSv4
   or federated name space to grant universal access to public data
   without any credentials.

   The AUTH_SYS security flavor uses a host-based authentication model
   where the [weakly-authenticated] client asserts the user's
   authorization identities using small integers as user and group
   identity representations. Because of the small integer authorization
   ID representation, AUTH_SYS can only be used in a name space where
   all clients and servers share a uidNumber and gidNumber translation
   service. A shared translation service is required because uidNumbers
   and gidNumbers are passed in the RPC credential; there is no
   negotiation of namespace in AUTH_SYS. Collisions can occur if
   multiple translation services are used. These and other issues are
   addressed in [I-D.williams-rpcsecgssv3] which describes a new version
   of RPCSEC_GSS that includes a modernized replacement for AUTH_SYS.

4. Terminology

    Identity: a way to refer to a user or group.

    Principal: an entity that is authenticated by RPCSEC_GSS (usually,
    but not always, a user; rarely, if ever, a group; sometimes a
    host).

    Authorization context: the set of user and group IDs, privileges,
    labels, and other items relevant to authorization, corresponding
    to a subject (user or principal).

    Domain-local ID: Most installations assign numeric, local
    identifiers to users and groups, using a namespace local to their
    domain. We call this a domain-local ID.

    Local representation of identity: an item such as a POSIX user
    IDentifier (UID) or group ID (GID), or a Windows Security
    IDentifier (SID), or other such representation of a user or a
    group of users. These can be local to a single host.

    Global representation of identity: a tuple consisting of a domain
    identifier (possibly the domain's name itself) and a domain-local
    user/group ID. We do not propose a standard global representation
    of identity, but the concept is useful. [NEEDSWORK: we refer to
    the global representation form in the RPCSEC_GSS PAC]

    Group: a security entity representing zero, one or more users and,
    possibly, other groups. Can appear as the subject of an ACE.

    Domain: a set of users, groups and computers administered by a
    single entity, and identified by a DNS domain name.

    POSIX IDs: small non-negative integer (typically $0..2^{31}$ or
    $0..2^{32}$) identifiers. The namespace of user IDs (UIDs) is
    distinct from the namespace of group IDs (GIDs).

    Windows SIDs: an identifier of security entities, including users
    and groups. The form of a SID is: S-1-<authority>-<RID_0>-<RID_1>
    -<RID_n> By convention some authority numbers denote security
    entities, identified by RID_n, local to a domain identified by
    RIDs 0..n-1. Domain RIDs are usually generated randomly within a
    "forest" of domains.

    Name resolution: mapping from {domain, name} to {domain, ID}, and
    vice-versa via lookups. Can be applied to local or remote
    domains.

ID mapping: {remote domain, remote domain-local ID} to {local representation of ID} mappings.

5. Local Representations of Global Identity

   Multi-domain support starts at the fileserver where local ID forms
   need to be able to represent global identities from both local and
   remote domains. Local representation of global identity also applies
   to clients, particularly clients with local filesystems. There's a
   range of local solutions to this multi-domain ID representation
   problem. In this section we describe several approaches to
   representing a <name>@<local or remote domain> on disk. None of
   these approaches are REQUIRED; all are INFORMATIVE. However,
   conventions relating to the use of name services are NORMATIVE.

5.1. Storing Name@Domainname

   One simple approach to the multiple domain problem is to store the
   name@domain on disk.

   This approach imposes a severe constraint on the administrators of
   these domains: user and group names must never be reused, as there is
   also no realistic way to keep the name@domain on disk representation
   up to date with user, group or domain renames and removals. Consider
   a remote domain's NFSv4 servers where real-time employee join/leave
   data may be (typically is!) considered privileged, and remote servers
   may not be sufficiently privileged to access it [NEEDSWORK].

5.2. Storing Remote-ID@Domainname

   Most installations assign numeric, local identifiers to users and
   groups, using a namespace local to their domain. We call this a
   domain-local ID. We can then construct a global identity form
   consisting of a domain name and a domain-local user/group ID.

   The user or group renaming issue can be addressed by using a global
   identity form where domain-local IDs are required. I.e., use name
   resolution to lookup name@domain to find the ID local to the
   specified domain, and join the ID with the specified domain name.
   This function still has a renaming problem with respect to DNS domain
   renames, but that is a more realistically manageable problem than the
   user/group renaming problem.

5.2.1. Storing Remote-ID@Domain-ID

   The DNS domain renaming issue in the previous section can be
   addressed by assigning and publishing a unique ID to each DNS domain.
   I.e., use name resolution to lookup name@domain to find some ID local
   to the domain, lookup the domain ID and store <remote-ID,domain-ID>.
   The Windows Security Identifier (SID) is an example of this form.

5.3. ID Mapping

   Many file systems exported by NFS only store 32-bit user and group
   IDs which limit their ability to utilize the on disk representation
   described in Section 5.2. Such systems may need to use an additional
   service to map between <remote user ID, local user IDs> and <remote
   group IDs, local group IDs>. We call this an "ID mapping service".

   The use of an ID mapping service is not strictly necessary if the
   system operates on IDs large enough and in a known format such that
   <user/group ID, domain ID> can be parsed and encoded into a native
   ID. However, a large class of operating systems, those which are
   Unix or Unix-like operating systems, such as Solaris and Linux, use
   32-bit UIDs and GIDs in many interfaces and therefore need mapping
   for backwards compatibility reasons.

   One example of such a service is to keep a local or distributed
   database for dynamically assigning a local 32 bit ID to every <ID>@
   <domain-ID>, or one could do that only for remote domains, reserving
   only a small part of the local 32-bit ID namespace for remote
   domains' users/groups.

   The remote ID and remote domain are then used as inputs to a name
   resolution service which contacts the remote domain name service to
   resolve the remote name.

5.4. Use of Name Services

   File systems often use a distributed directory service for resolving
   domain local 32 bit IDs to users and groups. The Network Information
   Service [NIS] and the Lightweight Directory Access Protocol [RFC4511]
   are the two broadly deployed distributed directory service protocols
   used for this purpose. LDAP is used instead of NIS in environments
   where scalabilty, security and/or extensibility are desired.
   Section 8.2 expands the LDAP protocol to include mappings between
   principals and local user and group IDs.

   Support for LDAP [RFC4511] with the RFC2307 schema [RFC2307] is
   REQUIRED.

5.4.1. Using LDAP with RFC2307 Schema

   Name resolution consists of searches with scope 'sub', a base DN
   corresponding to a domain (more on this below) and a filter of either
   of these forms, with matching on objectClass being optional:

    o (objectClass=posixAccount)(uid="<username>)

    o (objectClass=posixGroup)(cn="<groupname>)

    o (objectClass=posixAccount)(uidNumber="<UID>)

    o (objectClass=posixGroup)(gidNumber="<GID>)

    The base DN SHOULD be formatted from a domain's DNS domainname as
    follows. First format the domainname as a string, then strip the
    trailing dot ('.'), if any, then replace all dots ('.') with ",DC=",
    then prepend "DC=" to the resulting string. For example,
    foo.bar.example becomes "DC=foo,DC=bar,DC=example". This convention
    is REQUIRED to be implemented. Domains with base DNs that do not
    match this convention MAY be used, but their domainname-to-base-DN
    mappings must be published where NFSv4 clients and servers may find
    them; we provide no conventions for publishing such mappings. We
    RECOMMEND that LDAP referrals be used to publish such mappings (e.g.,
    the client does an LDAP search using "DC=foo,DC=example" as the base
    DN and gets a referral that includes the correct non-standard base DN
    for "foo.example").

    Client and server implementations MUST support the use of LDAP
    referrals to find LDAP servers authoritative for any given base DN.

    For example, to resolve a user named joe@foo.bar.example to a remote
    ID a system would do an LDAP search with DC=foo,DC=bar,DC=example as
    the base DN, scope='sub' and with a filter of
    (objectClass=posixAccount)(uid="<username>) looking for the uidNumber
    attribute.

5.4.2. Using Active Directory LDAP

    [NEEDSWORK: Add text describing searches by which to resolve
    name@domain to SIDs and vice versa.]

5.4.3. Mapping Domain Names to Domain IDs

    [NEEDSWORK: Add text on mapping domainnames to domain IDs. Note that
    Windows SID does this.]

    We need to have a common way to map Domain Names to Domain IDs to
    enable mult-domain numeric IDs as described in Section 5.2.1.
    Currently we have two suggestions:

    1. Just use SIDs, first asking MSFT to allocate a suitable authority
        for non-Windows domain SIDs.

   2. - Store 96-bit numeric IDs which means we:

       * cast those to domain SIDs later

       * define a non-SID large ID format. This is a fine fallback
          should MSFT be unwilling to assign authority numbers for this
          purpose

6. Resolving Cross-Domain Authorization Information

   In order to authorize client principal access to files, the NFS
   server must map the RPCSEC_GSS client principal name or the
   underlying GSS-API security context to authorization information
   including a local ID, a set of local group IDs and other local user
   privileges meaningful to the file system being exported.

   In the cross-domain case where a client principal is seeking access
   to files on a server in a different NFSv4 domain, the NFS server
   needs to obtain, in a secure manner, the authorization information
   from an authoritative source: e.g. a directory service in the client
   principals NFS domain.

   There are several methods the cross-domain authoritative
   authorization information can be obtained:

   1. A mechanism specific GSS-API authorization payload containing
       credential authorization data such as a "privilege attribute
       certificate" or PAC.

   2. An NFS server local domain directory query when there is a
       security agreement between the two cross-domain directory
       services plus regular update data feeds so that the NFS server
       local domain directory service is authoritative for the client
       prinicpal domain.

   3. A direct query from the NFS server to the client principal
       authoritative directory service.

   The authorization data information SHOULD be obtained via the GSS-API
   name attribute interface [I-D.ietf-kitten-gssapi-naming-exts] either
   via a single attribute for the credential authorization data or via
   discrete GSS-API name attributes corresponding to the authorization
   data elements described in Section 6.1. Details for those attributes
   are TBD.

   Note that the retrieval of attribute values used by the GSS-API name
   attribute interface implementation could utilize any of the above
   mentioned methods of obtaining the authorization information.

   If the named attribute interface is not available, or the attributes
   are not available, other means of determining a principal's
   authorization data SHOULD be used, such as those described in
   Section 6.2 and Section 6.3.

6.1. Credential Authorization Data

   Here we list in more detail the authorization information that an
   NFSv4 server needs in order to make a file access decision. The
   credential authorization data contains the user and group IDs
   corresponding to the client principal, in global representation of
   identity form. Note that the server may need to map the global IDs
   to local IDs as described in Section 5.3.

   The ability to map IDs to the name@domain form is required for the
   NFSv4 server to be able to respond to file authorization meta-data
   (ACL) set and retrieve requests.

   Credential authorization data consists of:

   o UserID: This field contains the principal's global ID and/or local
      ID mapping thereof, and the name@domain form thereof.

   o PrimaryGroupID: This field contains the global ID and/or local ID
      mapping thereof for the principal's primary group, and the
      name@domain form thereof.

   o Groups: This field contains an array of group IDs for the groups
      that the user is a member of, in global ID form and/or local ID
      mappings thereof, as well as in name@domain forms.

   o Optional field(s) for privileges and authorizations granted to the
      principal, if any.

   o Optional field(s) for other privilege information such as the
      multi-level security label range/set of the principal.

   o Optional implementation-specific items relevant to authorization.

6.2. Using Credential Authorization Data

   Authorization context information can sometimes be obtained from the
   credentials authenticating a principal; the GSS-API represents such
   information as attributes of the initiator prinicpal name. For
   example: Kerberos 5 [RFC4120] has a method for conveying
   "authorization data", both client-asserted as well as KDC-
   authenticated authorization data, and one KDC implementation uses
   this feature to convey a "privilege attribute certificate" (PAC)
   listing the principal's user and group "security identifiers" (SIDs).
   Another example is the Kerberos General PAC
   [I-D.sorce-krbwg-general-pac] which lists the principal's user and
   group "universal user identifiers" (UUIDs) as well as their string
   representations and DNS domains. PKIX [RFC5280] certificates allow

   for extensions that could be used similarly.

6.2.1. Using a PAC

   The Windows operating system uses an authorization context called a
   "PAC" [PAC], which contains a user Security IDentifier (SID) and a
   list of group SIDs. Some Kerberos Key Distribution Centers (KDCs),
   notably Windows KDCs, issue Kerberos Tickets with PACs as Kerberos
   authorization data.

   Some KDCs (will) issue Kerberos Tickets with the General PAC
   [I-D.sorce-krbwg-general-pac] as authorization data. The General PAC
   authorization data MUST be authenticated in the sense that its
   contents must come from an authenticated, trusted source, such as a
   directory server or the issuer of the client principal's credential.

   When a client principal is authenticated using such a ticket, the
   server SHOULD extract the PAC from the client's ticket and map, if
   need be, the SIDs or UUIDs in the PAC to local ID representations.

   The authorization context information in a PAC can be considered a
   single, authenticated, discrete GSS-API name attribute, in which case
   the server must parse it into its individual elements.

6.3. Using Directory Services

   If suitable and sufficient authenticated GSS-API name attributes for
   the client principal are not available, then the server may try to
   map the client principal name to a local notion of user account, and
   then lookup that user account's authorization context information
   through authenticated name service lookups.

6.3.1. Mapping Principal Names to Username

   One simple method for Kerberos principal-to-username mapping is to
   first apply an algorithmic or table-based Kerberos client principal
   realm name to domain name mapping, then a client principal name to
   username mapping. Finally, the server can look up the user's
   authorization context using the user's domain's name services.

   A trivial Kerberos realm-name-to-domainname mapping consists of using
   the realm name as the domainname. [NEEDSWORK: Add notes about
   internationalization.] Servers SHOULD implement this mapping as an
   option, possibly as a default option.

   A trivial Kerberos principal name to username mapping for 1-component
   principal names is to use the principal name, unmodified, as the
   username. Servers SHOULD implement this mapping as an option,

possibly as a default option.

6.3.2. Using a Name Service to Map Principal Names to User Accounts

   Name services such as the Solaris gsscred database where the local
   identity is looked up in a database keyed by the GSS exported name
   token, or LDAP with the extension described in Section 8.2, can be
   used to map principal names to user accounts.

7. Multi Domain User Group Membership Determination

   User group membership is easy to determine for users in a system's
   local domain: the operating system will already know how to do that.
   For users in remote domains, the authentication service may provide
   group membership information. If not, we need methods for group
   memebership determination.

   [NEEDSWORK:

   1. provide a[n obviously limited] mode of operation that depends
       only on RFC2307 and therefore does not support group nesting;

   2. provide a more full-fledged mode of operation that depends
       RFC2307bis;

   3. provide a more full-fledged mode of operation that depends AD's
       schema.]

   User group membership in remote domain's groups, and/or for remote
   users, may be determined using LDAP with the RFC2307 schema. The
   RFC2307 schema does not define the values of the 'memberUid'
   attribute, but in practice it seems that those are expected to be the
   names of users as found in the 'uid' attribute of 'posixAccount'
   entries. There is work in progress to update RFC2307
   [I-D.howard-rfc2307bis] to allow the use of DNs in the member
   attribute. Group nesting is also enabled.

   Assuming the ability to store DNs in the member attribute, then,
   group membership determination can be done as follows. Given a user
   ID whose DN has been determined:

   1. Search the user's domain for groups that the user is a member of
       in the user's home domain. Since we cannot assume a domain is
       authoritative for another domains group membership, filter out
       groups that are not local to the user's home domain.

   2. Search the server's domain for groups that the user is a member
       of in the server's home domain.

   3. Search the server's domain for groups that the user's group
       memberships determined in steps 1 and 2 are members of.

   4. Continue searching for nested group memberships given the list of
       groups from steps 2 and 3 while being careful to detect or
       prevent loops.

   However, the above procedure has the same user/group name renaming

issue. By skipping step 2 we can get down to just a group renaming
issue. To fully address the rename issue we need either a new
attribute or value type for memberUid, storing user/group IDs in some
global ID representation.

[NEEDSWORK: Add text defining such a new attribute/value type.]

8. LDAP and Multi-Domain NFSv4

    Each of the three methods of retrieving cross-domain authorization
    information described in section Section 6 can require a directory
    service query. Cross-domain querys are not only inefficient, but
    also implies knowledge of multiple systems where two different
    domains rely on completely different infrastructures for user
    information.

    [NEEDSWORK: Describe why LDAP is REQUIRED]

8.1. LDAP Service Discovery

    [NEEDSWORK: this is just an idea place holder.]

    Two potential methods:

    1. Use local methods (configuration, DNS SRV RR lookups, ...) to
        discover local domain's servers, then depend on LDAP referrals
        for discovering all other domains servers.

    2. Use DNS SRV RRs much the way AD does

    NICO: I would prefer that we have one REQUIRED to implement service
    discovery mechanism as follows:

    o   specify local DS discovery using DNS SRV RR lookups much like AD
        does (i.e., have a label to indicate the purpose of the LDAP
        service, not just _ldap). Make this general enough that clients
        could discover DSes of remote domains on their own.

    o   use LDAP referrals (and DNS resolution of the host parts of the
        referrals) to discover DSes of other domains.

    The main benefit of this mechanism is that we can leave the work of
    finding topologically-close caches and/or authoritative servers to
    the clients' local DSes, thus avoiding the need to deal directly with
    topology in our spec.

8.2. LDAP Attribute for Principal Name to Local ID Translation

    The gSSPrincipal objectclass allows for the use of the gSSAuthName
    attribute described in the following section.

                        objectclass (1.3.6.1.4.1.250.10.7
                        NAME 'gSSPrincipal'
                        DESC 'GSS Principal Name'
                        SUP posixAccount

                         MAY (gSSAuthName))

   The gSSAuthName attribute provides a method for the translations
   between a posixAccount and (multiple) GSS-API security principals,
   used as described in Section 6.3.1.

   The gSSAuthName attribute stores a user's GSS-API principal name in
   exported name token form (see [RFC2743]).

                         attributetype (1.3.6.1.4.1.250.10.6
                         NAME ('gSSAuthName')
                         DESC 'GSS-API exported principal name
                         exported token'
                         EQUALITY bitStringMatch
                         SYNTAX 1.3.6.1.4.1.1466.115.121.1.6)

8.3. Name Resolution and LDAP Caching

   As noted in Section 5.2, most local representations require a name
   service to perform ID to name translations. Implementations are
   REQUIRED to support the use of LDAP as a name service, relying on
   LDAP referrals for federated namespace construction.

   Note that in a topographically widely separated set of domains the
   need to do name service lookups in various domains' name services may
   prove brittle, resulting in non-deterministic server behavior (e.g.,
   sometimes a user can access share, sometimes they cannot; sometimes
   they appear to be members of some group, sometimes they do not). To
   avoid this, site administrators may wish to maintain local caches of
   remote domains' name services such that LDAP searches for users/
   groups in remote domains can be satisfied locally for some set of key
   attributes (such as naming and ID attributes), with referrals used in
   all other cases.

   Domains in a federated namespace may provide each other with LDAP
   LDIF delta feeds by which to maintain cached LDAP contents up to
   date.The LDAP DN hierarchy described in Section 5.4.1 has the
   advantage of aiding delta feeds from remote domains where each
   domain's information is in its own DN subtree.

9. Security Considerations

   Caching of remote domains' LDAP search results persents some security
   considerations. For example, some attributes' values may not be
   visible unless a user's credentials are used. Some attributes'
   values may not be intended to be visible to users, but to hosts.
   Caching servers MUST be capable of issuing referrals as needed for
   attributes whose values they may not read. Some domain federations
   will want to have their domains trust each others' caching servers.

   More considerations to come

10. References

10.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC1034] Mockapetris, P., "Domain names - concepts and facilities",
              STD 13, RFC 1034, November 1987.

   [RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
              Beame, C., Eisler, M., and D. Noveck, "Network File System
              (NFS) version 4 Protocol", RFC 3530, April 2003.

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

   [RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol
              Specification", RFC 2203, September 1997.

   [RFC2743] Linn, J., "Generic Security Service Application Program
              Interface Version 2, Update 1", RFC 2743, January 2000.

   [RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The
              Kerberos Network Authentication Service (V5)", RFC 4120,
              July 2005.

   [RFC4511] Sermersheim, J., "Lightweight Directory Access Protocol
              (LDAP): The Protocol", RFC 4511, June 2006.

   [RFC2307] Howard, L., "An Approach for Using LDAP as a Network
              Information Service", RFC 2307, March 1998.

   [I-D.howard-rfc2307bis]
              Howard, L. and H. Chu, "An Approach for Using LDAP as a
              Network Information Service", draft-howard-rfc2307bis-02
              (work in progress), August 2009.

   [I-D.sorce-krbwg-general-pac]
              Sorce, S., Yu, T., and T. Hardjono, "A Generalized PAC for
              Kerberos V5", draft-sorce-krbwg-general-pac-01 (work in
              progress), December 2010.

   [I-D.ietf-nfsv4-federated-fs-reqts]
              Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M.
              Naik, "Requirements for Federated File Systems",
              draft-ietf-nfsv4-federated-fs-reqts-06 (work in progress),
              October 2009.

   [I-D.ietf-kitten-gssapi-naming-exts]
             Williams, N. and L. Johansson, "GSS-API Naming
             Extensions", draft-ietf-kitten-gssapi-naming-exts-09 (work
             in progress), February 2011.

   [I-D.ietf-nfsv4-minorversion1]
             Shepler, S., Eisler, M., and D. Noveck, "NFS Version 4
             Minor Version 1", draft-ietf-nfsv4-minorversion1-29 (work
             in progress), December 2008.

   [PAC]   Brezak, J., "Utilizing the Windows 2000 Authorization Data
             in Kerberos Tickets for Access Control to Resources",
             October 2002.

10.2. Informative References

   [I-D.williams-rpcsecgssv3]
             Haynes, T. and N. Williams, "Remote Procedure Call (RPC)
             Security Version 3", draft-williams-rpcsecgssv3-01 (work
             in progress), March 2011.

   [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S.,
             Housley, R., and W. Polk, "Internet X.509 Public Key
             Infrastructure Certificate and Certificate Revocation List
             (CRL) Profile", RFC 5280, May 2008.

   [NIS]   Sun Microsystems, "System and Network Administration",
             March 1990.

Authors' Addresses

    William A. (Andy) Adamson
    NetApp

    Email: andros@netapp.com

    Kevin Coffman
    CITI, University of Michigan

    Email: kwc@umich.edu

    Nicolas Williams
    Oracle

    Email: nicolas.williams@oracle.com

                          NFS Pathless Objects
                draft-dipankar-nfsv4-pathless-objects-02.txt

   Abstract

      This document describes a set of NFS operations for creating,
      maintaining and searching filesystem objects independent of the
      traditional hierarchical namespace.

   Requirements Language

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

   Status of this Memo

      This Internet-Draft is submitted in full conformance with the
      provisions of BCP 78 and BCP 79.

      Internet-Drafts are working documents of the Internet Engineering
      Task Force (IETF). Note that other groups may also distribute
      working documents as Internet-Drafts. The list of current Internet-
      Drafts is at http://datatracker.ietf.org/drafts/current/.

      Internet-Drafts are draft documents valid for a maximum of six months
      and may be updated, replaced, or obsoleted by other documents at any
      time. It is inappropriate to use Internet-Drafts as reference
      material or to cite them other than as "work in progress."

      This Internet-Draft will expire on October 9, 2011.

carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Terminology

   NFS: Used to refer to Network File System irrespective of the
   version.

   NFSv3: Network File System Version 3

   NFSv4: Network File System Version 4

   NFSv4.1: Network File System Version 4.1

   ACL: Access Control List

   HTTP: Hyper Text Transfer Protocol

   REST: Representational State Transfer

2. Introduction

   The NFS protocol is presently capable of interacting with objects
   which can be represented by a pathname and a filehandle, residing in
   a hierarchical namespace exported by the NFS server. However, such a
   hierarchical namespace which tries to resemble the UNIX filesystem
   layout and interface imposes restrictions on the filesystem object
   locations and does not scale well in the case we need to store
   billions of files inside a flat directory structure.

   The rapidly developing distributed web applications of today, such as
   those implementing the HTTP REST protocol, need to store billions of
   objects, which do not need any directory hierarchy and instead must
   have the capability to specify custom object attributes and quickly
   search for the objects based on these attributes. To facilitate
   this, an object needs to become independent of the filesystem
   directory hierarchy that has been mandated by the NFS server until
   now. In other words, the requirement is to have filesystem objects
   which can be created, queried for and destroyed without being
   associated with a pathname. These objects do not need to become
   visible under a standard NFS exported hierarchical pathname but need
   to be looked up based on tags or custom attributes. This RFC
   presents the operations that are required by the NFS protocol to
   implement such a feature of pathless filesystem objects.

   Separation of the pathname from the filesystem object provides the
   implementation greater flexibility on where to store the object,
   which can lead to an optimal distribution of the filesytem objects
   based on application requirements. Such an filesystem object is
   looked up by the client based on the attributes of the object, rather

than it's location in a directory, and is accessed by it's NFS
filehandle directly. This new object, though it does not have a
pathname, can still optionally have a name, which MAY be implemented
as an attribute for the object. The existing set of NFS attributes
needs to be extended to support a model where lookup of filesystem
objects can be done based on attributes.

With the introduction of NFSv4, the NFS protocol enforces
statefulness for interacting with filesystem objects. There are many
applications which require the stateful model of NFSv4. But there
are also many web oriented object stores, which can be simultaneously
accessed over other stateless protocols such as http, ftp etc. and
hence are not very interested in statefulness. Rather, they would
like to have the flexibility of using the stateless nature of NFSv3
along with some interesting features of NFSv4 such as ACLs, Named
Attributes, Compound Operations etc. which do not depend on the
statefulness of the NFS server for functionality. Stateless
operations provide the benefit of better performance as functional
and maintenance costs for the implementations are significantly less.
The object stores which will potentially handle billions of objects
and have no need for state maintenance can greatly benefit from the
improved performance of a stateless NFS implementation.

In light of the above, anonymous states are RECOMMENDED to be used
with this RFC, which means a stateid of all zeroes SHOULD be used for
NFSv4 and NFSv4.1 READ, WRITE and SETATTR operations on pathless
objects. At the same time, also keeping in mind the requirements of
applications which need to maintain locks at the server, advisory
locking SHOULD be supported. Delegations and shared locking support
is OPTIONAL with the implementation of this RFC.

3. Protocol Overview

3.1. Pathless Objects and Object Sets

   To create an object independent of a pathname, the client sends a
   request to the server to create the object without specifying any
   name or pathname and the server returns the NFS filehandle for the
   object thus created. A new object type called NF4NOPATHOBJ is
   defined in this RFC, which SHOULD be used to create pathless objects.
   Since pathless objects cannot be looked up based on pathname, a new
   type of attribute, called Search Attribute is defined in this RFC,
   which SHOULD be used to lookup the pathless objects. The NFSv4.1
   READ and WRITE operations SHOULD be used to perform I/O on pathless
   objects and attributes, including search attributes, can be set using
   SETATTR and retrieved using GETATTR operations. A pathless object
   SHOULD support the mandatory set of attributes defined in RFC 5661

for a filesystem object.

Along with creating objects which are independent of pathnames, it is REQUIRED to have a mechanism to classify together such objects, for accessibility and scoped access resolution. This RFC uses the term "Object Set" for representing such a collection of objects. An Object Set works as a container for pathless objects and MUST be defined and created before a pathless object is created. It is analogous to a directory, where the object is analogous to a file inside the directory. An Object Set contains objects and MAY OPTIONALLY also contain other Object Sets.

When the client has created an Object Set or has access to an existing Object Set within the server, it can create pathless objects that are contained in the Object Set. The pathless object, even though it is contained in a Set, can be physically located anywhere. It is not necessary to implement a pathless object as a file. It can be any physical entity, which has a unique identifier in the form of a NFS filehandle. So the filehandle serves as an unique identifier for the object and there is no requirement that it SHOULD represent a file. The server is REQUIRED to maintain the linkage between the object and it's Set and is free to distribute and store the objects in the best possible way to satisfy the needs of the application.

An Object Set is expected to have certain access primitives associated with it, which are used by the server to provide access control for the objects contained in the Set. The server can implement a policy based mechanism to grant specific clients or groups of clients access to an Object Set. Such an implementation can be analogous to the exports mechanism commonly used with the NFS exported directories. The NFS server MUST keep track of all the Object Sets it has. The server SHOULD make visible all the exported Object Sets to the clients, subject to access control policies at the server. This RFC does not pose any other requirements on the implementation of an access policy for an Object Set.

An Object Set MUST have a name, which MAY be implemented as an attribute of the Set. However, unlike the name of a pathless object, the name of an Object Set MUST be unique for the NFS server. When the client first creates an Object Set, it MUST specify a name for the Object Set. The server returns a filehandle for the Object Set to the client.

3.2. Object Root Filehandle

The NFS server supporting the Object Set and pathless object creation MUST also have a well known public filehandle, hereby named as "Object Root Filehandle", in short form objrootfh. This public

filehandle is required for the purpose of informing the client that
the server implements support for this RFC.The client SHOULD send an
operation named PUTOBJROOTFH to set the current filehandle to
objrootfh. The server which implements this RFC, MUST set the
current filehandle to objrootfh and return NFS4_OK. This objrootfh
filehandle SHOULD be different from the public filehandle that an
NFSv4 server supports under the PUTROOTFH operation. The Object Root
Filehandle serves as a master container for all the Object Sets. The
client can send a query to the server to list all the Object Sets
that are available to it for access under the Object Root Filehandle.
Such a query can be specified as {PUTOBJROOTFH, READDIR}, where the
requested attributes specified in the READDIR request would indicate
if the server should reply with the Object Sets under the current
filehandle.

3.3. Optional Features

Since the pathless objects MAY not be implemented as files, and this
RFC RECOMMENDS stateless operation as much as possible, the following
features are explicitly being made OPTIONAL:

1. Supporting the POSIX semantics for interaction with pathless
objects.

2. Specifying a name to create a pathless object. Note that Object
Sets MUST have unique names.

3. Support for either Exclusive Create or Soft and Hard links.

4. Support for non-regular filesystem objects such as device files.

5. Support for delegations and share locks

If links are implemented, it SHOULD link to the object based on the
object name attribute, rather than a pathname. So it is a matter of
having multiple values for the attribute name, which is already a
feature for the search attributes.

3.4. Interaction with stateful NFS operations

The pathless objects are RECOMMENDED to be stateless. As such, the
anonymous stateid of zero SHOULD be used for operations like READ,
WRITE, SETATTR etc. However, if a server wants to implement stateful
NFSv4.1 operations with pathless objects, it can do so given that it
conforms to the specifications of NFSv4.1 RFC. So, existing NFSv4
READ, WRITE operations will work with pathless objects without any
changes to the operation definitions as stated in NFSv4.1 RFC. The
NFSv4.1 locking model is applicable to pathless objects, but only

advisory locking MUST be supported. But if a server decides to
implement support for share locks and delegations, it MUST follow the
NFSv4.1 RFC locking semantics.

4. New file types

The pathless objects are not necessarily files or any other
filesystem object that can be defined with the existing nfs_fype4
type as specified in RFC 5661. The same holds for Object Sets. So
two new types are being introduced with this RFC, namely NF4NOPATH
and NF4OBJSET. So the definition of nfs_ftype4 is changed to include
the new file types and is as follows:

```
enum nfs_ftype4 {
    NF4REG      = 0x1;
    NF4DIR      = 0x2;
    NF4BLK      = 0x3;
    NF4CHR      = 0x4;
    NF4LNK      = 0x5;
    NF4SOCK     = 0x6;
    NF4ATTRDIR  = 0x7;
    NF4NAMEDATTR = 0x8;
    NF4NOPATHOBJ = 0x9;
    NF4OBJSET   = 0x10;
}
```


5. Search Attributes

5.1. Search Attributes Definition

In a hierarchical filesystem, the NFS client can do LOOKUP operations
based on pathname for a filesystem object but this will not work in
the case of pathless objects. So with pathless objects, the server
SHOULD support some kind of attributes which can be used to search
for such objects. These attributes are hereby called "Search
Attributes". These attributes have a name and a list of values. The
values can be of type integer or string. Search attributes can be
combined to form a query which looks up objects matching the
attributes specified in the query, as per the query semantics.

Two new attributes are added to the existing set of RECOMMENDED
attributes for NFSv4.1. One is a boolean attribute called
sattrsupport and the other is an array of strings called
srchattrlist. The sattrsupport denotes whether the server supports
search attributes. The srchattrlist contains the search attributes.

The GETATTR and SETATTR operations can be used to retrieve and set
the search attributes. The sattrsupport applies to the filesystem
and the srchattrlist applies to an object in that filesystem.

The basic data types used in this RFC are same as the data types
defined in RFC 5661 Section 3.2. Some new structured data types are
added in this section to define the Search Attributes. One is a type
specifier for the Search Attribute value i. whether it is a number or
a string and is defined as "svaltype". The "sval" represents a
single value for a Search Attribute, of type svaltype. The "svalist"
is a set of such values for the Search Attribute. The Search
Attribute itself is defined as "srchattr" and contains a name of type
component4, the svaltype and svalist. A collection of Search
Attributes is defined as srchattrlist.

```
+----------------+----+--------------+-----+
| Name           | Id | Data Type    | Acc |
+----------------+----+--------------+-----+
| sattrsupport   | 75 | bool         | R   |
| srchattrlist   | 76 | srchattr<>   | R W |
+----------------------------------------+
```

```
bool sattrsupport; /* indicates search attributes are supported */

enum svaltype {
    SVAL_TYPE_NUM = 0; /* Search Attribute value is a number */
    SVAL_TYPE_STR = 1; /* Search Attribute value is a string */
};

/* single search attribute value */
union sval switch (svaltype type) {
    case SVAL_TYPE_NUM:
        int64_t svalnum;
    case SVAL_TYPE_STR:
        component4 svalstr;
    default:
        void;
};

typedef struct sval svalist<>; /* array of attribute values */

struct srchattr {
    component4  srchattrname; /* name of the search attribute */
    svaltype    type;         /* type of the search attribute */
    svalist     srchvalist;   /* list of values for this attr */
};

typedef struct srchattr srchattrlist<>;
```

5.2. Search Attributes usage

   Since there is no pathname, we cannot use the NFSv4 LOOKUP operation
   to lookup a pathless object. Instead, the new search attributes are
   used to look up a pathless object. These are represented as name
   value pairs where the name is a string and the value is an array of
   numbers or strings. A search attribute can have multiple values for
   the same object. A search can be done for one or more of these
   values. For example, a pathless object can have the attribute name
   as "weather" and values can be "sunny", "cloudy", "rainy" etc. If
   there are no values specified for a search attribute, a search is
   made for the objects having the search attribute, with or without any
   values.

   Access control for a search attribute is governed by the access
   control for the corresponding object. The permissions to read or
   write the object's attributes apply to search attributes as well.

   To lookup pathless objects, the client sends a list of the search
   attributes. A new operation PUTSRCHATTR is added to lookup objects
   based on search attributes. The search attributes as well as the
   operations are applicable to both pathless objects and Object Sets.
   To retrieve or set search attributes, GETATTR and SETATTR are used.

   The PUTSRCHATTR operation MUST be used in conjunction with the
   READDIR operation to make use of the features provided by the READDIR
   operation, namely, a reply cursor, requested set of object attributes
   and maximum count of bytes in the reply. The PUTSRCHATTR operation
   MUST be immediately followed by a READDIR operation in the same
   COMPOUND operation to this effect. The client MUST request the
   object filehandles in the bitmap for requested attributes in the
   READDIR request. The READDIR reply contains the filehandles of all
   the objects matching the search attributes specified in PUTSRCHATTR.

   A special search attribute with srchattrname as "objname" of type
   SVAL_TYPE_STR MUST always be present for a pathless object and
   denotes the name that the object was created with. For Object Sets,
   this should have a unique value in the NFS server. For pathless
   objects it defaults to an empty string i.e. "".

5.3. Search Attributes Query

   The Search Attributes can be queried using the semantics defined in
   this section. A simple query is based on the Search Attribute name
   and on whether the Search Attribute matches a set of Search Attribute
   values. The match can be based on whether the Search Attribute name
   has a value that equals the value specified in the query or the match
   can also be based on whether the Search Attribute has a value lesser

than or greater than the value in the query. The lesser than and
greater than comparisons are more effective when the Search Attribute
has a svaltype of a number. Such queries can be logically joined or
chained together using logical primitives such as AND and OR. A NOT
primitive is also present to provide a logical negation of the query,
which will match those objects that do not match the Search Attribute
values specified in the query. Each query has a priority assigned to
it and queries with higher priority will execute earlier. For
example, let's say that there are 3 queries, which are joined like
this: query 1 AND query 2 OR query 3". Suppose query 2 and query 3
have a higher priority than query 1. So the server would execute
this query like this : query 1 AND (query 2 OR query 3), where
brackets indicate a logical grouping and execution of queries of the
same priority.

Some new structured data types are added in this section to define a
Search Attribute query. The type "srelation" determines what is the
nature of the match being done for the Search Attribute and it's
values i.e. whether a match is done for equals, lesser than or
greater than. The type "srchqueryjointype" specifies how two queries
can be logically chained together or if a query needs to be logically
negated. A Search Attribute query is defined as a combination of:

1. A srchattrlist, as defined in the section 5.1 of this RFC

2. A srelation, which specifies how the match for the Search
Attributes inside the srachattrlist and their corresponing values are
done.

3. A srchqueryjoinype called sqjtypenext which specified how the
query should be chained with the next query. The last query in a
chain of queries MUST have this set to SQUERY_NONE.

4. A priority which determines an ordering of the queries. A
priority of 0 SHOULD be considered as the highest priority, followed
by 1 and so on.

5. A flag which tells if the query is a logical NOT. A value of 1
for this flag SHOULD be interpreted as a logical NOT.

```
enum srelation {
    SRELN_EQUALS  = 0;
    SRELN_GREATER = 1;
    SRELN_LESSER  = 2;
};

enum srchqueryjointype {
    SQUERY_NONE   = 0;
    SQUERY_AND    = 1;
    SQUERY_OR     = 2;
};

struct srchquery {
    srchattrlist      search_attrs;
    srelation         search_relation;
    srchqueryjoinype  sqjtypenext;
    uint32_t          priority;
    uint32_t          flag;
};

typedef struct srchquery srchquerylist<>;
```

The flag in srchquery denotes whether the query is a NOT. A value of
1 for the flag means it's a NOT. For example, if a seach attribute
list i.e. srchattrlist has 2 search attributes (A and B), each with
multiple values, and if the flag has a value of 1, it can be
described as NOT ((srchattrnameA (search_relation i.e. EQ, LT, GT)
srchattrvalues) && (srchattrnameB (search_relation i.e. EQ, LT, GT)
srchattrvalues))

The priority in srchquery determines the precedance. For example, in
the searchquery ((A == B) AND (C == D)) OR (F == G), we want A==B
and C==D computed before F==G. So assign equal priorities to query 1
and query 2, i.e. A==B and C==D and then assign a lower priority to
query 3 i.e. F==G. Similarly to effect the query (A == B) AND ((C
== D)) OR (F == G)), query 2 and query 3 are assigned a higher
priority than query 1. Since each query has a priority number, we
can group queries that need to be executed in the same priority
bucket, the same priority number. If the precedance is expressed in
the form of brackets, then the priority is directly proportional to
the number of brackets enclosing a query.

NOT combined with Greater gives us Lesser than or equal to.
Similarly NOT combined with Lesser than gives us Greater than or
equal to. So combining NOT with the search_relation gives the
flexibility to specify these special relations in a query.

6. New Operations

6.1. Operation 1: PUTOBJROOTFH - Set Object Root Filehandle

     SYNOPSIS

      - -> (cfh)

     ARGUMENT

      void;

     RESULT

```
   struct PUTOBJROOTFHres {
           /* CURRENT_FH: objrootfh */
           nfsstat4        status;
   };
```

     DESCRIPTION

     Replaces the current filehandle with the filehandle that
     represents the root of all the Objects Sets that the server
     contains.

     IMPLEMENTATION

     This is the first operator in a NFS request to set the
     context for the following operations. A READDIR following a
     PUTOBJROOTFH SHOULD list all the Object Sets with respect to
     this filehandle. The READDIR operation SHOULD list only Object
     Sets and not individual pathless objects.

     ERRORS

      NFS4ERR_BADSESSION
      NFS4ERR_DELAY
      NFS4ERR_NOTSUPP
      NFS4ERR_RESOURCE
      NFS4ERR_SERVERFAULT

6.2. Operation 2: PUTSRCHATTR: Search for an Object based on Search
     Attributes

      SYNOPSIS

```
(cfh), srchquerylist, -> (cfh)
```

ARGUMENT

```
struct PUTSRCHATTRargs {
        /* CURRENT_FH: Object Set filehandle */
        srchquerylist    search_query;
};
```

RESULT

```
struct PUTSRCHATTRres {
        /* CURRENT_FH:  Object Set filehandle */
        nfsstat4         status;
};
```

DESCRIPTION

The PUTSRCHATTR operation searches for objects matching
the search attributes. The scope is the Object Set as
specified in the current filehandle (cfh). Instead of
returning the matching filehandles, it just returns a
status and uses READDIR operation's reply to construct
the proper reply. The READDIR reply is used to return
a variable list of filehandles of all the objects that
matches the search query. The READDIR reply contains the
filehandles for the matching objects in the set of
attributes for the object. The object name is an empty
string by default for pathless objects. A PUTSRCHATTR
in a Compound request MUST be followed by a READDIR.
If the PUTSRCHATTR is not followed by a READDIR, then
NFS4ERR_OP_ILLEGAL MUST be returned by the NFS server.

IMPLEMENTATION

The compound operation for implementing the lookup
based on search attributes is like this:
PUTFH (filehandle of object set), PUTSRCHATTR (search
attributes), READDIR (cookie, verifier, dircount,
maxcount, requested_attrs). It is RECOMMENDED that
dircount be set to a value for zero for this sequence
of operations as clients are not supposed to implement
"ls" based on search attribute lookup.

ERRORS

```
NFS4ERR_ACCESS
NFS4ERR_ATTRNOTSUPP
```

```
NFS4ERR_BADCHAR
NFS4ERR_BADNAME
NFS4ERR_BADSESSION
NFS4ERR_BADXDR
NFS4ERR_DELAY
NFS4ERR_INVAL
NFS4ERR_IO
NFS4ERR_NAMETOOLONG
NFS4ERR_NOENT
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTSUPP
NFS4ERR_REP_TOO_BIG
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE
NFS4ERR_WRONG_TYPE
```

7. Modifications to existing NFSv4.1 operations

7.1. CREATE: Modifications

Object Sets MUST be created with the CREATE call. Pathless objects
are RECOMMENDED to be created with the CREATE call, though they can
also be created with the OPEN call. For an Object Set, an unique
objname is MANDATORY. For a pathless object, objname can be an empty
string, namely, "". In case any other objname is supplied with the
CREATE call for a pathless object, it MAY be allowed. The objname
SHOULD become part of the search attributes for the pathless object
or the Object Set.

7.2. OPEN: Modifications

OPEN with a empty objname SHOULD create a pathless object under the
current filehandle. The current filehandle MUST be the filehandle
for an Object Set.

7.3. LOOKUP: Modifications

Since pathless objects can have a name associated with them, LOOKUP
of an objname under the current filehandle of an Object Set can
return a filehandle which maps to the name. However, there can be
multiple objects which map to the same name and in that case, it MAY
not be correct to return the name of any one of them. So if an
objname maps to a single object filehandle, the LOOKUP operation MAY
return that filehandle. Otherwise, it SHOULD return
NFS4ERR_WRONG_TYPE.

7.4. READDIR: Modifications

   READDIR MUST be used immediately following a PUTSRCHATTR to lookup
   pathless objects. If the pathless objects do not have unique names
   READDIR will return empty names. If a READDIR operation is used
   standalone with current filehandle being set to the Object Set
   filehandle, the client MUST request filehandles in the requested set
   of attributes and the server SHOULD return filehandles for all the
   pathless objects in the Object Set. The READDIR following a
   PUTSRCHATTR MUST be used only to return the filehandles and
   attributes for the objects matching the query in PUTSRCHATTR. Any
   other intended use of READDIR following a PUTSRCHATTR SHOULD NOT be
   implemented.

8. Migration and Replication

   The RFC 5661 specifies the attributes fs_locations and
   fs_locations_info that can be used for migration and replication.
   For details, please refer to sections 11.9 and 11.10 of RFC 5661.
   Pathless objects can use the same attributes for migration and
   replication with some minor modifications. The Object Sets which act
   as containers for pathless objects are similar to the root path of a
   filesystem within a server. Hence, for pathless objects, "rootpath"
   and "fs-root" in fs_location4 SHOULD be Object Set names. Similarly
   the "fli_rootpath" and "fli_fs_root" for fs_locations_info4 SHOULD
   contain Object Set names.

9. Acknowledgements

   The authors would like to acknowledge Manjunath Shankararao for
   reviews of the various early versions of the draft. Thomas Haynes
   and Daniel Muntz have provided additional comments.

10. IANA Considerations

   This memo includes no request to IANA.

11. Security Considerations

   All considerations from RFC 3530 Section 16 [RFC3530]

12. References

12.1. Normative References

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

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997,
              <http://xml.resource.org/public/rfc/html/rfc2119.html>.

   [RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
              Beame, C., Eisler, M., and D. Noveck, "Network File System
              (NFS) version 4 Protocol", RFC 3530, April 2003.

   [RFC4506] Eisler, M., "XDR: External Data Representation Standard",
              STD 67, RFC 4506, May 2006.

   [RFC5378] Bradner, S. and J. Contreras, "Rights Contributors Provide
              to the IETF Trust", BCP 78, RFC 5378, November 2008.

   [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol
              Specification Version 2", RFC 5531, May 2009.

   [RFC5661] Shepler, S., Eisler, M., and D. Noveck, "Network File
              System (NFS) Version 4 Minor Version 1 Protocol",
              RFC 5661, January 2010.

12.2. Informative References

   [RFC1094] Nowicki, B., "NFS: Network File System Protocol
              specification", RFC 1094, March 1989.

   [RFC2624] Shepler, S., "NFS Version 4 Design Considerations",
              RFC 2624, June 1999.

   [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax
              Specifications: ABNF", STD 68, RFC 5234, January 2008.

Authors' Addresses

   Dipankar Roy
   NetApp
   495 East Java Drive
   Sunnyvale, CA 94089
   USA

   Phone: +1-408-822-4931
   Email: dipankar@netapp.com

   Mike Eisler
   NetApp
   5765 Chase Point Circle
   Colorado Springs, CO 80919
   USA

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

   Alex RN
   NetApp
   3rd Floor, Fair Winds Block, EGL Software Park,
   Bangalore, Karnataka 560071
   IN

   Phone: +91-80-41843352
   Email: rnalex@netapp.com

                   Storage Control Extensions for NFS Version 4
                        draft-dnoveck-storage-control-01

Abstract

   Developments in storage systems have made it important for
   applications to have control over the characteristics of the storage
   that will be used for their particular files. The development of
   pNFS has added to the usefulness of such control mechanisms as it has
   created the opportunity for the hierarchical organization of file
   names to be separated from the control of storage characteristics for
   individual files, including the assignment to storage locations to
   reflect the performance or other needs of those specific files. This
   document proposes extensions to NFS version 4 to allow storage
   requirements to be communicated to the NFS version 4 server.

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on September 15, 2011.

Copyright Notice

This document is subject to BCP 78 and the IETF Trust's Legal
Provisions Relating to IETF Documents
(http://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF
Contributions published or made publicly available before November
10, 2008. The person(s) controlling the copyright in some of this
material may not have granted the IETF Trust the right to allow
modifications of such material outside the IETF Standards Process.
Without obtaining an adequate license from the person(s) controlling
the copyright in such materials, this document may not be modified
outside the IETF Standards Process, and derivative works of it may
not be created outside the IETF Standards Process, except to format
it for publication as an RFC or to translate it into languages other
than English.

Table of Contents

1. Storage Control Issues

   Storage to which files may be assigned can differ in a number of
   ways, raising the issue of how to control the choice of storage for
   specific files. The range of such choices is not static but can be
   expected to increase as flash memory becomes an option whose use
   needs to be controlled, or various choices of types of local caching
   need to be made. Although all files may well be helped by such
   approaches, the degree to which they will be helped will vary with
   the type of file and the typical application reference pattern for
   it. In addition, the value of improved access will differ with quick
   access to certain files being of much greater value, thereby
   justifying the allocation of more expensive storage resources to such
   files.

   The traditional way that user decisions regarding assignment of
   storage resources have been effected is by assigning specific file
   systems to specific disks or sets of disks. Files placed in that
   file system thereby get the storage characteristics assigned to that
   file system. Where file systems contain storage of various types,
   various heuristics are used to assign files or pieces thereof, to
   storage of various types, generally without any external input about
   application needs.

   The creation of pNFS modifies this pattern in that data and metadata
   are separated. Where pNFS is used, assigning a file to a specific
   file system now controls only where the metadata is located.
   Different files may have their data assigned to different sorts of
   storage, potentially located on different servers. This gives rise
   to the need for a means by which the storage choice for a particular
   file may be made.

   NFS version 4.1 contains a layouthint attribute but this does not
   really address the problem. The focus of the layouthint attribute is
   on the striping configuration, but there is a need to control storage
   characteristics other than this. This is the case even when there is
   only a single stripe (that is, no striping). Even though this is not
   "parallel NFS," using pNFS in this way to provide a separation of
   data and metadata, with the ability to choose locations for data
   based on its characteristics subject to later change in a user-
   transparent manner is very powerful, particularly if the storage
   location is subject to intelligent management.

   Additionally, more sophisticated storage management arrangements make
   it desirable to have a way to specify details for storage handling,
   even when pNFS is not used. When a file system contains different
   sorts of storage, input regarding desired or necessary storage
   characteristics can be used to make storage assignment choices more

in line with application needs.

As a result, the ability to specify desired storage characteristics can provide benfits, both when pNFS is used and when it is not, although pNFS has the most immediate set of needs for means by which to control storage selection.

2. Storage Choice and API Definition

   It needs to be noted that existing API's may not provide means by
   which some of the storage characteristics described herein may be
   communicated to NFSv4 in-kernel clients and from there, to NFSv4
   servers. Nevertheless, definition of a means by which these storage
   characteristics may be communicated to the NFSv4 server is still
   useful for a number of reasons:

      Embedded clients for particular applications may specify this
      information even without any API deinition.

      Client implementations may use various less-than-perfect ways of
      specifying storage characteristics, assigning storage
      chatcteristics based on file ownership or other nominally
      unrealated characteristics that that corelate well with customer
      intentions.

   Note that if the absence of a standard kernel API were sufficient to
   stop this work, it also probably be the case that the absence of a
   means to communicate the information to remote servers might make the
   definition of that API not worth the effort. By defining some
   storage characteristics and a general means of communicating them and
   others (via an extension mechanism) we allow for either:

      The later development of API's to specify these storage
      characteristics.

      The developemt of API's to specify different sets of storage
      characteristics that can then be easily assimilated to this
      mechanism as extensions.

3. Modes of Storage Choice

    There are a number of different ways in which storage choices may be
    indicated:

    o The specific file system location(s) might be specified.

    o Specific types of storage might be specified with selection of
       such choices as SSD, SATA, or fiber channel SAN drives being made
       by the client and effected by the MDS.

    o Desired characteristics of storage including speed (latency and/or
       throughput), amount of storage that will be needed, safety (raid-
       level). Available storage would be selected to meet the required
       characteristics and would be subject to active management as the
       environment changes.

    These different modes of storage choice are all useful in different
    environments. Specification of a specific file system imposes the
    least need for a storage management infrastructure but it requires
    user/application knowledge.

    The other modes imply a sequence of progressively greater
    infrastructure requirements to map specifications to specific storage
    systems and a correspondingly smaller need for user/application
    knowledge of the storage environment. However, such modes of
    operation are very different from existing storage management
    paradigms and the precise ways in which applications and storage
    might communicate are not fully understood.

4. Assuring Extensability

4.1. Requirements for Extensability

   As the examples of different modes of storage choice suggest, there
   are potentially a large number of specific items that might be
   specified in order to effect storage choice. Further, in many cases,
   expected future developments in the area of storage can be expected
   to extend and otherwise modify the characteristics which might be
   specified.

   The need for extensibility is important as one might expect many
   ongoing developments, including those in the areas of storage
   hardware, and file systems, to create corresponding needs to specify
   relevant storage chatacteristics.

   For example, local caching, including writeback caching using flash,
   creates the opportunity for greatly improved performance, at the risk
   of greater complexity in dealing with network failures. This raises
   the issue of allowing the user to make the choice of whether this
   greater performance is worth the risks and difficulties.

   Similarly, the development of distributed file systems raises many
   choices where performance will need to be balanced against various
   forms of safety issues, with specific choices reflecting the specific
   needs of applications dealing with the storage.

   These situations and others that we may not be able to predict,
   require that any attribute scheme in this area allow the
   specification of multiple storage characteristics with the ability to
   easily extend the specification so that it incorporates new
   characteristics to govern storage selection. Further, the need for
   actual use testing before incorporation in an IETF standard, imposes
   new requirements as far as organizing specification of the
   characteristics.

   Having "working code" to effect characteristic selection is not
   sufficient to demonstrate usefulness. The working code may be
   trivial while finding out whether this set of characteristics make
   sense for applications to use or requires extension or modification
   before assuming its final form is not trivial. This may require
   significant trial use among a large set of users running different
   applications, before the details are ready to be standardized.

   These factors increase the need for flexibility, including non-
   private use of characteristics not yet standardized. Accommodating
   this need for flexibility has the potential for unduly interfering
   with interoperability and the design of this feature will need to

avoid that.

4.2. XDR Encoding for Extensability

While each storage property could conceivably be made its own
attribute, the burden that this would place on the IETF process would
be immense. There would be necessary co-ordination (and almost
certain confusion) as individual experimental properties needed
temporary attribute numbers and then had to shift them to other more
permanent numbers. Further, and even more of an issue, storage
property definition would seem to require a minor version, which
seems too heavyweight. This would slow down the process beyond what
should be for something which was its own standard-track RFC.

In order to address these issues, individual properties will be
treated as sub-attributes within a single storage_ctl attribute. To
simplify assignment of sub-attribute numbers, mainly in support of
experimental use, multiple sub-attribute spaces will be supported, to
allow independent development of features each involving multiple
storage properties. Once such a feature is standardized, the
definition of the specific sub-atribute space could simply be made
the subject of a standards-track RFC, with no change to those using
it.

```
    typedef uin32_t   spacenum_sc;    /* Individual property space id. */
    typedef uint32_t bitmap_sc<*>;    /* Bit map for the presence or
                                         absence of individual properties
                                         using bit numbers assigned for
                                         the space. Like bitmap4.     */
    typedef opaque    proplist_sc<*>; /* Data associated with each of the
                                         properties in the bitmap_sc.
                                         Like attrlist4.              */

    struct section_sc {
        spacenum_sc   SpaceSection;    /* Section number.               */
        bitmap_sc     WhichProperties;/* Bit map of properties present. */
        proplist_sc   PropertyData;   /* Data for each of the properties
                                         specified in this section.    */
    };

  typedef section_sc fattr4_storage_ctl<*>;
                                     /* The attribute may have one or
                                        more property sections. */
```

This form of property encoding allows the property set to be extended
without requiring a new minor version. Also, by allowing property

space numbers to be assigned, property sets can be developed
independently, and converted to a standard state without undue
interruption to those using the earlier form.

5. Storage Control

   Storage, along with compute, memory, and network, is an integral part
   of an application's resources. Much like the other types of
   resources consumed by an application, storage needs can be described
   using a set of properties. These properties may serve to describe
   the characteristics of the storage, the intended usage both temporal
   and spatial, quality of service expectations, physical layout over
   available storage media, data access locations, geographical
   distribution, just to name a few. The collection of such properties
   together define the control an application ultimately wants to have
   on storage; conversely, they enable the storage system to more
   effectively and dynamically meet the application's needs as
   specifically expressed, rather than inferred, based on fallible
   heuristics. Henceforth, we will use the term control to refer to the
   property collection.

   It is not difficult to conceive various storage properties. In fact,
   there are numerous of them, due to the diversity of applications and
   the corresponding workload characteristics, the ever increasing
   storage value-adds in the form of data services, and the fast
   changing business requirements. It is an impossible task to capture
   all of them here. Rather, the goal of this document is to define a
   framework in which new properties can be easily added and new
   semantics of the properties can be introduced as necessary without
   disruption. It is desired that they be capable of being used in more
   limited situations, refined as necessary.

5.1. Property Types

   There may be numerous storage properties as mentioned above. We
   need, however, to distinguish at least two types, namely, informative
   properties and enforceable properties. There may very well be other
   systems or criteria when it comes to the classification of storage
   properties; and extensibility shall apply in this case just as it
   does to adding new storage properties. However, there is a need to
   explicitly capture the distinctions between informative and
   enforceable properties in the data model, due to the impact on the
   storage protocol semantics.

5.1.1. Informative Properties

   An informative property, as the name suggests, provides some
   descriptive information about the storage in question. Such
   information is furnished in a single direction from the application
   to the storage system with absolutely no "contractual" implications.
   The storage system may use the information captured in such a
   property for storage optimization. But it is not obligated to do so.

More importantly, the application is not offered any transparency as
to how the storage system may utilize this information. As such, the
information flow is strictly one-way without the prospect for any
feedback. Examples of informative properties are the access pattern
of the storage in use, the expected capacity need, and the estimated
growth rate.

5.1.2. Enforceable Properties

In contrast, an enforceable property may have embedded in it varying
degrees of binding effect. By that, it means the application
specifying the property has expectations that the storage system not
only acts upon but also conveys the action status back in some way.
Unlike the case of an informative property, the information flow in
this case is truly bi-directional, with the backward direction for
monitoring property status, including information on whether a
property has been satisfied or is in the process of being satisfied.
In that sense, an enforceable property has a resemblance to an
agreement, where one might monitor the performance of the other
party.

Applications seeking tighter control of the storage may resort to the
enforceable properties. Examples of enforceable properties could
include the type and speed of sorage but could also include the
availability, reliability, and average throughput and latency.

5.1.2.1. Enforcement Level

To allow varying degrees of control, an enforcement level may be
associated with an enforceable property. There are two levels of
control possible, namely, advisory and mandatory. Regardless of the
level, the storage system should strive to fulfill an enforceable
property. The difference lies in the treatment of an inability to do
so. With an advisory enforcement level, the storage system shall
continue to carry out the operation even if the property could not be
fulfilled; whereas with mandatory, the storage shall fail the
operation without making any modification. In any case, the failure
to fulfill an enforceable property can be communicated to the
application.

5.1.2.2. Compliance Status

While control may suffice to describe the ultimate storage
requirements, i.e., the intended behavior once it has been fully
implemented, it does not by itself capture the dynamic aspects of the
implementation process. This is encompassed by the concept of
"compliance" which indicates the extent to which requested storage
properties have or have not been provided or whether they are still

in the process of being provided. Note that the word "compliance" as
used here has no connection with this word as used to describe issues
conformance with a set of legal requirements for record-keeping,
among other matters.

Control implementation can be a fairly heavyweight process by nature
due to the data intensity involved. This may be true whether it is
during the initial provisioning of storage, or the subsequent change
management, or the remediation of compliance violation. The data
intensive nature of the control implementation process implies that
the transition from non-compliance to compliance will not be
instantaneous in the general case. In other words, the
implementation process remains asynchronous relative to the operation
that triggers it.

The asynchronous nature of the control implementation process may be
captured by the compliance status. The compliance status may have
three different values, namely, Current, Complying, and Failed. The
value Current represents a fully compliant state. The value
Complying refers to a transient state in which the transition to
current is in progress.

The value Failed represents an indefinite state of non-compliance.
In the last case, the storage system may have made the determination
that it is unable to fulfill some or all of the storage properties
given the physical resources available. The application will work
without, but its performance may not be what is desired.

The compliance status describes the state of the control fulfillment
as it pertains to each property. It applies to an enforceable
property only. Its presence is not a syntactic requirement as
defined by the XDR specification. Depending on the operational
context in which the enforceable property is specified, specification
of compliance status may be either invalid, required, or optional
with the specification of more that one such status values possible
in some cases.

5.1.2.3. XDR Encoding for Enforceable Properties

Enforceable properties contain a word which is of type enforce_sc and
allows the enforcement level and compliance status to be specified.
To allow greatest flexibility, all enforcement statuses and
compliance status values are specified as bit values, allowing sets
of enforcement levels and complicance status, to be specified, as
appropriate.

```
      typedef uint32_t enforce_sc;

      const enforce_sc ENFORCE_MANDATORY = 0x1;
      const enforce_sc ENFORCE_ADVISORY = 0x2;
      const enforce_sc ENFORCE_CURENT = 0x10;
      const enforce_sc ENFORCE_COMPLYING = 0x20;
      const enforce_sc ENFORCE_FAILED = 0x40;
```

For most purposes, enforcement words should have a single enforcement
level, either ENFORCE_MANDATORY ENFORCE_ADVISORY. Any enforcement
word containing both bits will result in NFS4ERR_SCTL_BADENF being
returned. Specification of an enforcement word containing neither
will generally result in in NFS4ERR_SCTL_BADENF being returned.
However, it may be specified, when doing a SETATTR that specifies a
reserved empty parameter value to remove a property specification.
Also, it may be specified when doing an VERIFY or NVERIFY to specify
a property without a defined enforcement level.

When specifying a storage property as part of a OPEN, CREATE. or
SETATTR, no enforcement level bits should be specified. If they are,
the error NFS4ERR_SCTL_BADENF is returned. For values returned by
the server in response to GETATTR, enforcement words, containing
exactly one compliance status bit will be returned. When using
storage properties as part of VERIFY or NVERIFY compliance words
containing no compliance bits or any subset of the valid compliance
status bits may be specified.

5.2. Base Property Specifications

The goal for initial inclusion in an NFS version 4 minor version is
to define a small set of property specifications that are generally
useful and do not require a large management infrastructure to
implement. The following are the three property specifications that
fit the description.

```
const spacenum_sc SCNUM_BASE = 1;    /* Base property space id for
                                        all properties in this
                                        group. */

const uint32_t SCBASE_SIZE = 0;      /* Informative property for
                                        size. */
const uint32_t SCBASE_DURATION = 1; /* Informative property for
                                        duration. */
const uint32_t SCBASE_DEVFAIL = 2;  /* Enforceable property for
                                        a device failure limit. */
const uint32_t SCBASE_SYSFAIL = 3;  /* Enforceable property for
                                        a system failure limit. */
const uint32_t SCBASE_FAIL_RPO = 4; /* Enforceable property for
                                        a recovery point objective
                                        in the event of failure. */
const uint32_t SCBASE_SFAIL_RTO = 5;/* Enforceable property for
                                        a recovery time objective
                                        in the event of system
                                        failure. */
const uint32_t SCBASE_DLOSS_RTO = 6;/* Enforceable property for
                                        a recovery time objective
                                        in the event of data loss. */
const uint32_t SCBASE_DISASTER_RTO = 7;/* Enforceable property for a
                                        recovery time objective in
                                        the event of disaster. */
```

5.2.1. Storage Size

   The storage size is an informative property that allows the
   specification of the expected amount of storage to be needed. It may
   be used by the server in seeing if appropriate space is available and
   in reserving space. It is specified as a 64-bit unsigned value
   giving a quantity of storage expressed in bytes.

```
typedef uint64_t propbase_size;
```

   This value may be different from the expected file size. Areas not
   allocated, because of holes for example, are not included. This
   amount of storage may not be required immediately if the file starts
   small and grows. Any derating of specified values is purely a matter
   of server implementation choice and will typically reflect the
   ability to move data to respond to storage overcommitment.

   A value of zero is invalid and would result in the error
   NFS4ERR_SCTL_BADPARM when used in an OPEN or CREATE. When used in
   SETATTR, it causes deletion of a previous storage size specification.

5.2.2. Storage Use Duration

   The storage use duration is an informative property that allows the
   specification of the amount of time that the storage is expected to
   be needed. It may be used in assigning files to storage so that
   space conflicts are reduced. It is specified as a 64-bit unsigned
   value giving a duration in milliseconds.

         typedef uint64_t propbase_duration;

   This allows times from 1 millisecond up to approximately 500 million
   years to be specified. A value of zero is invalid and would result
   in the error NFS4ERR_SCTL_BADPARM when used in an OPEN or CREATE.
   When used in SETATTR, it causes deletion of a previous storage
   duration specification.

5.2.3. Storage Device Failure Limit

   The storage device failure limit is an enforceable property that
   allows the specification of a number of disk drives (or other
   devices) that can fail simultaneously with no data loss and that
   incurs zero recovery time. It must be the case that any set of
   devices of the specified can fail without data loss and with zero
   recovery time.

   Even though there is no recovery time, there may be a significant
   recovery period of modestly reduced performance while adaptation to
   the failure is done and until the completion of which, additional
   device failures will be considered simultaneous.

   The limit is specified as a 32-bit unsigned value giving the minimum
   count of simultaneous failures that can result in data loss to
   clients accessing the file. Storage is assigned which either matches
   this specification or provides a greater value. When pNFS is
   involved the specification applies to storage for the MDS and each
   DS.

         typedef uint32_t prop_dev_fail_lim;

         struct propbase_device_failure_limit {
             enforce_sc   DflEnforce;
             prop_dev_fail_lim DflLimit;
         };

   This allows values from zero to approximately 4 billion to be
   specified. A value of zero is valid and specifies that data loss is
   tolerable in the event of single device failure. (e.g. RAID-0)

5.2.4. Storage System Failure Limit

   The storage system failure limit is an enforceable property that
   allows the specification of the number of storage systems that must
   be able to fail simultaneously without complete data loss. Storage
   is assigned which either matches this specification or provides a
   greater value. When pNFS is involved the specification applies to
   storage for the MDS and DS's as a unit.

```
      typedef uint32_t prop_sys_fail_lim;

      struct propbase_system_failure_limit {
          enforce_sc        SflEnforce;
          prop_sys_fail_lim SflLimit;
      };
```

   This allows values from zero to approximately four billion to be
   specified. A value of zero is valid and specifies data loss in the
   event of a single storage system failure is tolerable.

5.2.5. Storage System Failure RPO

   The recovery point objective (RPO) is the age of files that must be
   recovered from backup storage for normal operations to resume if a
   computer, system, device, or network failure results in data loss.
   The RPO is expressed backward in time (that is, into the past) from
   the instant at which the failure occurs, and can be specified in
   seconds. It is an important consideration in disaster recovery
   planning.

```
      typedef uint64_t prop_sys_fail_RPO;

      struct propbase_system_failure_RPO {
          enforce_sc        SfrpoEnforce;
          prop_sys_fail_RPO SfrpoTime;
      };
```

   This allows values from zero seconds to a value far beyond the age of
   the universe to be specified. A value of zero is valid and indiactes
   that a real-time backup that reflects changes immediately as made is
   required.

5.2.6. Storage System Failure RTO Properties

   Recovery time objective (RTO) properties specify is the maximum
   tolerable length of time that storage assigned may be unavailable in
   the event of various classes of failures. There are three associated
   properties, each of which specifies this value for a particular class

of failure:

> The system failure RTO property, with the property id
> SCBASE_SFAIL_RTO, defines the recovery time objective in the event
> of failures that do not not involve data loss or data corruption.

> The data loss RTO property, with the property id SCBASE_DLOSS_RTO,
> defines the recovery time objective in the event of failures that
> do not not involve the occurrence of a disaster, defined as a
> major environmental event such as a hurricane, earthquake, or
> flood, etc.

> The system failure RTO property, with the property id
> SCBASE_DISASTER_RTO, defines the recovery time objective in the
> event of any falure including disasters.

The actual RTO is a function of the extent to which the interruption
disrupts normal operations and the provisions made to ameliorate this
situation. The desired RTO is a function of the urgency to re-
establish operations and the consequences of failure to promptly do
so. It is an important consideration in recovery planning.

```
typedef uint64_t propbase_sys_fail_RTO;

struct propbase_system_failure_RTO {
    enforce_sc         SfrtoEnforce;
    prop_sys_fail_RTO SfrtoTime;
};
```

RTO values for all of these properties is specified as a 64-bit
integer which specifies a number of microseconds. Although sub-
second RTO values may be difficult, the specification allows small
values which might be useful in the future. The maximum value is
approximately five-hundred thousand years.

6. Uses of the Attribute storage_ctl

   There are four occasions in which the storage_ctl attribute is
   referred to as part of an fattr4 when the storage_ctl mask is
   present.

   o As an attribute specified when creating a file or similar object
      by means of an OPEN or CREATE operation, in order to specify the
      specific storage properies to control the locations on which the
      data is to be put and other associated properties.

   o As an attribute set in a SETATTR operation to change the requested
      location properties. Servers may or may not have the ability to
      change locations on request, but the operation structure will
      indicate whether the server has or doesn't have this ability when
      it is requested.

   o As an attribute read in a GETATTR or READDIR operation to
      determine the currently requested storage properties and the
      degree to which they are current being complied with.

   o As an attribute specified in VERIFY or NVERIFY to test for current
      location property compliance status.

   In addition to the above, a fattr4_storage_ctl of the of the same
   structure as storage_ctl attribute (although not within an fattr)
   also appears within the response data in the following situations.

      For the OPEN, CREATE, and SETATTR operations, when the error
      returned is NFS4ERR_SCTL_FAIL. (See Use of storage_ctl when
      creating a file and Use of storage_ctl in SETATTR for details).

      For the response to the FETCH_SCNOTE operation, when there is a
      pending storage control note to be reported.

   For most purposes, a fattr4_storage_ctl which appears in OPEN,
   CREATE, and SETATTR requests are handled the same and a
   fattr4_storage_ctl which appears in the responses for OPEN, CREATE,
   and SETATTR are handled similarly, while the VERIFY and NVERIFY
   requests form a third similarity group.

6.1. Use of storage_ctl when creating a file

   When the storage_ctl attribute is specified when creating a file, it
   helps decide on the location selected for the file data. If all
   enforceable properties can be immediately satisfied, then the
   operation proceeds normally.

If an enforceable property specified as with the manadatory
enforcement level cannot be satisfied then the operation fails with
the error NFS4ERR_SCTL_FAIL. The response contains, for the case
NFS4ERR_SCTL_FAIL, a fattr4_storage_ctl value which consists all such
enforceable properties which could not be satisfied.

If there is a situation which is not as serious as the failure above,
but still of note, then information relevant to that situation is
stored as a pending storage control note, where it can be fetched (in
the same COMPOUND) by the FETCH_SCNOTE operation.

The following three classes of items are included in situations
leading to a pending storage control note being created.

o An enforceable property of the advisory enforcment level which
   could not be satisfied, i.e its compliance status is indicated as
   failed.

o An enforceable property of the advisory enforcement level which
   could not be immediately satisfied, i.e. its compliance status is
   indicated as Complying.

o An enforceable property of the mandatory enforcement level which
   could not be immediately satisfied, i.e. its compliance status is
   indicated as Complying.

6.2. Use of storage_ctl in SETATTR

A value of the storage_ctl attribute with a structure similar to the
OPEN case is used to change properties for an existing file.
Existing elements properties, not changed by the storage_ctl
attribute remain in effect.

An enforceable property type and the same enforcement level status is
overridden by a corresponding one in the new attributes. To delete
such an enforceable property element without setting a new one, an
enforceable property with no parameter values is used. Similarly, an
informative property will override an existing one of the same type
and use of the that property specification with no parameters is used
to delete an existing informative propety specification without
replacing it.

Failures and notifications are indicated via the error code
NFS4ERR_SCTL_FAILED and creation of pending storage control notes,
just as in the case of OPEN.

6.3. Use of storage_ctl in GETATTR/READDIR

   When the storage_ctl attribute is requested as part of GETATTR or
   READDIR, the fattr4_storage_ctl returned within the file attributes
   reflects the current informative properties together with the
   enforceable properties and together with its current compliance
   status.

   The order of the elements need not reflect that used when the
   attribute was first set. When enforceable properties specify a range
   of multiple possible values, the one returned in the attribute will
   reflect the value actually assigned.

6.4. Use of storage_ctl in VERIFY/NVERIFY

   The storage_ctl attribute presented to VERIFY or NVERIFY is
   interpreted as a series of properties each of which results in a
   truth value. When the truth value for all properties presented is
   true, VERIFY succeeds and NVERIFY fails. Conversely when not all
   properties have that truth value, VERIFY fails and NVERIFY succeeds.

   When informative properties are present they are compared to the
   value set at OPEN, CREATE, or the last SETATTR. If no such value had
   been previously set, the result is treated as non-matching.

   Enforceable properties are classified according to three criteria:

   o Whether they have parameters that indicate specific values
      (With-P) or are the special values defined for that purpose for
      each parameter, which are treated as without parameters (Non-P)
      where the parameter values taken are those specified in the
      corresponding property within the file's attributes.

   o Whether they are, an enforcement level specified (With-Enf) or not
      (Non-Enf).

   o Whether they are together with one or more compliance level levels
      specified (With-Comp) or not (Non-Comp).

   Given the above classifications, the following sets of
   characteristics for enforceable properties in the context of
   storage_ctl for VERIFY, NVERIFY are treated as errors and should
   cause the return of the error NFS4ERR_SCTL_BAD.

   o Non-Comp/Non-Enf/Non-P

   o Non-Comp/Non-Enf/With-P

   o With-Comp/non-Enf/Non-P

   o With-Comp/With-Enf/With-P

   Given the above classifications, the following sets of
   characteristics for enforceable properties in the context of
   storage_ctl for VERIFY, NVERIFY are handled as discussed below.

   Non-Comp/With-Enf/Non-P: is true iff there exists an enforceable
      property containing elements of the associated enforcement status
      as part of the storage_ctl attribute of the file.

   Non-Comp/With-Enf/With-P: is true iff the enforceable proeprty
      specified is compatible with the corresponding enforceable
      property of the associated enforcement level, i.e. if it is
      possible to satisfy both at the same time, without reference to
      whether both or either actually is satisfied.

   With-Comp/Non-Enf/With-P: is true iff the enforceable property
      (including a set of of property specifications of the same type)
      which appear in the storage_ctl attribute passed to the op is
      consistent with the set of compliance levels (often a single level
      but sometimes two) in the specification. That is, the actual
      compliance level must be one of the ones that is specified.

   With-CompB/With-Enf/Non-P: is true iff the enforceable property
      designated by this specification (i.e. that being of the same type
      of specification and the same enforcement level) is consistent
      with the set of compliance levels (often a single level but
      sometimes two) in this specification. That is, the actual
      compliance level must be one of the ones that is specified.

7. The FETCH_SCNOTE Operation

7.1. SYNOPSIS

   (cfh) -> note_pres, note_fattr

7.2. ARGUMENT

   /* CURRENT_FH: */
   void;

7.3. RESULT

   enum SCFres_type {
           SCFres_ABSENT = 0,
           SCFres_PRESENT = 1
   };

   union SCFresok switch (SCFres_type note_pres) {
    case FETCH_PRES:
           fattr4_storage_ctl note_attr;

    case FETCH_ABS:
           void;
   };

   union FETCHres switch (nfsstat4 status) {
    case NFS4_OK:
           /* CURRENT_FH: opened file */
           FETCH4resok   resok4;
    default:
           void;
   };

7.4. DESCRIPTION

   The FETCH_SCNOTE operation is used to fetch a pending storage control
   note for a specified file handle (the current file handle). Note
   that these notes are stored according to the current file handle when
   the operation which gave rise to them was executed. Thus it will be
   the directory on (most) OPENs, and the specific file in the event of
   SETATTR.

   This operation uses the current filehandle value to identify the
   storage control note being sought.

   The operation returns an indication of whether the note is present

and if it is a fattr4_storage_ctl value which consists all
enforceable properties where there is a lack of adequate compliance
to be noted. The use of the the enum scnote_respval rather than a
boolean value allows later extension.

If the note is present, it ceases to be so once the operation is
executed.

7.5. IMPLEMENTATION

Storage control note items are maintained on a per-COMPOUND-request
basis and cease to exist when a COMPOUND fails due to completion or
an the occurrence of an error. This makes it desirable to place the
FETCH_SCNOTE operation close to, generally immediately after the
operation capable of generating the storage control note.

8. Attribute Extension

8.1. Experimental and Other Non-standardized Extensions

   In order to support development of extensions to allow control of new
   file system support attributes, extensions may be defined, each with
   their own proper space id. The goal is to allow quick deployment of
   new features, including those that are vendor-specific at the time
   with the definitions of extensions being publicly available.

   Each such extension set should be registered with IANA. The
   registration will include

   o A short name (a few words) by which the extension will be known.

   o The name or corporate identity of the owner of the extension.

   o Data for the first version of the namespace extension, as
      described below.

   IANA will assign a space id by which the extension will be known.

   Successive versions of spaceid properties should be registered by the
   owner of the extension. The registration should include:

   o The namespace name and number.

   o The namespace version number. The version number is in the form a
      series of small (< 256) integers. The length of the series will
      probably be restricted to something between four and six. The
      version numbers will not be checked for order but only that they
      are unique for a given extension.

   o A document in the form of an internet draft with information on
      the namespace elements paralleling this one. The document will
      contain definitions and property numbers with the space id for all
      of properties within the extension.

      Successive version may add properties but may not delete them,
      clarifications to the semantics of existing properties may be made
      but substantive changes in their semantics should not be made.

      Existing properties may not be defines as invalid or mandatory-to-
      not-implement but they may be defined as incompatible with some
      set of new properties.

   The definitional document should be subject to expert review but the
   purpose of the review is to ensure that the document describes the

extension adequately. It should not be rejected simply because the
expert would do things differently or believe the specified
properties are useful.

8.2. Standardized Extensions

Storage properties may be extended via a standards-track document in
a number of ways. Such an extension may be part of a new minor
version, but may also be done independent of in a standards-track
document other than for a new NFSv4 minor version. When the
extension occurs in a new minor version the document should make
clear whether the additional properties are recommended (as is
normally the case) or mandatory.

The following forms of extension are all valid options:

   Adding additional properties to existing standardized property set
   such as PROP_BASE.

   Creating a new property set its own property set id.

   Converting a previous experimental property set to standards-track
   status based on the publication of the RFC [Need to clarify any
   possible transfer of ownership issues.]

8.3. The storage_ext attribute

The storage_ext attribute is a per-fs attribute which contains
information on the storage_ctl extensions suported by the server when
used on the associated file system. Servers will often report the
same value of the storage_ext attribute for all file systems, but
client should not assume that this is the case.

```
struct section_se {
    spacenum_sc   SpaceSction;    /* Section number. */
    bitmap_sc     WhichProperties;/* Supported properties. */
};

typedef section_se fattr4_storage_ext&lt&#65533;>;
```

The storage_ext attribute consists of section_se arrays, each of
which specify the supported properties for a specific space_id. The
section_se arrays should be reported in ascending numeric order of
spacenum_sc values.

9. Summary

   This chapter serves a reference guide to things discussed above. For
   a more discursive treatment, with less attention due syntax details,
   see above.

9.1. Errors

   This proposal would involve adding the following new errors to the
   NFS version 4 minor version in which it is included.

   NFS4ERR_SCTL_BADPROP Returned when the storage_ctl attribute
      contains properties with a space id unknown to the server, or with
      property bits whose diplacement in the bitmap corresponds to
      property numbers not known to the server as being associated with
      the current space id.

      This error is returnable by OPEN, CREATE, SETATTR, VERIFY, and
      NVERIFY.

   NFS4ERR_SCTL_BADPARM Returned when the storage_ctl attribute
      contains parameters defined as not valid in connection with the
      current property. This includes situations in which multiple
      properties contain values that are defined as inconsistent (as
      opposed to not being satisfiable).

      This error is returnable by OPEN, CREATE, SETATTR, VERIFY, and
      NVERIFY.

   NFS4ERR_SCTL_BADENF Returned when the the storage_ctl attribute
      contains a enforceable property whose enforce_sc is invalid, in
      that it contain multiple enforcement level bits, contains no
      enforcement level bits, in a context in which that is not allowed
      or contains a set of compliance specification bits that is not
      appropriate in the current context.

      This error is returnable by OPEN, CREATE, SETATTR, VERIFY, and
      NVERIFY.

   NFS4ERR_SCTL_BADDATA Returned when the storage_ctl contains a
      section_sc whose PropertyData array does not match the length of
      the properties specified in the associated WhichProperties.

      This error is returnable by OPEN, CREATE, SETATTR, VERIFY, and
      NVERIFY.

NFS4ERR_SCTL_FAIL Returned when a required storage_ctl element
   cannot be satisfied. This is as opposed to the case in which it
   is not being able to be satisfied immediately but is in the
   process of being satisfied.

   This error is returnable by OPEN, CREATE, and SETATTR only.

9.2. Semantic constraints

   This section lists the semantic contraints on property
   specifications. We will have situations in which the attribute will
   fully match specified XDR specification but the specification will
   not be in line with appropriate contextual constraints. This section
   will list those constraints, in order to complement the XDR
   definition above.

   There are four categories of constraints that need to be dealt with:

   o Whether the properties have the associated parameters specified.

   o Whether the properties have an associated enforcement level
      specified.

   o Whether the properties have associated compliance level(s)
      specified.

   o Constraints that involve the validity of combinations of what are
      otherwise allowed situations with regard to the above.

   Each property specifies a particuar value which is invalid and is to
   be treated as inicateing the absence of property parameters (zero
   values, zero-length arays, etc.). Specification of the parameters
   associated with storage properties are generally required and so
   these special value result in NFS4ERR_SCTL_BADPARM being returned.
   The only exceptions are SETATTR, for which a storage property without
   parameters serves to delete the corresponding storage propery in the
   existing attribute, and VERIFY/NVERIFY where it is allowed under some
   circumstances, to be discussed below.

   Specification of the enforcement level is generally required for
   enforceable properties. The only exception is VERIFY/NVERIFY where
   it is allowed under some circumstances, to be discussed below.

   Specification of the compliance status for enforceable properties
   depends on the context in which the properties appears. For OPEN,
   CREATE, and SETATTR, specification of compliance status is not
   allowed. VERIFY/NVERIFY specification of multiple compliance status
   values is allowed, subject to the specific combination constraints

appropriate to VERIFY and NVERIFY as listed below. For all other
contexts, whether in GETATTR, READDIR, the responses in the
NFS4ERR_SCTL_FAIL case, or in the response to the FETCH_SCNOTE
operation, specification of compliance status is required but only a
single compliance status must appear.

In addition to the constraints listed above, in the case of a
storage_ctl attribute within VERIFY/NVERIFY, the properties within
the attribute must meet the additional constraints described in the
section Use of storage_ctl in VERIFY/NVERIFY

When sending responses to GETATTR, READDIR, OPEN, CREATE, and
SETATTR, the server MUST obey these constraints. When receiving
OPEN, SETATTR, VERIFY, and NVERIFY requests that contain the
storage_ctl attribute, the server MUST return the error
NFS4ERR_SCTL_BADENF if the attribute does not follow the specified
constraints and is otherwise valid (matching the XDR property
deinition).

These constraints apply to properties introduced by extensions to the
storage_ctl attirbute unless explicitly overridden in the document
defining the extension. Such a document may add other contextual
constraints that apply to the properties defined by that extension.

10. Possible Future Work

   This document describes a basic framework for storage control and a
   basic set of properties. It is a base for development of this
   feature and could have considerable additions before incorporation in
   NFSv4 an minor version. On the other hand, the feature is intended
   to be defined with sufficient flexibility that many of these
   additions to the feature might be done as subsequent extensions,
   after the basic feature is made part of an NFSv4 minor version.

   The question of which additions are required for an initial version
   of the feature, which are best deferred to later and which proposed
   extensions don't really belong is a complex one and will be a major
   subject of the development of the feature.

   The following list, illustrates some of the possible additions that
   have had some preliminary discussion. It is not intended to be
   exhaustive, and the examination of other additions not yet thought of
   is definitely part of the work to be done:

      Addition of other properties to those in this document, that make
      sense as a basic set of properties, both informative and
      enforceable, for an initial set to be part of an NFSv4 minor
      version.

      Mechanisms to allow a set of properties to be applied to a large
      set of files, including those that are directory-based (with
      inheritance a possible part of the mix), by bulk attribute change
      on a client-specified set of files, or by allowing the client to
      store some set of properties as a persistent object in file
      system, and allowing subsequent storage control attributes to
      reference that persistent object.

      Mechanisms to enable the client to determine possible choices (or
      ranges) for some properties within the context of a given server.
      This would be to simplify and streamline property negotation.

      Mechanisms by which a server could advertise various possible sets
      of property choices to deal with environments where there only
      exists a small set of possible choices each effecting a particular
      choice for many properties, as opposed to a case where multiple
      independent property choices are possible.

11. Acknowledgments

   Mike Eisler reviewed early drafts of this work and made important
   contributions in helping define the direction of the effort.

   David Black reviewed many drafts of this work and made many helpful
   suggestions that improved the quality of the result.

Authors' Addresses

   David Noveck
   EMC
   228 South St.
   Hopkinton, MA 01748
   US

   Phone: +1 508 249 5748
   Email: david.noveck@emc.com

   Pranoop R. Erasani
   NetApp
   48980 Oat Grass Terrace
   Fremont, CA 94539
   US

   Phone: +1 408 822 3282
   Email: pranoop@netapp.com

   Lakshmi N. Bairavasundaram
   NetApp
   475 East Java Drive
   Sunnyvale, CA 94089
   US

   Phone: +1 408 419 5616
   Email: lakshmib@netapp.com

   Peng Dai
   Vmware
   5 Cambridge Center
   Cambridge, MA 02142
   US

   Phone: +1 617 528 7592
   Email: pdai@vmware.com

    Christos Karamonolis
    Vmware
    3401 Hillview Ave.
    Palo Alto, CA 94304
    US

    Phone: +1 650 427 2329
    Email: ckaramonolis@vmware.com

           Extending NFS to Support Enterprise Applications
                draft-eisler-nfsv4-enterprise-apps-01

Abstract

   This document proposes a new operating to efficiently initialize
   files.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

   Enterprise applications (such as databases) have requirements that go
   beyond the traditional use cases for NFS. The requirements falls
   into two broad categories: (1) data integrity and (2) quality of
   service. This document proposes a set of operatons for a future
   minor version of NFSv4 to support requirements of enterprise
   applications.

1.1. Requirements Language

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

2. Operation XX: INITIALIZE - Initialize File

2.1. ARGUMENT

```
   struct INITIALIZE4args {
           /* CURRENT_FH: file */
           stateid4        ia_stateid;
           offset4         ia_offset;
           length4         ia_blocksize
           length4         ia_blockcount;
           length4         ia_reloff_pattern;
           length4         ia_reloff_blocknum;
           opaque          ia_pattern<>;
   };
```


2.2. RESULT

   nfsstat4;

2.3. MOTIVATION

   Most enterprise applications that use files almost always need to
   initialize such files to a known state. Even with existing files,
   after such a file grows, the application needs to initialize the
   expanded region the file. The most trivial initial state is
   intialize every byte to zero. The problem with initializing to zero
   is that it is often difficult to distinguish a byte-range of
   initialized to all zeroes from data corruption, since a pattern of
   zeroes is a probable pattern for corruption. Instead, some
   applications, such as database management systems, use pattern
   consisting of bytes or words of non-zero values. Ideally one would

like to efficiently initialize an entire file to a specified pattern
without having to send WRITE requests for the entire file. The
INITIALIZE operation is bandwidth conserving operation for
initializing file state.

2.4. DESCRIPTION

The INITIALIZE operation is used to initialize an open file to an
iterated pattern. The pattern consists of a fixed string, and a
block number. The pattern is defined by the arguments.

   o ia_offset: where to start the iterated pattern. This value is
      specified in bytes.

   o ia_blocksize: the size of each iteration of the pattern. Each
      iteration is called a block.

   o ia_reloff_pattern: the relative offset within a block where to
      write the specified pattern encoded in ia_pattern.

   o ia_reloff_blocknum: the relative offset within a block where to
      write a 64 bit block number. The block number is incremented once
      a block is written. The block number is always written in little
      endian order. If ia_reloff_blocknum is set to NFS4_UINT64_MAX,
      then this informs the server that no block number is to be
      written.

   o ia_pattern: a fixed string written to every block. If the length
      of ia_pattern is zero, then this informs the server that no string
      is to be written.

The field ia_stateid is the stateid corresponding to the current
filehandle's share reservation, delegation, or byte range lock.

An example will illustrate how the client uses INITIALIZE. Suppose
the arguments (except for ia_stateid) are: { 0, 500, 1000, 8, 0,
"DeadBeef" }. Then starting with offset zero, the content of the
file will have these contents.

```
offset value (decimal or ASCII)
0       0   0   0   0   0   0   0   0
8       'D' 'e' 'a' 'd' 'B' 'e' 'e' 'f'
16-499 zeroes

500     0   0   0   0   0   0   0   1
508     'D' 'e' 'a' 'd' 'B' 'e' 'e' 'f'
516-999 zeroes

...

499500  0   0   0   0   0   0   3   231
499508  'D' 'e' 'a' 'd' 'B' 'e' 'e' 'f'
499516-499999 zeroes
```

2.5.   IMPLEMENTATION

When an NFS server receives this operation, instead of writing the
iterated pattern over each block, it should de-allocate the data of
affected range of the file and record the the values of ia_offset,
ia_blocksize, ia_blockcount, ia_reloff_pattern, ia_reloff_blocknum,
and ia_pattern<> in the file's system metadata. When a client sends
a READ request, instead of returning zeroes, it should construct a
response corresponding to the pattern specified in the arguments to
INITIALIZE.

An application likely has a legacy pattern for initialized blocks
which cannot be mapped to that specified for INITIALIZE. The
application should modified to detect that the block corresponds to
INITIALIZE's pattern. When the application sees such a block, it can
overwrite the block with the legacy pattern. Note that will cause
the block to be allocated on the NFS server.

When the length of ia_pattern is zero and the value of
ia_reloff_blocknum is NFS4_UINT64_MAX, then the client is requesting
that a hole be punched into the file.

3.   Operation XX: IO_ADVISE - Advise server of client's intended I/O
     access pattern

3.1. ARGUMENT

```
        enum io_advise_type {
           IO_ADVISE4_SEQUENTIAL_CACHE       = 0,
           IO_ADVISE4_SEQUENTIAL_DONTCACHE      = 1,
           IO_ADVISE4_RANDOM            = 2,
           IO_ADVISE4_PREFETCH         = 3,
           IO_ADVISE4_PREFETCH_OPPORTUNISTIC = 4,
           IO_ADVISE4_INTENT_TO_WRITE = 5,
           IO_ADVISE4_RECENTLY_USED = 6
        };


        struct io_directions {
                stateid4        iod_stateid;
                offset4         iod_offset;
                bitmap4         iod_flags;
        };

        struct IO_ADVISE4args {
                /* CURRENT_FH: file */
                io_directions ioaa_directions;
                length4         ioaa_count;
        };
```

3.2. RESULT

```
     struct IO_ADVISE4resok {
             bitmap4                 ioar_flags;
     };

     union IO_ADVISE4res switch (nfsstat4 ioar_status) {
      case NFS4_OK:
             IO_ADVISE4resok  ioar_resok4;
      default:
             void;
     };
```

3.3. MOTIVATION

   The client is in a better position to deduce the intended I/O pattern
   than the server, especially if the application provides this
   information. With this information, the server can optimize I/O to
   the file.

3.4. DESCRIPTION

   The IO_ADVISE operation is used advise the server as to how the
   holder of the stateid intends to access the file over the specified
   byte range (iod_offset through iod_offset + ioaa_count - 1).

   o IO_ADVISE4_SEQUENTIAL_CACHE: Sequential access to data expected.
      The server should leave data in its cache.

   o IO_ADVISE4_SEQUENTIAL_DONTCACHE: Sequential access to data
      expected. The server does not need to leave data in its cache.

   o IO_ADVISE4_RANDOM: Random access to data expected.

   o IO_ADVISE4_PREFETCH: Stateid holder expects to access the data
      soon; prefetch data in preparation.

   o IO_ADVISE4_PREFETCH_OPPORTUNISTIC: Stateid holder expects to
      access the data soon; prefetch if it can be done at a marginal
      cost.

   o IO_ADVISE4_INTENT_TO_WRITE: Byte range will be written soon so no
      point in caching data.

   o IO_ADVISE4_RECENTLY_USED: The client has recently accessed the
      byte range in its own cache. This informs the server that the
      data in the byte range remains important to the client. When the
      server reaches resource exhaustion, knowing which data is more
      important allows the server to make better choices about which
      data to, for example purge from a cache, or move to secondary
      storage. It also informs the server which delegations are more
      important, since if delegations are working correctly, once
      delegated to a client, a server might never receive another I/O
      request for the file.

   The results indicate which advice the server intends to follow. The
   server MUST NOT return an error if it does not recognize or does not
   support the requested advice. The server MAY return different advice
   than what the client requested. If it does, then this might be due
   to one of several conditions, including, but not limited to: another
   client advising of a different I/O access pattern; a different I/O
   access pattern from another client that that the server has
   heuristically detected; or the server is not able to support the
   requested I/O access pattern, perhaps due to a temporary resource
   limitation (for example, a request for IO_ADVISE4_SEQUENTIAL_CACHE
   might not be supported because the server cannot afford to cache
   data, and/or cannot afford to queue read-a-head requests).

4. Operation XX: READ_WITH_ADVICE - READ with advice

4.1. ARGUMENT

```
        enum io_advise_type {
            IO_ADVISE4_SEQUENTIAL_CACHE      = 0,
            IO_ADVISE4_SEQUENTIAL_DONTCACHE      = 1,
            IO_ADVISE4_RANDOM           = 2,
            IO_ADVISE4_PREFETCH         = 3,
            IO_ADVISE4_PREFETCH_OPPORTUNISTIC = 4,
            IO_ADVISE4_INTENT_TO_WRITE = 5,
            IO_ADVISE4_RECENTLY_USED = 6
        };


        struct io_directions {
                stateid4        iod_stateid;
                offset4         iod_offset;

                bitmap4         iod_flags;
        };

    struct READ_WITH_ADVICE4args {
            /* CURRENT_FH: file */
            io_directions rwaa_directions;
            length4         rwaa_count;
    };
```

4.2. RESULT

```
      const NFS4_TWO_GB            = 0x80000000;

      typedef opaque twoGB_byte_array4[NFS4_INT32_MAX];

      struct fourGB_buffer4 {
           twoGB_byte_array4[2];
      }

      struct large_buffer4 {
              fourGB_buffer4 lb_big_buffers<>;
              opaque         lb_small_buffer<>;
      }


      struct READ_WITH_ADVICE4resok {
              bool                    rwar_eof;
              bitmap4                 rwar_flags;
              large_buffer4           rwar_data;

      };

      union READ_WITH_ADVICE4res switch (nfsstat4 rwar_status) {
       case NFS4_OK:
              READ_WITH_ADVICE4resok  rwar_resok4;
       default:
              void;
      };
```

4.3. MOTIVATION

   Under some circumstances, the IO_ADVISE operation is insufficient
   when the client is also performing a READ operation. Some advice
   needs to be communicated atomically with the READ operation and an
   IO_ADVISE in the same COMPOUND operation as the READ operation would
   fail to provide the necessary advice. For example, if IO_ADVISE
   proceeded READ, and the server was given advice to not cache the data
   requested by READ, the IO_ADVISE would be too late, because the
   server might already have cached the data. If IO_ADVISE preceded
   READ, in order to be effective, the advice would have to be
   communicated across two operations in the same COMPOUND. This would
   complicate the server implementation.

4.4. DESCRIPTION

   The READ_WITH_ADVICE operation is used read from a file and to advise
   the server as to how the reader of intends to access the file over
   the specified byte range (iod_offset through iod_offset + rwaa_count
   - 1).

   o IO_ADVISE4_SEQUENTIAL_CACHE: Sequential access to data expected.
      The server should leave data in its cache.

   o IO_ADVISE4_SEQUENTIAL_DONTCACHE: Sequential access to data
      expected. The server does not need to leave data in its cache.

   o IO_ADVISE4_RANDOM: Random access to data expected.

   o IO_ADVISE4_PREFETCH: Not applicable.

   o IO_ADVISE4_PREFETCH_OPPORTUNISTIC: Not applicable.

   o IO_ADVISE4_INTENT_TO_WRITE: Byte range will be written soon so no
      point in caching data.

   o IO_ADVISE4_RECENTLY_USED: Explicit hint to keep data of byte range
      in cache.

   The results indicate which advice the server intends to follow. The
   server MUST NOT return an error if it does not recognize or does not
   support the requested advice.

   The intent is that READ_WITH_ADVICE is preferred over READ. In
   addition to providing I/O hints, READ_WITH_ADVICE uses 64 bit data
   lengths, which anticipates the expected improvements in average
   network speeds and network buffer capacities. Because the XDR
   standard does not support 64 bit array lengths, the large_buffer4
   data type is introduced to encode an array of zero or more buffers of
   fixed size of 2^32 bytes, followed by a variable length array of up
   to 2^32 - 1 bytes

5. Operation XX: WRITE_WITH_ADVICE - WRITE with advice

5.1. ARGUMENT

```
          enum stable_how4 { /* from NFSv4.0 */
           UNSTABLE4       = 0,
           DATA_SYNC4      = 1,
           FILE_SYNC4      = 2,
           LAYOUT_SYNC4    = 3 /* new */
          };

       enum io_advise_type {
          IO_ADVISE4_SEQUENTIAL_CACHE       = 0,
          IO_ADVISE4_SEQUENTIAL_DONTCACHE      = 1,
          IO_ADVISE4_RANDOM           = 2,
          IO_ADVISE4_PREFETCH         = 3,
          IO_ADVISE4_PREFETCH_OPPORTUNISTIC = 4,
          IO_ADVISE4_INTENT_TO_WRITE = 5,
          IO_ADVISE4_RECENTLY_USED = 6
       };

       struct io_directions {
               stateid4        iod_stateid;
               offset4         iod_offset;
               bitmap4         iod_flags;
       };

    struct WRITE_WITH_ADVICE4args {
           /* CURRENT_FH: file */
           stable_how4     wwaa_stable;
           io_directions   wwaa_directions;
           large_buffer4   wwaa_data<>;
    };
```

5.2. RESULT

```
struct WRITE_WITH_ADVICE4resok {
        length4             wwar_count;
        stable_how4         wwar_committed;
        bitmap4             wwar_flags;


};

union WRITE_WITH_ADVICE4res switch (nfsstat4 wwar_status) {
 case NFS4_OK:
        WRITE_WITH_ADVICE4resok  wwar_resok4;
 default:
        void;
};
```

5.3. MOTIVATION

   Under some circumstances, the IO_ADVISE operation is insufficient
   when the client is also performing a WRITE operation. Some advice
   needs to be communicated atomically with the WRITE operation and an
   IO_ADVISE in the same COMPOUND operation as the WRITE operation would
   fail to provide the necessary advice. For example, if IO_ADVISE
   proceeded WRITE and the server was given advice to not cache the data
   requested by WRITE the IO_ADVISE would be too late, because the
   server might already have cached the data. If IO_ADVISE preceded
   WRITE in order to be effective, the advice would have to be
   communicated across two operations in the same COMPOUND. This would
   complicate the server implementation.

   This operation adds a new enumerated value for stable_how4 called
   LAYOUT_SYNC4 in order to reduce the need for LAYOUT_COMMIT
   operations.

5.4. DESCRIPTION

   The WRITE_WITH_ADVICE operation is used write to a file and to advise
   the server as to how the writer intends to access the file over the
   specified byte range (iod_offset through iod_offset + amount of data
   in wwaa_data - 1).

   o IO_ADVISE4_SEQUENTIAL_CACHE: Sequential access to data expected.
      The server should leave data in its cache.

o IO_ADVISE4_SEQUENTIAL_DONTCACHE: Sequential access to data
   expected. The server does not need to leave data in its cache.

o IO_ADVISE4_RANDOM: Random access to data expected.

o IO_ADVISE4_PREFETCH: Not applicable.

o IO_ADVISE4_PREFETCH_OPPORTUNISTIC: Not applicable.

o IO_ADVISE4_INTENT_TO_WRITE: Byte range will be over-written soon
   so no point in caching data.

o IO_ADVISE4_RECENTLY_USED: Explicit hint to keep data of byte range
   in cache.

The results indicate which advice the server intends to follow. The
server MUST NOT return an error if it does not recognize or does not
support the requested advice.

The intent is that WRITE_WITH_ADVICE is preferred over WRITE. In
addition to providing I/O hints, WRITE_WITH_ADVICE uses 64 bit data
lengths, which anticipates the expected improvements in average
network speeds and network buffer capacities. Because the XDR
standard does not support 64 bit array lengths, the large_buffer4
data type is introduced to encode an array of zero or more buffers of
fixed size of 2^32 bytes, followed by a variable length array of up
to 2^32 - 1 bytes

If general, if the value of wwaa_stable is valid, then the value of
wwar_committed in the reply MUST NOT be less than the value of
wwaa_stable. The exception is if the wwaa_stable is LAYOUT_SYNC4.
LAYOUT_SYNC4 is an enumerated value that can be used by the client
when the server is an pNFS data server, and the client has a layout
that covers the byte range specified by iod_offset and the amlount of
data in wwaa_data. If the client sends a WRITE_WITH_ADVICE to a data
server with wwaa_stable set to LAYOUT_SYNC4, then a successful reply
MUST return value of wwar_committed equal to LAYOUT_SYNC4 or
FILE_SYNC4. Regardless what value wwaa_stable is, if the server is a
pNFS data server, it MAY return a value of wwar_committed equal to
LAYOUT_SYNC4. Whenever wwar_committed is LAYOUT_SYNC4, this
indicates that range of the layout covered by iod_offset and
wwar_count has been committed to the metadata server, and there is
not need to send a LAYOUT_COMMIT for that range.

6. Operation XX: SET_WORKFLOW_TAG - Sets the workflow tag of a given
    session

6.1. ARGUMENT


```
struct SET_WORKFLOW_TAG 4args {

        uint64_t  swta_tag;
};
```

6.2. RESULT


```
nfsstat4
```


6.3. MOTIVATION

   Enterprise applications require guarantees of quality and/or priority
   of service Providing end-to-end guarantees requires awareness at the
   file services level of the necessary quality and/or priority.

6.4. DESCRIPTION

   Sets the workflow tag of a given session. All operations in progress
   before the server receives SET_WORKFLOW_TAG use the previous tag (if
   any). All operations received after the server receives
   SET_WORKFLOW_TAG use the new tag.

7. Operation XX: SESSION_CTL - Adjust session parameters

7.1. ARGUMENT

```
      struct channel_attrs4 { /* from NFSv4.1 */
              count4                    ca_headerpadsize;
              count4                    ca_maxrequestsize;
              count4                    ca_maxresponsesize;
              count4                    ca_maxresponsesize_cached;
              count4                    ca_maxoperations;
              count4                    ca_maxrequests;
              uint32_t                  ca_rdma_ird&lt1>;
      };

      /* from NFSv4.1 */

   const CREATE_SESSION4_FLAG_PERSIST              = 0x00000001;
   const CREATE_SESSION4_FLAG_CONN_BACK_CHAN       = 0x00000002;
   const CREATE_SESSION4_FLAG_CONN_RDMA            = 0x00000004;

   struct session_ctl4 {
         uint32_t                sc_flags;
         channel_attrs4          sc_fore_chan_attrs;
         channel_attrs4          sc_back_chan_attrs;
   };

      typedef session_ctl SESSION_CTL4args;
```

7.2. RESULT

```
      union SESSION_CTL4res switch (nfsstat4 scr_status) {
      case NFS4_OK:
            session_ctl4  scr_resok4;
      default:
            void;
      };
```

7.3. MOTIVATION

   The introduction of the session model in NFSv4.1 imposes an explicit
   limitation on the number of outstanding requests a client can make of
   an NFS server. In enterprise applications, it is possible each NFS
   request corresponds to a single application request. Thus, the size
   of the slot table can bound the number of outstanding application

requests. While there are workarounds (examples include (1)implement
a mapping layer between application's request slot list and the
client's slot table (2) create additional sessions in order to
preserve a one-to-one mapping between application and client slots),
these workarounds introduce complexity. The application's needs for
more slots are dynamic. The NFSv4.1 model assumes a dynamic slot
table, but the size of the slot table is driven by the server via the
reply to the SEQUENCE operation and the CB_RECALL_SLOT operation.
What is missing is a method for the client to request a larger slot
table.

7.4. DESCRIPTION

This operation allows the client to request changes to the session's
parameters. There are three major fields in the arguments and
results:

o sc_flags. These flags correspond to the csa_flags and csr_flags
   argument and result of CREATE_SESSION. In the result, the value
   of a bit in sc_flags MUST be one of:

   * The corresponding bit in sc_flags of the arguments to
      SESSION_CTL.

   * The corresponding bit in sc_flags of the result of the previous
      SESSION_CTL that the server executed.

   * If the server has not executed a previous SESSION_CTL, then the
      corresponding bit in the csr_flags field of the reply the
      CREATE_SESSION operation that created the session.

o sc_fore_chan_attrs. In the arguments of SESSION_CTL, the fields
   within sc_fore_chan_attrs correspond to the fields of the argument
   csa_fore_chan_attrs in the arguments of CREATE_SESSION. In the
   results of SESSION_CTL, the values fields within
   sc_fore_chan_attrs correspond to the fields of the result
   csr_fore_chan_attrs in the response to CREATE_SESSION. The values
   of the fields in the result sc_fore_chan_attrs are governed
   according to the same rules that govern the values of the fields
   of csr_fore_chan_attrs.

o sc_back_chan_attrs. In the arguments of SESSION_CTL, the fields
   within sc_back_chan_attrs correspond to the fields of the argument
   csa_back_chan_attrs in the arguments of CREATE_SESSION. In the
   results of SESSION_CTL, the values fields within
   sc_back_chan_attrs correspond to the fields of the result
   csr_back_chan_attrs in the response to CREATE_SESSION. The values
   of the fields in the result sc_back_chan_attrs are governed

according to the same rules that govern the values of the fields
of csr_back_chan_attrs.

The SESSION_CTL operation MUST be sent on a COMPOUND operation
prefixed by a SEQUENCE operation with the sa_slotid argument set to
zero. If SESSION_CTL requests a smaller slot table on the fore
channel, and there are operations in progress on other slots of the
fore channel, the server MUST do one of (1) return
NFS4ERR_FORE_CHAN_BUSY (a new error); (2) allow SESSION_CTL to
succeed, wait for the in progress operations to complete and reply to
those operations before replying to SESSION_CTL; or (3) if all the in
progress operations allow the one or both of the errors NFS4ERR_DELAY
or NFS4ERR_SERVERFAULT, allow SESSION_CTL to succeed, abort the in
progress operations, reply with to those operations with either
NFS4ERR_DELAY or NFS4ERR_SERVERFAULT, and then reply to SESSION_CTL.
Because a server is free to return NFS4ERR_FORE_CHAN_BUSY, it is
strongly RECOMMENDED that when a client sends a SESSION_CTL operation
that it have no other requests in progress.

If SESSION_CTL request a smaller slot table on the backchannel and
there are operations in progress on other slots of the backchannel,
the server MUST do one of (1) return NFS4ERR_BACK_CHAN_BUSY; (2)
allow SESSION_CTL to succeed, and for wait replies to the in progress
backchannel operations before replying to SESSION_CTL; or (3) if all
the in progress operations allow the one or both of the errors
NFS4ERR_DELAY or NFS4ERR_SERVERFAULT, allow SESSION_CTL to succeed,
abort the in progress operations, reply with to those operations with
either NFS4ERR_DELAY or NFS4ERR_SERVERFAULT, and then reply to
SESSION_CTL. Before a client sends a SESSION_CTL operation, it
SHOULD reply to all in progress backchannel requests of the same
session as the SESSION_CTL operation.

8. Modification to Operation 42: EXCHANGE_ID - Instantiate Client ID

8.1. ARGUMENT

```
/* new */
const EXCHGID4_FLAG_SUPP_FENCE_OPS     = 0x00000004;
```

8.2. RESULT
Unchanged

8.3. MOTIVATION

   Enterprise applications require guarantees that an operation has
   either aborted or completed. NFSv4.1 provides this guarantee as long
   as the session is alive: simply send a SEQUENCE operation on the same
   slot with a new sequence number, and the successful return of
   SEQUENCE indicates the previous operation has completed. However, if
   the session is lost, there is no way to know when any in progress
   operations have aborted or completed. In hindsight, the NFSv4.1
   specification should have mandated that DESTROY_SESSION abort/
   complete all outstanding operations.

8.4. DESCRIPTION

   A client SHOULD request the EXCHGID4_FLAG_SUPP_FENCE_OPS capability
   when it sends an EXCHANGE_ID operation. The server SHOULD set this
   capability in the EXCHANGE_ID reply whether the client requests it or
   not. If the client ID is created with this capability then the
   following will occur:

   o The server will not reply to DESTROY_SESSION until all operations
      in progress are completed or aborted.

   o The server will not reply to subsequent EXCHANGE_ID invoked on the
      same Client Owner with a new verifier until all operations in
      progress on the Client ID's session are completed or aborted.

   o When DESTROY_CLIENTID is invoked, if there are sessions (both idle
      and non-idle), opens, locks, delegations, layouts, and/or wants
      (Section 18.49) associated with the client ID are removed.
      Pending operations will be completed or aborted before the
      sessions, opens, locks, delegations, layouts, and/or wants are
      deleted.

   o The NFS server SHOULD support client ID trunking, and if it does
      and the EXCHGID4_FLAG_SUPP_FENCE_OPS capability is enabled, then a
      session ID created on one node of the storage cluster MUST be
      destroyable via DESTROY_SESSION. In addition, DESTROY_CLIENTID
      and an EXCHANGE_ID with a new verifier affects all sessions
      regardless what node the sessions were created on.

9. Acknowledgements

   Contributors to this document include: Sumanta Chatterjee, Steve
   Daniel, Mike Eisler, Jeff Kimmel, Akshay Shah, Margaret Susairaj, and
   Lynne Thieme. Reviewers of this document include: Dave Noveck.

10. IANA Considerations

   The IO_ADVISE4 flags are considered extendable. Values 32 through 63
   are reserved for private use. All others are standards track.

11. Security Considerations

   None.

12. References

12.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

12.2. Informative References

   [I-D.eisler-nfsv4-pnfs-dedupe]
              Eisler, M., "Storage De-Duplication Awareness in NFS",
              draft-eisler-nfsv4-pnfs-dedupe-00 (work in progress),
              October 2008.

   [I-D.eisler-nfsv4-pnfs-metastripe]
              Eisler, M., "Metadata Striping for pNFS",
              draft-eisler-nfsv4-pnfs-metastripe-01 (work in progress),
              October 2008.

   [I-D.faibish-nfsv4-pnfs-access-permissions-check]
              Faibish, S., Black, D., Eisler, M., and J. Glasgow, "pNFS
              Access Permissions Check",
              draft-faibish-nfsv4-pnfs-access-permissions-check-03 (work
              in progress), July 2010.

   [I-D.ietf-nfsv4-minorversion1]
              Shepler, S., Eisler, M., and D. Noveck, "NFS Version 4
              Minor Version 1", draft-ietf-nfsv4-minorversion1-29 (work
              in progress), December 2008.

   [I-D.lentini-nfsv4-server-side-copy]
              Lentini, J., Eisler, M., Kenchammana, D., Madan, A., and
              R. Iyer, "NFS Server-side Copy",
              draft-lentini-nfsv4-server-side-copy-05 (work in
              progress), July 2010.

   [I-D.myklebust-nfsv4-pnfs-backend]

            Myklebust, T., "Network File System (NFS) version 4 pNFS
            back end protocol extensions",
            draft-myklebust-nfsv4-pnfs-backend-00 (work in progress),
            July 2009.

   [I-D.quigley-nfsv4-sec-label]
            Quigley, D. and J. Morris, "MAC Security Label Support for
            NFSv4", draft-quigley-nfsv4-sec-label-01 (work in
            progress), February 2010.

   [RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
            Beame, C., Eisler, M., and D. Noveck, "Network File System
            (NFS) version 4 Protocol", RFC 3530, April 2003.

Authors' Addresses

   Michael Eisler (editor)
   NetApp
   5765 Chase Point Circle
   Colorado Springs, CO 80919
   US

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

   Margaret Susairaj (editor)
   Oracle
   7806 Garden Bend
   Sugar Land, TX 77479
   US

   Phone: +1 408 431 7405
   Email: Margaret.Susairaj@oracle.com

                Storage De-Duplication Awareness and Sub-File Caching in NFS
                      draft-eisler-nfsv4-pnfs-dedupe-01.txt

Abstract

   This Internet-Draft describes a means to add awareness of de-
   duplication storage to NFS in order to save resources on NFS client
   and to reduce bandwidth for servicing READ and WRITE operations. The
   means presented leads to a second benefit of providing sub-file,
   block-granular caching.

Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [1].

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 21, 2011.

Copyright Notice

publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction and Motivation

    De-duplication is an emerging trend in the data storage. De-
    duplication means that two files that have common content derive that
    content from a common location on the same storage device. As a
    result, the total storage used is less than the total length of each
    file. De-duplication is also called folding.

    Some file systems have the capability to avoid allocation of storage
    space when the value of each byte in a contiguous range is zero.
    Such a range of a file in such a file system is called a "hole", and
    a file with one or more holes is called a "sparse" file. Sparse
    files represent a trivial form of de-duplication since the value of
    every hole of X bytes in length is the common.

    De-duplication is accomplished in several ways including,

    o Hierarchical de-duplication, where one file is derived from
       another, usually by one file starting of as copy of another, but
       zero, or nearly zero bytes of data are actually copied or moved.
       Instead, the two files share common blocks of data storage. An
       example is a snapshot, where a snapshot is made of a file system,
       such that the snapshot and active file system are equal at the
       time snapshot is taken, and share the same data storage, and thus
       are effectively copies that involve zero or near zero movement of
       data. As the source file system changes, the number of shared
       blocks of data storage reduces. A variation of this is a writable
       snapshot (aka clone) which is taken of a file system. In this
       variation as the source and cloned file systems each change, there
       are fewer shared blocks.

    o In-line de-duplication, where a storage access protocol initiator
       (e.g. an NFS client) creates content via write operations, and the
       target of the storage access protocol checks if the content being
       written is duplicated some where else on the target's storage. If
       so, the data is not written, but instead the logical content
       refers to the duplicate.

    o Background de-duplication, where a background task on the storage
       access protocol target scans for duplicate blocks, and frees all
       but one of the duplicates, mapping the pointers to the now free
       blocks to the remaining duplicate.

    The use of de-duplicated storage does not require changes to the NFS
    protocol. However if the NFS client is caching content from an NFS
    server that provides access to de-duplicated files, without changes
    to the protocol, inefficient use of the resources like memory and
    network bandwidth will result. E.g., two files of length 1024 bytes

are exactly the same and are de-duplicated. The client reads, and
caches the first file. A process on the client requests to read the
second file. If the client were aware the second file was a
duplicate of the first, it would not have read the second file, nor
would it have to cache the second file. A classic use case is
hypervisors, which switch between multiple guest operating systems on
a single physical computer. If each of these guest operating systems
were cloned from a single source, or if each guest was installed from
the same operating system installation image, then much of the data
of each guest might be highly de-duplicated. De-duplication
awareness is consistent with the typical reasons for deploying a
hypervisor: reducing costs by reducing utilization of memory,
computer cycles, and network.

Sub-file caching is most useful when two conditions are met:

o Multiple NFS clients need to access the same file.

o At least one client is modifying the same file, provided this
   client updates a relatively small subset of the file.

Under these two conditions many situations can occur where whole file
caching, as enabled by NFSv4 delegations, at best provides no benefit
and at worst presents a drawback. Examples include:

o One client frequently updates range X of a file, and another
   client frequently reads range Y of a file where X and Y do not
   overlap. With whole file delegations, each client enters a cycle
   of obtain a delegation, process a recall, perform a READ or WRITE
   to the server, with delegations providing no benefit, and thus
   resources being unnecessarily consumed on the client and server.

o Two clients randomly read and write different ranges of the same
   file, and for a sufficiently large file, the probability that they
   need the to access overlapping ranges is very small. Again, with
   whole file delegations, the clients are locked in the same cycle
   as above.

This document describes a method by which NFSv4.1 clients can be
aware of de-duplicated storage for optimizing READ requests. As
proposed, optimization of READ requests not require a new minor
version of NFSv4. Instead, it requires several new layout types, and
thus uses the pNFS protocol [2]. The approach presented here for de-
duplication awareness is easily extended to support sub-file caching
at arbitrary granularities and for abitrary sets of byte ranges of a
file.

This document also describes a method by which NFSv4.x clients can

optimize WRITE requests. The method does require a minor version of NFS.

The XDR description is provided in this document in a way that makes it simple for the reader to extract into a ready to compile form. The reader can feed this document into the following shell script to produce the machine readable XDR description of the de-duplication layout:

```
#!/bin/sh
grep "^  *///" | sed 's?^  *///  ??' | sed 's?^.*///??'
```

I.e. if the above script is stored in a file called "extract.sh", and this document is in a file called "spec.txt", then the reader can do:

```
 sh extract.sh < spec.txt > dd.x
```

The effect of the script is to remove leading white space from each line of the specification, plus a sentinel sequence of "///".

2. Terminology

   o Source file, the file that contains the de-duplicated data.

   o Target file, the file the client has opened.

   o Block, the smallest unit of de-duplication or caching that the
      server is willing to support.

   o Slab, a byte range that refers to lists of other byte ranges that
      contain de-duplicated data (either in whole, or part). A slab can
      refer to a lists of smaller slabs, or lists of blocks.

   o Regular file: An object of file type NF4REG or NF4NAMEDATTR.

3. De-Duplication

3.1. Scope of De-Duplication

   This document only de-duplicates the data contents of regular files.
   Everything else is considered metadata, and de-duplication of
   metadata is not considered in this document. [[Comment.1: Some
   metadata, including the contents of directories and symbolic links,
   as well as attributes (e.g. ACLs) are practical to de-duplicate, but
   not at the granularity of fixed sized blocks. A future revision of

this document might address de-duplication of metadata.]]

De-duplication awareness of regular file content in NFS has two aspects:

o Optimizing READ requests. Here the goal is to avoid reading a pattern of data the client might already have cached.

o Optimizing WRITE requests. Here the goal is to avoid writing a pattern of data the server might already have elsewhere, such that the pattern can be de-duplicated.

3.2. READ Optimization via De-Duplication and pNFS

Providing awareness of de-duplication to clients needs to be practical. If the data structures the server provides to the client are not compact, or require expensive processing and/or network bandwidth, then de-duplication awareness is not practical. The approach presented in this document uses leaf bitmaps to indicate whether a byte range of a file has been de-duplicated, and if so from what offset of what file. Since the granularity of de-duplication will vary by implementation, and by file, the NFS server has the option of providing indirect bitmaps that refer to bitmaps of finer grained byte ranges. An indirect bitmap can refer to another indirect bitmap or a leaf bitmap.

As noted in Section 1, de-duplication can be the result of hierarchical, inline, or background processes. This document presents an approach to providing awareness of de-duplication allows servers to optimize for any approach.

NFSv4.1 introduces pNFS, which allows clients to access data from multiple storage devices. This means that the NFS server is distributed across a set of nodes on a network. Such a server might be capable of de-duplication among the server's nodes. The de-duplication awareness feature will allow servers to present awareness of cross-node de-duplication to NFS clients.

3.2.1. The Definition of De-Duplication Layouts

3.2.1.1. Name of De-Duplication Striping Layout Type

There are multiple de-duplication layout types, in order to support multiple levels of indirection plus a leaf level. Since the maximum sized file in pNFS is $2^{64} - 1$ bytes, a total of 63 levels of indirection are provided.

There are two sets of de-duplication layout types.

   o Within the first set, the name of the top-level de-duplication
      layout type is LAYOUT4_DEDUP_TOP. The names of the remaining de-
      duplication layout types are in this set LAYOUT4_DEDUP_LEVEL_<xx>,
      where <xx> is a two digit decimal number that ranges between 02
      and 64. The server MUST NOT return LAYOUT4_DEDUP_LEVEL_<xx> in
      the response to a GETATTR request for the fs_layout_type
      attribute.

   o Within the second set, the name of the top-level de-duplication
      layout type is LAYOUT4_DEDUP_ROC_TOP. The names of the remaining
      de-duplication layout types are in this set
      LAYOUT4_DEDUP_ROC_LEVEL_<xx>, where <xx> is a two digit decimal
      number that ranges between 02 and 64. The server MUST NOT return
      LAYOUT4_DEDUP_LEVEL_<xx> in the response to a GETATTR request for
      the fs_layout_type attribute.

3.2.1.2. Value of De-Duplication Striping Layout Type

   See Section 7.

3.2.1.3. Definition of the da_addr_body Field of the device_addr4 Data
          Type

   /// %#include "nfs4_prot.h"
   ///
   /// %/* Encoded in the da_addr_body field. */
   ///
   /// union dd_layout_addr switch (bool ddla_simple) {
   ///   case TRUE:
   ///   multipath_list4 ddla_simple_addr;
   ///   case FALSE:
   ///   layouttype4   ddla_complex_addr;
   /// };

                              Figure 1

   The device address is only used in leaf layouts, and even then, only
   when cross server-node de-duplication is in effect. There are two
   types of device addresses, a simple network address, with zero or
   more alternate addresses for multipathing, or a complex address which
   is the value of another layout type. The value of
   ddla_complex_addr.ddldp_ltype MUST NOT be LAYOUT4_DEDUP_TOP or any of
   LAYOUT4_DEDUP_LEVEL_<xx>.

3.2.1.4. Definition of the loh_body Field of the layouthint4 Data Type

```
///   enum dd_layout_hint_care4 {
///
///           DD4_CARE_STRIPE_UNIT_SIZE    = 0x040,
///           DD4_CARE_STRIPE_UNIT_ALIGN   = 0x100
///   };
///   %
///   %/* Encoded in the loh_body field of type layouthint4: */
///   %
///   struct dd_layouthint4 {
///           uint32_t        ddlh_care;
///           length4         ddlh_stripe_unit_size;
///           length4         ddlh_stripe_unit_align;
///   };
```

                              Figure 2

   The layout-type specific content for the LAYOUT4_DEDUP_TOP layout
   type is composed of three fields. The first field, ddlh_care, is a
   set of flags indicating which values of the hint the client cares
   about. If DD4_CARE_STRIPE_UNIT_SIZE is set, then the client
   indicates in the second field, preferred unit of granularity for de-
   duplication in bytes. If DD4_CARE_STRIPE_UNIT_ALIGN is set, then the
   client indicates in the third field, the preferred minimum alignment
   de-duplicated units. For example, if the client specifies
   ddlh_stripe_unit_size as 1024, and ddlh_stripe_unit_align as 128,
   then if two files have in common content a string of bytes that is
   1024 bytes long, and the string is at offset zero in the first file,
   and offset 1024 + 128 = 1152 in the second file, then the client
   would like the server to de-duplicate the common 1024 byte string.
   Note that the leaf layouts returned by the server are unable to
   indicate byte ranges that are not whole multiples of the unit size
   the server uses, so if the server accepts a layout hint with
   ddlh_stripe_unit_align less than ddlh_stripe_unit_size, it will
   report units that are equal to ddlh_stripe_unit_align. If the client
   specifies a value in ddlh_stripe_unit_align that is greater than the
   value of ddlh_stripe_unit_size, the server will ignore the
   ddlh_stripe_unit_align hint.

3.2.1.5. Definition of the loc_body Field of the layout_content4 Data
          Type

```
///  %/*
///  %/* How the bits of each element
///  % * of ddll_blockmap are split up
///  % */
///  const DDLL4_BLKMAP_MASK_ACTIVE      = 0x8000000000000000;
///
///  %/* The remain bits follow DDLL4_BITS_* */
///  const DDLL4_BLKMAP_MASK_PARTITIONED = 0x7FFFFFFFFFFFFFFF;
///
///  %/* These constants index into ddll_bmap_partition */
///  const DDLL4_BITS_FOR_DEVID_IDX   = 0;
///  const DDLL4_BITS_FOR_FH_IDX      = 1;
///  const DDLL4_BITS_FOR_BLK_NUM_IDX = 2;
///
///  struct dd_layout_leaf4 {
///    length4   ddll_block_size;
///
///  % /* ddll_blockmap_partition[0-2] MUST add up to 63 */
///
///    opaque    ddll_blockmap_partition[4];
///    verifier4 ddll_fhsuffix;
///    nfs_fh4   ddll_fhlist<>;
///    uint64_t  ddll_change_attr<>;
///    deviceid4 ddll_devlist<>;
///    uint64_t  ddll_blockmap<>;
///  };
///
///  struct dd_layout_indirect4 {
///    length4     ddli_slab_size;
///    layouttype4 ddli_next_level;
///    bitmap4     ddli_bitmap;
///  };
///
///  union dd_layout4_u switch (bool ddl_is_leaf) {
///    case TRUE:
///      dd_layout_leaf4     ddl_leaf;
///    case FALSE:
///      dd_layout_indirect4 ddl_indirect;
///  };
///  struct dd_layout4 {
///    offset4     ddl_firstoff;
///    offset4     ddl_lastoff;
///    dd_layout4_u ddl_u;
///  };
```

Figure 3

The first fields further bound the layout.

o ddl_firstoff, the first offset in the file that the layout has de-
   duplication information for. The relationship between the
   lo_offset field of the layout4 data type that envelops the de-
   duplication layout and ddl_firstoff is that ddl_firstoff MUST be
   greater than or equal to lo_offset. If ddl_firstoff is not equal
   to lo_offset, then this means that the byte range from lo_offset
   through ddl_firstoff - 1 inclusive either has not been de-
   duplicated or the server has decided to not provide the
   information. The value of the field ddl_firstoff MUST be a whole
   multiple of ddli_slab_size or ddll_block_size.

o ddl_lastoff, the last offset in the file that the layout has de-
   duplication information for. Field ddl_lastoff MUST be greater
   than or equal to ddl_firstoff. Field ddl_lastoff MUST be less
   than or equal to lo_offset + lo_length - 1. If the difference
   between ddl_lastoff and lo_offset + lo_length - 1 exceeds zero,
   then this means that byte range from offset ddl_lastoff + 1
   through lo_offset + lo_length - 1 inclusive either has not be been
   de-duplicated or the server has decided to not provide the
   information. The value of the ddl_lastoff + 1 MUST be a whole
   multiple of ddli_slab_size or ddll_block_size, even if this means
   ddl_lastoff goes beyond the end of file.

The remainder of the de-duplication layout is either a leaf layout or
an indirect layout.

An indirect layout consists of,

o ddli_slab_size is the length, in bytes of each slab represented by
   the ddli_bitmap bitmap array.

o ddli_next_level is the layout type the NFS client MUST use when
   using LAYOUTGET to get finer grained de-duplication information
   about the de-duplication of one or more slabs. This field SHOULD
   be one of LAYOUT4_DEDUP_LEVEL_<xx>. The use of ddli_next_level
   provides a hint to the server for what slab or block size to use
   on the next level of de-duplication.

o ddli_bitmap is a bitmap. If bit N is set in ddli_bitmap, then
   this means that slab N has de-duplicated content. Each bit
   respects a byte range (a slab) of size ddli_slab_size, such that
   ddl_firstoff is the start of the first slab (slab zero, relative
   to ddl_firstoff). Slab N represents the byte range ddl_firstoff +
   N * ddli_slab_size to ddl_firstoff + (N + 1) * ddli_slab_size - 1,

inclusive. The field ddli_bitmap is an array of elements each
consisting of a 32 bit unsigned integer. The number of elements
in ddli_bitmap MUST be greater than or equal to ((((ddl_lastoff -
ddl_firstoff) + 1) / ddli_slab_size) / 32) rounded up to the next
whole number.

A leaf layout consists of,

o ddll_block_size is the length, in bytes of each slab represented
   by the ddll_blockmap array.

o ddll_blockmap_partition is an array of bytes, the first three of
   which are inspected by the client. This array indicates how each
   element of ddll_blockmap is partitioned.

o ddll_fhlist is an array of zero or more filehandles. Each element
   of ddll_blockmap can correspond to a filehandle in ddll_fhlist.
   Each filehandle represents a source file that has a de-duplicated
   block that it shares with the target file. If the array is of
   zero length, then the source file for all de-duplicated blocks is
   the target file.

o ddll_fhsuffix MUST be appended to each filehandle in ddll_fhlist
   that the client uses for READ or LAYOUTGET operations. This
   allows the server to detect if the client is using an invalid
   layout.

o ddll_change_attr is an array of zero or more change attributes.
   If the value of the layout type is between LAYOUT4_DEDUP_TOP and
   LAYOUT4_DEDUP_LEVEL_64, inclusive, then the length of
   ddll_change_attr MUST be greater than or equal to 1. If the value
   of the layout type is between LAYOUT4_DEDUP_ROC_TOP and
   LAYOUT4_CACHE_LEVEL_64, inclusive, then the length of
   ddll_change_attr MUST be zero.

o If ddll_change_attr is not zero in length, then each element
   corresponds an element in ddll_fhlist with the same position in
   the array. I.e. ddll_change_attr[i] is the change attribute for
   the source file identified by ddll_fhlist[i]. If the array is of
   zero length, then for each byte range represented by an element of
   ddl_blockmap that has DDLL4_BLKMAP_MASK_ACTIVE set, the server
   promises to recall the layout of the byte range before the data on
   the range mapped from the source file (represented by an element
   of ddl_fhlist) is changed and before data on range of the target
   file changed. If the ddll_fhlist array is of zero length, and the
   ddll_change_attr array has one element, then ddll_change_attr[0]
   is the change attribute for the source file, which also happens to
   be the target file.

o ddll_devlist is an array of zero or more device IDs, for the
   purpose of enabling cross-node de-duplication. Each element of
   ddll_blockmap can correspond to a device ID in ddll_devlist. Each
   device ID represents a device that has a source file with a de-
   duplicated block. The device ID is always for a LAYOUT4_DEDUP_TOP
   device, and can either map to a network address of an MDS, or a
   non-de-duplication layout type. The device ID will map to an MDS
   network address if the source file has not been striped.
   Otherwise, the device ID will be the layout type used for striping
   the file. By providing the layout type, the client does not have
   to send a GETATTR request on the source file for fs_layout_type
   attribute.

o ddll_blockmap is an array of elements, each a 64 bit unsigned
   integer. Each element corresponds to a block of size
   ddll_block_size. E.g., the first element, ddll_blockmap[0]
   corresponds to the byte range, ddl_firstoff through ddl_firstoff +
   ddll_block_size - 1 inclusive.

   * If ddll_blockmap[i] & DDLL4_BLKMAP_MASK_ACTIVE is non-zero,
      then this element corresponds to a block that is de-duplicated.
      Otherwise, the element does not correspond to a de-duplicated
      block, and the rest of the element is undefined.

   * The mask ddll_blockmap[i] & DDLL4_BLKMAP_MASK_PARTITIONED
      represents a bit field that is partitioned according to the
      content of ddll_blockmap_partition.

      The element ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX]
      indicates how many bits at the start of the bit field are for
      indexing into the ddll_devlist array. The number of elements
      in ddll_devlist MUST be less than or equal to
      2^ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX]. If
      ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX] is zero, then
      this means that the blocks of the source file come from the
      same MDS as the target file.

      The element ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX]
      indicates how many bits in the middle of the bit field are for
      indexing into the ddll_fhlist array. The number of elements in
      ddll_fhlist MUST ne less than or equal to
      2^ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX]. If
      ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX] is zero, this
      means that the source file is the same as the target file in
      every element of ddll_blockmap_partition.

      The element ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX]
      indicates how many bits at the end of the bit field correspond

        to an absolute block number into the source file. The absolute
        offset is calculated by computing the product of
        ddll_block_size and the absolute block number. If
        ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX] is zero,
        then this means the absolute block number of the source is the
        same as the absolute block number of the target.

        The dynamic partitioning of the ddll_blockmap element allows
        for several optimizations. If the de-duplication in the range
        identified by the layout is due to hierarchical de-duplication,
        then there is no need for a block number, so
        ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX] will be
        zero. If there is no cross node de-duplication in the range
        then ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX] will be
        zero. If all the de-duplication in the range is confined to
        the target file, i.e. the duplicate blocks were only in the
        target file and no other file, then
        ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX] will be zero.

   An outline for an algorithm for processing a read() system call when
   the potential for de-duplicated data exists follows. This algorithm
   illustrates how the layout is interpreted. In this algorithm, we
   assume that the client always starts with a layout that spans the
   entire file.

```
   /*
    * Returns a vector call "result" of elements
    * containing key / value pairs of ((offset,
    * length), (status, source_mds, source_fh,
    * source_offset)).
    */

   dedupe_read(read_offset, read_length, target_fh,
       layout4 logr_layout[]) {

     if (number of elements in logr_layout == zero) {
       result[(read_offset, read_length)] =
           NO_DEDUP_AVAILABLE;

       return result;
     }

     for i from the end of logr_layout to start {
       if (logr_layout[i].lo_offset > read_offset) {
         continue;
       }
```

```
        /* check for range split across segments */
        if (logr_layout[i].lo_length <
            read_length) {

          read_offset_A = read_offset;
          read_length_A = logr_layout[i].lo_length;
          read_offset_B = logr_layout[i+1].lo_offset;
          read_length_B = read_length -
            read_length_A;

          result[(read_offset_A, read_length_A)] =
            dedupe_read(read_offset_A, read_length_A,
            target_fh, logr_layout);

          result[(read_offset_B, read_length_B)] =
            dedupe_read(read_offset_B, read_length_B,
            target_fh, logr_layout);

          return result;
        }

        /*
         * If requested offset exceeds last offset of this layout
         * segment, then we have no de-dupe opportunity.
         */
        if (read_offset > ddl_lastoff) {
          result[(read_offset, read_length)] =
            NO_DEDUP_AVAILABLE;
          return result;
        }

        last_offset = read_offset + read_length - 1;

        if (last_offset > ddl_lastoff) {
          /* we cannot de-dupe the entire range */

          result[(ddl_lastoff + 1, last_offset -
            ddl_lastoff)] = NO_DEDUP_AVAILABLE;
          last_offset = ddl_lastoff;
        }
        if (read_offset < ddl_firstoff) {
          /* we cannot de-dupe the entire range */

          result[(read_offset, ddl_firstoff -
            read_offset)] = NO_DEDUP_AVAILABLE;
          read_offset = ddl_firstoff;
        }
```

```
    if (ddl_is_leaf == FALSE) {
      /*
       * Indirect layout. See if the slabs that correspond
       * to the affected range are de-duplicated.
       */

      let trunc_read_off = read_offset truncated
        to next lowest multiple of
        ddli_slab_size;

      let round_last_off = (last_offset rounded
        to next highest multiple of
        ddli_slab_size) - 1;

      first_bit = trunc_read_off /
        ddli_slab_size;
      last_bit =
        (round_last_off + 1) / ddli_slab_size;

      for (j = first_bit; j++; j <= last_bit) {
        k = j / 32;
        l = j mod 32;
        bit = l << 1;

        if (j == first_bit) {
          read_offset_A = read_offset;
          read_length_A = trunc_read_off +
            ddli_slab_size - read_offset;

        } else {
          read_offset_A = ddl_firstoff + (j *
            ddli_slab_size);
          read_length_A = ddli_slab_size;
        }

        if ((ddli_bitmap[k] & bit) == 1) {
          next_layout_off = j * ddli_slab_size +
            trunc_read_off;

          next_layout_length = ddli_slab_size;
          next_layout_type = ddli_next_level;

          if (client does not have layout for
              (next_layout_off,
               next_layout_length, and
               ddli_next_level) {

              send a LAYOUTGET request;
```

```
            }
            let logr_layout_A = logr_layout array
                of layout for (next_layout_off,
                next_layout_length,
                next_layout_type);

            result[(read_offset_A, read_length_A)]
              = dedupe_read(read_offset_A,
              read_length_A, target_fh,
              logr_layout_A);

          } else {
            result[(read_offset_A, read_length_A)]
              = NO_DEDUP_AVAILABLE;

          }
        }
      } else {
        /* process a leaf layout */

        /*
         * determine the masks for block number, filehandle index, and
         * device ID index.
         */
        let trunc_read_off = read_offset truncated
          to next lowest multiple of
          ddll_block_size;

        let round_last_off = (last_offset rounded
          to next highest multiple of
          ddll_block_size) - 1;

        bits_for_blknum = ddll_blockmap_partition
          [DDLL4_BITS_FOR_BLK_NUM_IDX];

        mask_for_blknum = 0;
        for (j = 0; j < bits_for_blknum; j++) {
          mask_for_blknum = (mask_for_blknum
            << 1) | 1;
        }

        bits_for_fh = ddll_blockmap_partition
          [DDLL4_BITS_FOR_FH_IDX];

        mask_for_fh = 0;
        for (j = 0; j < bits_for_fh; j++) {
          mask_for_fh = (mask_for_blknum <<
            1) | 1;
```

```
        }

        mask_for_fh = mask_for_fh <<
          bits_for_blknum;

        bits_for_dev = ddll_blockmap_partition
          [DDLL4_BITS_FOR_DEVID_IDX];

        mask_for_dev = 0;
        for (j = 0; j < bits_for_dev; j++) {
          mask_for_dev = (mask_for_dev << 1)
              | 1;
        }
        mask_for_dev = mask_for_dev <<
          (bits_for_blknum + mask_for_fh);

        if ((bits_for_blknum + bits_for_fh +
            bits_for_dev) != 63) {

          result[(read_offset, read_length)] =
            CORRUPT_LAYOUT;

          return result;
        }

        first_block = trunc_read_off /
          ddll_block_size;
        last_block = (round_last_off + 1) /
          ddll_block_size;
        slopoff = read_offset - trunc_read_off;
        sloplen = round_last_off - last_offset;

        read_offset_A = trunc_read_off;

        for (j = first_block; j++, read_offset_A +=
            ddll_block_size; j <= last_block) {

          if (ddll_blockmap[j] &
              DDLL4_BLKMAP_MASK_ACTIVE) {

            blockmap = ddll_blockmap[j] &
              DDLL4_BLKMAP_MASK_PARTITIONED;

            source_length = ddll_block_size;
            source_change = 0;
            source_dev = 0;

            if (mask_for_blknum == 0) {
```

```
            source_offset = ddl_firstoff + j *
              ddll_block_size;
          } else {
            source_offset = (blockmap &
              mask_for_blknum) * ddll_block_size;
          }

          if (j == first_block) {
            source_offset += slopoff;
            read_offset_B = read_offset;
          } else {
            read_offset_B = read_offset_A;
          }

          if (j == last_block) {
            source_length -= sloplen;
          }

          if (mask_for_fh == 0) {
            source_fh = target_fh;

            if (number of elements in
                ddll_change_attr > 0) {
              source_change = ddll_change_attr[0];
            }
          } else {
            fhidx = (blockmap & mask_for_fh) >>
              bits_for_blknum;
            source_fh = ddll_fhlist[fhidx];
            if (number of elements in
                ddll_change_attr > 0) {
              source_change =
                ddll_change_attr[fhidx];
            }
          }
          read_source_fh = source_fh concatenated
            with ddll_fhsuffix;
          source_ltype = 0;
          source_mds = MDS of target_fh;
          if (mask_for_dev != 0) {
            devidx = (blockmap & mask_for_dev) >>
              bits_for_blknum;
            source_dev = ddll_devlist[devidx];

            if (client does not have device
                address for source_dev) {
              send a GETDEVICEINFO
                (LAYOUT4_DEDUP_TOP, source_dev);
```

```
              }

              if (ddla_simple from GETDEVICEINFO is
                 TRUE) {
                let source_mds be an element of
                  ddla_simple_addr;
              } else {
                source_ltype = ddldp_ltype;

                if (client does not have layout for
                    (source_mds, source_fh,
                     source_ltype, source_offset,
                     source_length)) {

                   send a LAYOUTGET request for
                     (read_source_fh, source_ltype,
                      source_dev, source_offset,
                      source_length) to target_fh's
                      MDS;

                   cache LAYOUTGET result;
                }

                if (client still does not have
                    layout for (source_mds, source_fh,
                     source_ltype, source_offset,
                     source_length)) {
                  source_ltype = 0;
                } else {
                  let source_layout = the layout
                    from cache;
                }
              }
            }

            if (source_change == 0 || client has
                delegation on source_fh) {

              if ({source_fh, source_mds,
                  source_offset, source_length} in
                  cache) {

                result[(read_offset_B,
                  source_length)] =

                  (SATISFY_READ_FROM_CACHE,
                  source_mds, source_fh,
                  source_offset;)
```

```
                } else {
                  if (source_ltype == 0) {
                    if (read_source_fh not yet open)
                    {
                      send an OPEN request for
                        read_source_fh;
                    }
                    send a { PUTFH read_source_fh,
                      READ source_offset,
                      source_length } request to
                      source_mds;

                    enter results in cache;

                  } else {
                    read from read_source_fh,
                      source_offset, source_length
                      according to source_layout;

                    enter results in cache;
                  }
                  result[(read_offset_B,
                    source_length)] =
                    (SATISFY_READ_FROM_CACHE,
                    source_mds, source_fh,
                    source_offset);

                }
              } else {
                if ({source_mds, source_fh,
                    source_offset, source_length} in
                    cache) {

                  send a { PUTFH source_fh, GETATTR
                    change } request to source_mds;

                  if (change attribute ==
                      source_change) {

                    result[(read_offset_B,
                      source_length)] =
                      (SATISFY_READ_FROM_CACHE,
                      source_mds, source_fh,
                      source_offset);

                  } else {
                    result[(read_offset_B,
                      source_length)] =
```

```
                  (STALE_DEDUP_LAYOUT,
                  source_mds, source_fh,
                  source_offset);

              }
            }
          }
        }
      }
    return result;
  }

  /* should never get here */
  result[(read_offset, read_length)] =
    CORRUPT_LAYOUT;

  return result;
}
```

                              Figure 4

   There is a trade off between resources (space and time) used for
   providing de-duplication layouts (especially leaf layouts) and
   resources for redundant caching of de-duplicated storage. E.g., if a
   client has to descend through 52 levels of caching to avoid caching a
   single 4096 byte block twice, then it is not cost effective for the
   server to return a layout. On the other hand, if 99% of a file is
   using de-duplicated storage, then having a complete block map for a
   one gigabyte file, or at least the parts of the file the client wants
   to cache, is more effective than redundantly caching nearly one
   gigabyte of storage.

3.2.1.6. Definition of the lou_body Field of the layoutupdate4 Data
          Type

```
   ///  %/*
   ///  % * LAYOUT4_DEDUP_TOP or any of LAYOUT4_DEDUP_LEVEL_<xx>.
   ///  % * Encoded in the lou_body field of type layoutupdate4:
   ///  % *      Nothing. lou_body is a zero length array of octets.
   ///  % */
   ///  %
```

                              Figure 5

   The LAYOUT4_DEDUP_TOP and LAYOUT4_DEDUP_LEVEL_<xx> layout types have
   no content for lou_body filed of the layoutupdate4 data type.

3.2.1.7. Storage Access Protocols

   The LAYOUT4_DEDUP_TOP and LAYOUT4_DEDUP_LEVEL_<xx> layout types use
   NFSv4.1 operations (and potentially, operations of higher minor
   versions of NFSv4, subject to the definition of a minor version of
   NFSv4) to access de-duplicated data. The de-duplication layout types
   do not affect access to storage devices. Thus a client might be able
   to obtain both a de-duplication layout type and a non-de-duplication
   layout type (e.g., LAYOUT4_NFSV4_1_FILES, LAYOUT4_OSD2_OBJECTS, or
   LAYOUT4_BLOCK_VOLUME) on the same regular file.

3.2.1.8. Revocation of Layouts

   Servers MAY revoke de-duplication layouts. A client using a de-
   duplication layout SHOULD check if the change attribute of the source
   file has changed. The use of the ddll_fhsuffix will prevent clients
   using revoked de-duplication layouts from using potentially stale
   information. Attempts to use filehandles with the value of
   ddll_fhsuffix appended, will result in NFS4ERR_STALE.

3.2.1.9. Recovery

   [[Comment.2: it is likely this section will follow that of the files
   layout type specified in the NFSv4.1 specification.]]

3.2.1.9.1. Failure and Restart of Client

   TBD

3.2.1.9.2. Failure and Restart of Server

   TBD

3.2.1.9.3. Failure and Restart of Storage Device

   TBD

3.2.2. Negotiation

   A pNFS client sends a GETATTR request for the fs_layout_type
   attribute to see if the LAYOUT4_DEDUP_TOP layout type is supported.

3.2.3. Operational Recommendation for Deployment

   Deploy the de-duplication layouts when it a significant fraction of
   data storage is de-duplicated.

3.3. WRITE Optimization When De-Duplication Is Present

   There are two goals

   o Avoid a WRITE of a pattern if client knows that server has stored
      that pattern somewhere else besides the combination of target file
      and byte range. the server

   o Even if the client does not know if the pattern is stored
      somewhere, provide a hint to the server that allows it to quickly
      determine if the pattern is present.

   Accomplishing the former merely requires an operation that refers the
   server to a byte of a file it has stored. One way to is to leverage
   the proposed COPY operation [3]. Accomplishing the latter can be
   done by the client providing checksums of byte range it would like to
   avoid writing. However, to do so would require that client and
   server agree on checksum algorithm, which has the practical problem
   that clients and servers with pre-existing de-duplication features
   are likely to not agree on the checksum algorithm. For this reason,
   this version of the document does not pursue the second goal.

   One caveat using COPY to achieve the first goal (avoiding a WRITE
   when the client knows the server has stored the pattern elsewhere) is
   that there is a window between the time the client has cached a byte
   range of the source file and the time the server receives the COPY
   request. The use of a de-duplication layout that guarantees a recall
   before the relevant byte range of the source file is changed. Note
   that this guarantee is only present if ddll_change_attr is of zero
   length. The client requires a way to force the server to return such
   de-duplication layouts. When the client requests the top level de-
   duplication layout with a type equal to LAYOUT4_DEDUP_TOP |
   LAYOUT4_DEDUP_RECALL_ON_CHANGE. The value of
   LAYOUT4_DEDUP_RECALL_ON_CHANGE is mask with one bit set:

   ///   const LAYOUT4_DEDUP_RECALL_ON_CHANGE = 0x40;

                                Figure 6

4. Sub-File Caching

   Sub-file caching is built using the concepts and data structures
   defined in Section 3.2, which introduces a set of layout types that
   allow customers to optimize READ operations when the NFS client and
   server support de-duplication. Sub-file caching provides a subset of
   the functionality defined by the LAYOUT4_DEDUP_ROC_TOP layout type
   (and layout types LAYOUT4_DEDUP_ROC_LEVEL_02 through

LAYOUT4_DEDUP4_ROC_LEVEL_64 inclusive). The primary similarity is
that a sub-file cache leaf layout provides a guarantee that if a
block is mapped in the bitmap, then the server will recall a layout
covering that block before allowing the block to be modified. The
primary difference is that sub-file cache leaf layout does not have
de-duplication references.

4.1. Value of the Sub-File Caching Layout Type

   See Section 7.

4.2. Sub-File Caching Indirect Layouts

   Indirect layouts for sub-file caching have the same format and data
   types as indirect layouts for de-duplication.

4.3. Sub-File Caching Leaf Layouts

   Leaf layouts for sub-file caching have the same format and data types
   as indirect layouts for de-duplication. However, there are the
   following restrictions:

   o The value of ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX]
      MUST be zero.

   o The value of ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX] MUST
      be zero.

   o The value of ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX]
      MUST be 63.

   o The length of ddll_fhlist MUST be zero.

   o The length of ddll_change_attr MUST be zero.

   o The length of ddll_devlist MUST be zero.

   The effect of the length of ddll_change_attr being of zero length is
   that server will recall the layout of a block before allowing that
   block to be modified. Except for the restriction that
   ddll_change_attr is of zero length, the effect of the above
   restrictions is to disable de-duplication when using the sub-file
   caching layout types. If client wants both sub-file caching and de-
   duplication awareness, it can request the LAYOUT4_DEDUP_ROC_TOP
   layout type.

   Note that the client can safely cache a block of file only if block's
   corresponding element in the ddll_blockmap array has the

DDLL4_BLKMAP_MASK_ACTIVE bit set. The rest of the bits of the element of ddll_blockmap MUST be equal to the array index of the element.

5. Acknowledgements

   Thanks to Pranoop Erasani, Arthur Lent, and Dave Noveck for validating the strategy described in this document.

6. Security Considerations

   If an ACCESS operation by the principal on the source file would fail, then the server has take care when processing requests for de-duplication layouts of the target file. If the server is unable to perform access control at the granularity of the a byte-range, then the server MUST NOT allow the principal to read the source file. A related concern is that if the server can provide per-byte-range access, then the server will need to allow an OPEN operation of the source file by the principal. The server will need to reject READ operations for the non-de-duplicated data. The reader should adjust the algorithm in Figure 4 accordingly.

7. IANA Considerations

   This specification requires 196 additions to the Layout Types registry described in Section 22.4 of [2]. Each added entry has five fields. The first entry is:

   1. Name of layout type: LAYOUT4_DEDUP_TOP.

   2. Value of layout type: TBD1. [[Comment.3: Note to IANA. Assign LAYOUT4_DEDUP_TOP a value that is a whole multiple of 64.]]

   3. Standards Track RFC that describes this layout: RFCTBD65, which is the RFC of this document.

   4. How the RFC Introduces the specification: L.

   5. Minor versions of NFSv4 that can use the layout type: 1.

   The second through 64th additions to the Layout Types registry each have the following form, where <xx> is a decimal number between 02 and 64, inclusive:

1. Name of layout type: LAYOUT4_DEDUP_LEVEL_<xx>.

2. Value of layout type: The result of the expression: <xx> - 1 +
    LAYOUT4_DEDUP_TOP.

3. Standards Track RFC that describes this layout: RFCTBD65, which
    is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 65th entry is:

1. Name of layout type: LAYOUT4_DEDUP_ROC_TOP

2. Value of layout type: The value assigned to LAYOUT4_DEDUP_TOP
    logically ORed with LAYOUT4_DEDUP_RECALL_ON_CHANGE.

3. Standards Track RFC that describes this layout: RFCTBD65, which
    is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 66th through 128th additions to the Layout Types registry each
have the following form, where <xx> is a decimal number between 2 and
64, inclusive:

1. Name of layout type: LAYOUT4_DEDUP_ROC_LEVEL_<xx>.

2. Value of layout type: The result of the expression: <xx> - 1 +
    LAYOUT4_DEDUP_ROC_TOP.

3. Standards Track RFC that describes this layout: RFCTBD65, which
    is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 129th entry is:

1. Name of layout type: LAYOUT4_CACHE_TOP

2. Value of layout type: The value assigned to LAYOUT4_DEDUP_TOP + 2
    * LAYOUT4_DEDUP_RECALL_ON_CHANGE.

3. Standards Track RFC that describes this layout: RFCTBD65, which
    is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 130th through 192nd additions to the Layout Types registry each
have the following form, where <xx> is a decimal number between 2 and
64, inclusive:

1. Name of layout type: LAYOUT4_CACHE_LEVEL_<xx>.

2. Value of layout type: The result of the expression: <xx> - 1 +
    LAYOUT4_CACHE_TOP.

3. Standards Track RFC that describes this layout: RFCTBD65, which
    is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

8. Normative References

    [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement
         Levels", RFC 2119, March 1997.

    [2] Shepler, S., Eisler, M., and D. Noveck, "NFS Version 4 Minor
         Version 1", RFC RFC5661, Jan 2010.

    [3] Lentini, J., Eisler, M., and D. Kenchammana, "NFS Version 4
         Minor Version 1", draft-lentini-nfsv4-server-side-copy-05.txt
         (work in progress), Jul 2010.

Author's Address

   Mike Eisler
   NetApp
   5765 Chase Point Circle
   Colorado Springs, CO 80919
   US

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

                    Metadata Striping for pNFS
                draft-eisler-nfsv4-pnfs-metastripe-02.txt

Abstract

   This Internet-Draft describes a means to add metadata striping to
   pNFS.

Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [1].

include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction and Motivation

   The NFSv4.1 specification describes pNFS [2]. In NFSv4.1, pNFS is
   limited to the data contents of regular files. The content of
   regular files is distributed (striped) across multiple storage
   devices. Metadata is not distributed or striped, and indeed, the
   model presented in the NFSv4.1 specification is that of a single
   metadata server. This document describes a means to add metadata
   striping to pNFS, which includes the notion of multiple metadata
   servers. With metadata striping, multiple metadata servers may work
   together to provide a higher parallel performance.

   This document does not require a new minor version of NFSv4.
   Instead, it requires a new layout type.

   The XDR description is provided in this document in a way that makes
   it simple for the reader to extract into a ready to compile form.
   The reader can feed this document into the following shell script to
   produce the machine readable XDR description of the metadata layout:

   #!/bin/sh
   grep "^ *///" | sed 's?^ */// ??' | sed 's?^.*///??'

   I.e. if the above script is stored in a file called "extract.sh", and
   this document is in a file called "spec.txt", then the reader can do:

    sh extract.sh < spec.txt > md.x

   The effect of the script is to remove leading white space from each
   line of the specification, plus a sentinel sequence of "///".

2. Terminology

      o Initial Metadata Server (I-MDS). The I-MDS is the metadata server
         the client obtains a filehandle from prior to acquiring any layout
         on the file.

      o Layout Metadata Server (L-MDS). The L-MDS is the metadata server
         the client obtains a filehandle from after direction from a
         layout.

      o Regular file: An object of file type NF4REG or NF4NAMEDATTR.

3. Scope of Metadata Striping

   This proposal assumes a model where there are two or more servers
   capable of supporting NFSv4.1 operations. At least one server is an
   I-MDS, and the I-MDS should be thought of as a normal NFSv4.1 server,
   with the additional capability of granting metadata layouts on
   demand. The I-MDS might also be capable of granting non-metadata
   layouts, but this is irrelevant to the scope of metadata striping.
   The model also requires at least one additional server, an L-MDS,
   that is capable of supporting NFSv4.1 operations that are directed to
   the server by the I-MDS. It is permissible for an I-MDS to also be
   an L-MDS, and an L-MDS to also be an I-MDS. Indeed, a simple
   submodel is for every NFSv4.1 server in a set to be both an I-MDS and
   L-MDS.

   Metadata striping applies to all NFSv4.1 operations that operate on
   file objects. These operations can be broken down into three
   classes:

   o Filehandle-only. These are operations that take just filehandles
      as arguments, i.e. the current filehandle, or both the current
      filehandle and the saved filehandle, and no component names of
      files. When a client obtains a filehandle of an file object from
      an NFS server, it can obtain a metadata layout that indicates the
      optimal destination in the network to send filehandle-only
      operations for that file object. For example, after obtaining the
      filehandle via OPEN, and the metadata layout via LAYOUTGET, the
      client wants to get a byte range lock on the file. The client
      sends the LOCK request to the network address specified in the
      metadata layout.

   o Name-based. These are operations that take one or two filehandles
      (i.e. the current file handle, or both the current file handle and
      the saved filehandle) and one or two component names of files.
      When a client obtains a filehandle of a file object that is of
      type directory, it can obtain a metadata layout that indicates the
      optimal destinations in the network to send name-based operations
      for that directory. The optimal destinations MUST apply to the
      current filehandle that the operation uses. In other words, for
      LINK and RENAME, which take both the saved filehandle and the
      current filehandle as parameters, the pNFS client would use the
      metadata layout of the target directory (indicated in the current
      filehandle) for guidance where to send the operation. Note that
      if an L-MDS accepts a LINK or RENAME operation, the L-MDS MUST
      perform the operation atomically. If it cannot, then the L-MDS
      MUST return the error NFS4ERR_XDEV, and the client MUST send the
      operation to the I-MDS.

The choice of destination is a function of the name the client is
requesting. For example, after the client obtains the filehandle
of a directory via LOOKUP and the metadata layout via LAYOUTGET,
the client wants to open a regular file within the directory. As
with the LAYOUT4_NFSV4_1_FILES layout type, the client has a list
network addresses to which to send requests. With the
LAYOUT4_NFSV4_1_FILES layout, the choice of the index in the list
of network addresses was computed from the offset of the read or
write request. With the metadata layout, the choice of the index
is derived from the name (or some other method, such as the name
and one or more attributes of the directory, such as the
filehandle, fileid, etc.) passed to OPEN.

o Directory-reading. These are operations that take one filehandle
   and return the contents of a directory (currently, NFSv4 has just
   one such operation, READDIR). When a client obtains a filehandle
   of a file object that is of type directory, it can obtain a
   metadata layout that indicates the optimal destination in the
   network to send directory reading operations for that directory.
   For example, after the client obtains the filehandle of a
   directory via LOOKUP and the metadata layout via LAYOUTGET, the
   client wants to read the directory. As with the
   LAYOUT4_NFSV4_1_FILES layout type, the client has a list network
   addresses to which to send requests. With the
   LAYOUT4_NFSV4_1_FILES layout, the choice of the index in list of
   network addresses was computed from the offset of the read or
   write request. Since directories have cookies which resemble
   offsets, the choice of the index is computed from the the "cookie"
   argument to the operation.

4. The Definition of Metadata Striping Layout

4.1. Name of Metadata Striping Layout Type

   The name of the metadata striping layout type is LAYOUT4_METADATA.

4.2. Value of Metadata Striping Layout Type

   The value of the metadata striping layout type is TBD1.

4.3. Definition of the da_addr_body Field of the device_addr4 Data Type

```
///  %#include "nfs4_prot.h"
///  union md_layout_addr4 switch (bool mdla_simple) {
///    case TRUE:
///      multipath_list4              mdla_simple_addr;
///    case FALSE:
///      nfsv4_1_file_layout_ds_addr4 mdla_complex_addr;
///  };
```

                              Figure 1

   If mdla_simple is TRUE, the remainder of the device address contains
   a list of elements (mdla_simple_addr), where each element represents
   a network address of an L-MDS which can serve equally as the target
   of metadata operations (typically the filehandle-only operations).
   See Section 13.5 of [2] for a description of how the multipath_list4
   data type supports multi-pathing.

   If mdla_simple is FALSE, the remainder of the device address is the
   same as the LAYOUT4_NFSV4_1_FILES device address, consisting of an
   array of lists of L-MDSes servers (nflda_multipath_ds_list), and an
   array of indices (nflda_stripe_indices). Each element of
   nflda_multipath_ds_list contains one or more subelements, and each
   subelement represents a network address of an L-MDS which may serve
   equally as the target of name-based and directory-reading operations
   (see Section 13.5 of [2]). The number of elements in
   nflda_multipath_ds_list array might be different than the stripe
   count. The stripe count is the number of elements in
   nflda_stripe_indices. The value of each element of
   nflda_stripe_indices is an index into nflda_multipath_ds_list, and
   thus the value of each element of nflda_stripe_indices MUST be less
   than the number of elements in nflda_multipath_ds_list.

4.4. Definition of the loh_body Field of the layouthint4 Data Type

```
///   enum md_layout_hint_care4 {
///           MD4_CARE_STRIPE_UNIT_SIZE   = 0x040,
///           MD4_CARE_STRIPE_CNT_NAMEOPS = 0x080,
///           MD4_CARE_STRIPE_CNT_DIRRDOPS = 0x100
///   };
///   %
///   %/* Encoded in the loh_body field of type layouthint4: */
///   %
///   struct md_layouthint4 {
///           uint32_t        mdlh_care;
///           count4          mdlh_stripe_cnt_nameops;
///           count4          mdlh_stripe_cnt_dirrdops;
///           nfs_cookie4     mdlh_stripe_unit_size;
///   };
```

                              Figure 2

   The layout-type specific content for the LAYOUT4_METDATA layout type
   is composed of four fields. The first field, mdlh_care, is a set of
   flags indicating which values of the hint the client cares about. If
   MD4_CARE_STRIPE_CNT_NAMEOPS is set, then the client indicates in the
   second field, mdlh_stripe_cnt_nameops the preferred stripe count for
   name-based operations. If MD4_CARE_STRIPE_CNT_DIRRDOPS is set, then
   the client indicates in the third field, mdlh_stripe_cnt_dirrdops,
   the preferred stripe count for directory-reading operations. If
   MD4_CARE_STRIPE_UNIT_SIZE is set, then the client indicates in the
   fourth field, mdlh_stripe_unit_size, the preferred stripe unit size
   for directory-reading operations.

4.5. Definition of the loc_body Field of the layout_content4 Data Type

```
///   struct md_layout_fhonly {
///     deviceid4   mdlf_devid;
///     nfs_fh4     mdlf_fh<1>;
///   };
///
///   struct md_layout_namebased {
///     deviceid4   mdln_devid;
///     uint32_t    mdln_namebased_alg;
///     uint32_t    mdln_first_index;
///     nfs_fh4     mdln_fh_list<>;
///   };
///
///   union md_layout_dirread_fhlist
///         switch (bool mdldf_use_namebased) {
///     case TRUE:
///       void;
///     case FALSE:
///       nfs_fh4     mdldf_fh_list<>;
///   };
///
///   struct md_layout_dirread {
///     deviceid4                 mdld_devid;
///     nfs_cookie4               mdld_first_cookie;
///     nfs_cookie4               mdld_unit_size;
///     uint32_t                  mdld_first_index;
///     md_layout_dirread_fhlist mdld_fh_list;
///   };
///
///   struct md_layout4 {
///     md_layout_fhonly    mdl_fhops_layout<1>;
///     md_layout_namebased mdl_nameops_layout<1>;
///     md_layout_dirread   mdl_dirrdops_layout_segments<>;
///   };
```


                             Figure 3

   The reply to a successful LAYOUTGET request MUST contain exactly one
   element in logr_layout. The element contains the metadata layout.
   The metadata layout consists of three variable length arrays. At
   least one of the arrays MUST be of non-zero length.

   o mdl_fhops_layout. This is an array of up to one element. If
      there is one element, the element indicates the preferred set
      L-MDSes as the target of filehandle-only operations. The element
      contains two fields, mdlf_devid, the pNFS device ID of the L-MDS

and mdlf_fh, an array of up to one filehandle.

When the client receives a layout that has a mdl_fhops_layout
array with one element, it uses GETDEVICEINFO to map mdlf_devid to
a device address, of data type md_layout_addr4. The value of the
device address field mdla_simple MUST be TRUE. The client can
then select any element in mdla_simple_addr to send a filehandle-
only operation. The field mdlf_devid MUST map to a device address
with mdla_simple set to TRUE. The current filehandle REQUIRED for
use with the filehandle-only operation is either mdlf_fh[0] (if
and only if mdlf_fh has one element) or it is the filehandle the
pNFS client used as the current filehandle to the LAYOUTGET
operation that returned the metadata layout.

o mdl_nameops_layout. This is an array of up to one element. If
   there is one element, the element indicates the preferred set of
   L-MDS servers to as the target of name-based operations. The list
   of L-MDSes is mapped from the mdln_devid device ID. The array
   mdln_fh_list is used to select a filehandle for accessing an
   L-MDS. The number of elements in this array MUST be one of three
   values:

   * Zero. The means that filehandles used for each L-MDS are the
      same as the filehandle used as the current filehandle to
      LAYOUTGET.

   * One. This means that every L-MDS uses filehandle in
      mdln_fh_list[0].

   * The same number of elements as
      mdla_complex_addr.nflda_multipath_ds_list. Thus, when sending
      a name-based operation to any L-MDS in
      mdla_complex_addr.nflda_multipath_ds_list[X], the filehandle in
      mdln_fh_list[X] MUST be used.

   The field mdld_first_index is the index into the first element of
   the of mdla_complex_addr.nflda_stripe_indices array to use. The
   field mdln_namebased_alg identifies the algorithm used to compute
   the actual element in the mdla_complex_addr.nflda_stripe_indices
   array to use.

   When the client receives a layout that has a mdl_nameops_layout
   array with one element, it uses GETDEVICEINFO to map mdln_devid to
   a device address of data type md_layout_addr4. The value of the
   device address field mdla_simple MUST be set to FALSE.

   The client determines the filehandle and the set of L-MDS network
   addresses to send a name-based operation via the following

      algorithm:

    let F be the function designated by
      mdln_namebased_alg;

    let X = (x1, x2, x3, ...) some set of inputs for
      function F, such that x1 SHOULD be the
      component name of the file;

    stripe_unit_number = F(X);
    stripe_count = number of elements in
      mdla_complex_addr.nflda_stripe_indices;

    j = (stripe_unit_number + mdln_first_index) %
      stripe_count;

    idx = nflda_stripe_indices[j];

    fh_count = number of elements in mdln_fh_list;
    lmds_count = number of elements in
    mdla_complex_addr.nflda_multipath_ds_list;

    switch (fh_count) {
    case lmds_count:
      fh = mdln_fh_list[idx];
      break;

    case 1:
      fh = mdln_fh_list[0];
      break;

    case 0:
      fh = current filehandle passed to LAYOUTGET;
      break;

    default:
      throw a fatal exception;
      break;
    }

    address_list =
      mdla_complex_addr.nflda_multipath_ds_list[idx];

                                Figure 4

      The client would then select an L-MDS from address_list, and send
      the name-based operation using the filehandle specified in fh.

If value of stripe_count is one, then in the above, the value of
the stripe_unit_number derived from mdln_namebased_alg and the
value of mdln_first_index will not change the index into
nflda_stripe_indices because that index will always be zero.
Hence when stripe_count is one, the value mdln_namedbased_alg does
not matter. Thus, when mdla_complex_addr.nflda_stripe_indices has
a length of one, the client MUST ignore the value of
mdln_namebased_alg. This means that all name-based operations on
the directory can be be sent any among the set of L-MDSes
indicated in one element of
mdla_complex_addr.nflda_multipath_ds_list. This serves the common
case of where whole directories are distributed across a set
L-MDSes, but the directories themselves are not striped.

o mdl_dirops_layout_segments. This is an array of zero or more
   elements. Each element indicates the preferred set of L-MDSes as
   the preferred destination for directory reading operations and the
   pattern over which directory reading operations iterates over the
   L-MDSes. The set of L-MDSes is mapped from the value of the
   device ID in the field mdld_devid. The field mdld_first_cookie
   indicates the first directory entry cookie that a directory
   reading operation can use for the first unit of the pattern in
   this element. E.g., the value of mdld_first_cookie can be used as
   the value of the "cookie" field in READDIR4args. In the first
   element, mdld_first_cookie MUST be zero. The last cookie that can
   be used on the pattern can be no higher than one less than the
   value of mdld_first_cookie of the next element. If there is no
   next element, then the pattern is valid for all cookies from
   mdld_first_cookie through NFS4_UINT64_MAX inclusive. The field
   mdld_unit_size indicates the maximum number of cookies that can be
   read from each unit of a pattern, and thus indicates the lowest
   value of the "cookie" field in READDIR4args for each unit after
   the first unit. For example, if mdld_unit_size is 100000, and
   mdld_first_cookie is zero, then value of the "cookie" field in the
   READDIR4args of the READDIR operation sent to the second unit MUST
   be greater than or equal to 100000, and less than 200000. The
   field mdld_fh_list is used to select a filehandle for accessing an
   L-MDS. It is a switched union with a boolean discriminator
   mdldf_use_namebased. If mdldf_use_namebased is TRUE, then the
   array mdl_nameops_layout MUST be of length equal to one and the
   filehandle MUST be selected from mdl_nameops_layout.mdln_fh_list.
   Note however, that the device address MUST still be mapped from
   mdld_devid and not mdln_devid.

o If mdldf_use_namebased is FALSE, then mlld_fh_list is present, and
   number of elements in mdld_fh_list MUST be one of three values:

          * Zero. The means that filehandles used for each L-MDS are the
             same as the filehandle used as the current filehandle to
             LAYOUTGET.

          * One. This means that every L-MDS uses the filehandle in
             mdld_fh_list[0].

          * The same number of elements as
             mdld_complex_addr.nflda_multipath_ds_list. Thus, when sending
             a directory-reading operation to any L-MDS in
             mdld_complex_addr.nflda_multipath_ds_list[X], the filehandle in
             mdld_fh_list[X] MUST be used.

       The field mdld_first_index is the index into the first element of
       the mdld_complex_addr.nflda_stripe_indices array to use.

       When the client receives a layout that has a
       mdl_dirops_layout_segments array with more than zero elements, it
       uses GETDEVICEINFO to map the mdln_devid of each element of the
       array to a device address of data type md_layout_addr4. The value
       of the device address field mdla_simple MUST be set to FALSE. The
       client determines the filehandle and the set of L-MDS network
       addresses to send a name-based operation via the following
       algorithm:

   let cookie_arg be the cookie the pNFS client will
     use as the value of the cookie argument to a
     directory reading operation;

   segment_count = number of elements in
     mdl_dirrdops_layout_segments;

   find index k, such that (cookie_arg >=
     mdl_dirrdops_layout_segments[k].mdld_first_cookie)
     && ((k == (segment_count - 1)) || (cookie_arg
     < mdl_dirrdops_layout_segments[k+1]));

   relative_cookie = cookie_arg -
     mdl_dirrdops_layout_segments[k].mdld_first_cookie;

   address = the result of GETDEVICEINFO on
     mdl_dirrdops_layout_segments[k].mdld_devid;

   i = floor(relative_cookie /
     mdl_dirrdops_layout_segments[k].mdld_unit_size);

   stripe_count = number of elements in
     address.mdla_complex_addr.nflda_stripe_indices;

```
   j = (stripe_unit_number + mdld_first_index) % stripe_count;

   idx = nflda_stripe_indices[j];
   lmds_count = number of elements in
       address.mdla_complex_addr.nflda_multipath_ds_list;

   if (mdl_dirrdops_layout_segments[k].
       mdldf_use_namebased == TRUE) {
     fh_count = number of elements in mdl_nameops_layout[0].mdln_fh_list;
       address.mdla_complex_addr.nflda_multipath_ds_list;
   } else {
     fh_count = number of elements in
       mdl_dirrdops_layout_segments[k].mdld_fh_list.
         mdldf_fh_list;
   }

   switch (fh_count) {
     case lmds_count:
       if (mdl_dirrdops_layout_segments[k].
           mdldf_use_namebased == TRUE) {
         fh = mdln_fh_list[idx];
       } else {
         fh = mdl_dirrdops_layout_segments[k].mdld_fh_list.
           mdldf_fh_list[idx];
       }
       break;

     case 1:
       if (mdl_dirrdops_layout_segments[k].
           mdldf_use_namebased == TRUE) {
         fh = mdln_fh_list[0];
       } else {
         fh = mdl_dirrdops_layout_segments[k].mdld_fh_list.
           mdldf_fh_list[0];
       }
       break;

     case 0:
       fh = current filehandle passed to LAYOUTGET;
       break;

     default:
       throw a fatal exception;
       break;
   }

   address_list =  address.mdla_complex_addr.
     nflda_multipath_ds_list[idx];
```

                              Figure 5

     The client would then select an L-MDS from address_list, and send
     the directory-reading operation using the filehandle specified in
     fh. When the client is reading the beginning of the directory,
     cookie_arg is always zero. Subsequent directory-reading
     operations to read the rest of the directory will use the last
     cookie returned by the L-MDS. An MDS returning a metadata layout
     SHOULD return cookies that can be used directly to the I-MDS that
     returned the layout. However this might not always be possible.
     For example, the directory design of the filesystem of the MDS,
     might not return cookies in ascending order, or any order at all
     for that matter. Whereas, striping by definition requires an
     ordering. In such cases, if a directory is restriped while a pNFS
     client is reading its contents from the L-MDSes, it is possible
     that client will be unable to complete reading the directory, and
     as a result an error is returned to process reading the directory.
     To mitigate this, servers that have sent a CB_LAYOUTRECALL on the
     directory SHOULD NOT revoke the layout as long as they detect that
     the client is completing a read of the entire directory. Once a
     client has received a CB_LAYOUTRECALL, it SHOULD NOT send a
     directory-reading operation to an L-MDS with a cookie argument of
     zero. If the server has sent a CB_LAYOUTRECALL, the L-MDS SHOULD
     reject requests to read the directory that have a cookie argument
     zero and return the error NFS4ERR_PNFS_NO_LAYOUT.

4.6. Definition of the lou_body Field of the layoutupdate4 Data Type

     /// %/*
     /// % * LAYOUT4_METADATA.
     /// % * Encoded in the lou_body field of type layoutupdate4:
     /// % *   Nothing. lou_body is a zero length array of octets.
     /// % */
     /// %

                              Figure 6

   The LAYOUT4_METADATA layout type has no content for lou_body filed of
   the layoutupdate4 data type.

4.7. Storage Access Protocols

   The LAYOUT4_METADATA layout type uses NFSv4.1 operations (and
   potentially, operations of higher minor versions of NFSv4, subject to
   the definition of a minor version of NFSv4) to access striped
   metadata. The LAYOUT4_METADATA does not affect access to storage
   devices. Thus a client might be able to obtain both a
   LAYOUT4_METADATA layout, and a non-LAYOUT4_METADATA layout type

(e.g., LAYOUT4_NFSV4_1_FILES, LAYOUT4_OSD2_OBJECTS, or
LAYOUT4_BLOCK_VOLUME) on the same regular file. Of course, for a
non-regular file, a pNFS client will be unable to get layouts of
types LAYOUT4_NFSV4_1_FILES, LAYOUT4_OSD2_OBJECTS, or
LAYOUT4_BLOCK_VOLUME).

4.8. Revocation of Layouts

   Servers MAY revoke layouts of type LAYOUT4_METADATA. A client
   detects if layout has been revoked if the operation is rejected with
   NFS4ERR_PNFS_NO_LAYOUT. In NFSv4.1, the error NFS4ERR_PNFS_NO_LAYOUT
   could be returned only by READ and WRITE. When the server returns a
   layout of type LAYOUT4_METADATA, the set of operations that can
   return NFS4ERR_PNFS_NO_LAYOUT is: ACCESS, CLOSE, COMMIT, CREATE,
   DELEGRETURN, GETATTR, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP,
   NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, READ, READDIR, READLINK,
   REMOVE, RENAME, SECINFO, SETATTR, VERIFY, WRITE, GET_DIR_DELEGATION,
   SECINFO, SECINFO_NO_NAME, and WANT_DELEGATION.

4.9. Stateids

   The pNFS specification for LAYOUT4_NFSV4_1_FILES states data servers
   MUST be aware of the stateids granted by MDS so that the stateids
   passed to READ and WRITE can be properly validated. This requirement
   extends to the LAYOUT4_METADATA layout type: the L-MDS MUST be aware
   of any non-layout stateids granted by the I-MDS, if and only if the
   client is in contact the L-MDS under direction of a metadata layout
   returned by the I-MDS, and the I-MDS has not recalled or revoked that
   layout. In addition, because an L-MDS can accept operations like
   OPEN and LOCK that create or modify stateids, the I-MDS MUST be aware
   of stateids that an L-MDS has returned to a client, if and only if
   the I-MDS granted the client a metadata layout that directed the
   client to the L-MDS.

   In some cases, one L-MDS MUST be aware of a stateid generated by
   another L-MDS. For example a client can obtain a stateid from the
   L-MDS serving as the destination of name-based operations, which
   includes OPEN. However operations that use the stateid will be
   filehandle-only operations, and the L-MDS the OPEN operation is sent
   to might differ from the L-MDS the LOCK operation for the same target
   file is sent to.

4.10. Lease Terms

   Any state the client obtains from an I-MDS or L-MDS is guaranteed to
   last for an interval lasting as long as the maximum of the lease_time
   attribute of the the I-MDS, and any L-MDS the client is directed to
   as the result of a metadata layout. The client has a lease for each

client ID it has with an I-MDS or L-MDS, and each lease MUST be
renewed separately for each client ID.

4.11. Layout Operations Sent to an L-MDS

An L-MDS MAY allow a LAYOUTGET operation. One reason the L-MDS might
allow a LAYOUTGET operation is to allow hierarchical striping. For
example, for name-based operations, the pNFS server might use a radix
tree, (which the field mdln_namebased_alg would indicate). The first
four bytes of the component name would be combined to form a 32 bit
stripe_unit_number. Once the client contacted the L-MDS, it would
repeat the algorithm on the second four bytes of the component, and
so on until the component name was exhausted.

One an L-MDS grants a layout, the client MUST use only the L-MDS that
granted to the layout to send LAYOUTUPDATE, LAYOUTCOMMIT, and
LAYOUTRETURN.

4.12. Filehandles in Metadata Layouts

The filehandles returned in a metadata layout are subject to becoming
stale at any time. The L-MDS SHOULD NOT return NFS4ERR_STALE unless
the I-MDS has recalled or revoked the corresponding layout.

4.13. READ and WRITE Operations

READ and WRITE are filehandle-only operations, and thus the pNFS
client SHOULD attempt to obtain a non-metadata layout for a regular
file. If it cannot, then it MAY use the metadata layout to send READ
and WRITE operations to an L-MDS. An L-MDS MUST accept a READ or
WRITE operation if the layout the I-MDS returned to the client
included a filehandle-only layout.

4.14. Recovery

[[Comment.1: it is likely this section will follow that of the files
layout type specified in the NFSv4.1 specification.]]

4.14.1. Failure and Restart of Client

TBD

4.14.2. Failure and Restart of Server

TBD

4.14.3. Failure and Restart of Storage Device

   TBD

5. Negotiation

   An pNFS client sends a GETATTR operation for attribute
   fs_layout_type. If the reply contains the metadata layout type, then
   metadata striping is supported, subject to further verification by a
   LAYOUTGET operation. If not, the client cannot use metadata
   striping.

6. Operational Recommendation for Deployment

   Deploy the metadata striping layout when it is anticipated that the
   workload will involve a high fraction of non-I/O operations on
   filehandles.

7. Acknowledgements

   Brent Welch had the idea of returning a separate device ID for
   filehandle-only operations in the metadata layout. Pranoop Erasani,
   Dave Noveck, and Richard Jernigan provided valuable feedback.

8. Security Considerations

   The security considerations of Section 13.12 of [2] which are
   specific to data servers apply to lMDSes. In addition, each lMDS
   server and client are, respectively, a complete NFSv4.1 server and
   client, and so the security considerations of [2] apply to any client
   or server using the metadata layout type.

9. IANA Considerations

   This specification requires an addition to the Layout Types registry
   described in Section 22.4 of [2]. The five fields added to the
   registy are:

   1. Name of layout type: LAYOUT4_METADATA

   2. Value of layout type: TBD1.

   3. Standards Track RFC that describes this layout: RFCTBD2, which is
       the RFC of this document.

   4. How the RFC Introduces the specification: L.

   5. Minor versions of NFSv4 that can use the layout type: 1.

   This specification requires the creation of a registry of hash
   algorithms for supporting the field mdln_namebased_alg. Details TBD.

10. Normative References

   [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement
        Levels", RFC 2119, March 1997.

   [2] Shepler, S., Eisler, M., and D. Noveck, "NFS Version 4 Minor
        Version 1", draft-ietf-nfsv4-minorversion1-26 (work in
        progress), Sep 2008.

Author's Address

   Mike Eisler
   NetApp
   5765 Chase Point Circle
   Colorado Springs, CO 80919
   US

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

NFSv4 Working Group   J. Lentini
Internet-Draft   NetApp
Intended status: Standards Track   D. Ellard
Expires: June 15, 2013   Raytheon BBN Technologies
                                                             R. Tewari
                                                           IBM Almaden
                                                         C. Lever, Ed.
                                                    Oracle Corporation
                                                     December 12, 2012

                Administration Protocol for Federated Filesystems
                     draft-ietf-nfsv4-federated-fs-admin-15

Abstract

   This document describes the administration protocol for a federated
   file system that enables file access and namespace traversal across
   collections of independently administered fileservers. The protocol
   specifies a set of interfaces by which fileservers with different
   administrators can form a fileserver federation that provides a
   namespace composed of the filesystems physically hosted on and
   exported by the constituent fileservers.

Requirements Language

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

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on June 15, 2013.

Copyright Notice

Table of Contents

1. Introduction

   A federated filesystem enables file access and namespace traversal in
   a uniform, secure and consistent manner across multiple independent
   fileservers within an enterprise (and possibly across multiple
   enterprises) with reasonably good performance.

   Traditionally, building a namespace that spans multiple fileservers
   has been difficult for two reasons. First, the fileservers that
   export pieces of the namespace are often not in the same
   administrative domain. Second, there is no standard mechanism for
   the fileservers to cooperatively present the namespace. Fileservers
   might provide proprietary management tools and in some cases an
   administrator might be able to use the proprietary tools to build a
   shared namespace out of the exported filesystems. Relying on vendor-
   proprietary tools does not work in larger enterprises or when
   collaborating across enterprises because it is likely that the system
   will contain fileservers running different software, each with their
   own protocols, with no common protocol to manage the namespace or
   exchange namespace information.

   The requirements for federated namespaces are described in [RFC5716].

   The filesystem federation protocol described in [FEDFS-NSDB] allows
   fileservers from different vendors and/or with different
   administrators to cooperatively build a namespace.

   This document describes the protocol used by administrators to
   configure the fileservers and construct the namespace.

1.1. Definitions

   Administrator: A user with the necessary authority to initiate
      administrative tasks on one or more servers.

   Admin Entity: A server or agent that administers a collection of
      fileservers and persistently stores the namespace information.

   File-access Client: Standard off-the-shelf network attached storage
      (NAS) client software that communicates with fileservers using a
      standard file-access protocol.

   Federation: A set of fileserver collections and singleton
      fileservers that use a common set of interfaces and protocols in
      order to provide to file-access clients a federated namespace
      accessible through a filesystem access protocol.

Fileserver: A server that stores physical fileset data, or refers
   file-access clients to other fileservers. A fileserver provides
   access to its shared filesystem data via a file-access protocol.

Fileset: The abstraction of a set of files and the directory tree
   that contains them. A fileset is the fundamental unit of data
   management in the federation.

   Note that all files within a fileset are descendants of one
   directory, and that filesets do not span filesystems.

Filesystem: A self-contained unit of export for a fileserver, and
   the mechanism used to implement filesets. The fileset does not
   need to be rooted at the root of the filesystem, nor at the export
   point for the filesystem.

   A single filesystem MAY implement more than one fileset, if the
   file-access protocol and the fileserver permit this.

File-access Protocol: A network filesystem access protocol such as
   NFSv3 [RFC1813], NFSv4 [3530bis], or CIFS (Common Internet File
   System) [MS-SMB] [MS-SMB2] [MS-CIFS].

FSL (Fileset Location): The location of the implementation of a
   fileset at a particular moment in time. An FSL MUST be something
   that can be translated into a protocol-specific description of a
   resource that a file-access client can access directly, such as an
   fs_locations attribute (for NFSv4), or a share name (for CIFS).

FSN (Fileset Name): A platform-independent and globally unique name
   for a fileset. Two FSLs that implement replicas of the same
   fileset MUST have the same FSN, and if a fileset is migrated from
   one location to another, the FSN of that fileset MUST remain the
   same.

Junction: A filesystem object used to link a directory name in the
   current fileset with an object within another fileset. The
   server-side "link" from a leaf node in one fileset to the root of
   another fileset.

Namespace: A filename/directory tree that a sufficiently authorized
   file-access client can observe.

NSDB (Namespace Database) Service: A service that maps FSNs to FSLs.
   The NSDB may also be used to store other information, such as
   annotations for these mappings and their components.

   NSDB Node: The name or location of a server that implements part of
      the NSDB service and is responsible for keeping track of the FSLs
      (and related info) that implement a given partition of the FSNs.

   Referral: A server response to a file-access client access that
      directs the client to evaluate the current object as a reference
      to an object at a different location (specified by an FSL) in
      another fileset, and possibly hosted on another fileserver. The
      client re-attempts the access to the object at the new location.

   Replica: A replica is a redundant implementation of a fileset. Each
      replica shares the same FSN, but has a different FSL.

      Replicas may be used to increase availability or performance.
      Updates to replicas of the same fileset MUST appear to occur in
      the same order, and therefore each replica is self-consistent at
      any moment.

      We do not assume that updates to each replica occur
      simultaneously. If a replica is offline or unreachable, the other
      replicas may be updated.

   Server Collection: A set of fileservers administered as a unit. A
      server collection may be administered with vendor-specific
      software.

      The namespace provided by a server collection could be part of the
      federated namespace.

   Singleton Server: A server collection containing only one server; a
      stand-alone fileserver.

2. Protocol

   The RPC protocol used by the administration operations is ONC RPC
   [RFC5531]. The data structures used for the parameters and return
   values of these procedures are expressed in this document in XDR
   [RFC4506].

   The XDR definitions below are formatted to allow the reader to easily
   extract them from the document. The reader can use the following
   shell script to extract the definitions:

```
<CODE BEGINS>

#!/bin/sh
grep '^ *///' | sed 's?^ */// ??' | sed 's?^ *///$??'

<CODE ENDS>
```

If the above script is stored in a file called "extract.sh", and this document is in a file called "spec.txt", then the reader can do:

```
<CODE BEGINS>

sh extract.sh < spec.txt > admin1.xdr

<CODE ENDS>
```

The effect of the script is to remove leading white space from each line, plus a sentinel sequence of "///".

The protocol definition in XDR notation is shown below. We begin by defining basic constants and structures used by the protocol. We then present the procedures defined by the protocol.

```
<CODE BEGINS>


  /// /*
  ///  * Copyright (c) 2010-2012 IETF Trust and the persons identified
  ///  * as authors of the code.  All rights reserved.
  ///  *
  ///  * The authors of the code are the authors of
  ///  * [draft-ietf-nfsv4-federated-fs-admin-xx.txt]: J. Lentini,
  ///  * C. Everhart, D. Ellard, R. Tewari, and M. Naik.
  ///  *
  ///  * Redistribution and use in source and binary forms, with
  ///  * or without modification, are permitted provided that the
  ///  * following conditions are met:
  ///  *
  ///  * - Redistributions of source code must retain the above
  ///  *   copyright notice, this list of conditions and the
  ///  *   following disclaimer.
  ///  *
  ///  * - Redistributions in binary form must reproduce the above
  ///  *   copyright notice, this list of conditions and the
  ///  *   following disclaimer in the documentation and/or other
```

```
/// *    materials provided with the distribution.
/// *
/// * - Neither the name of Internet Society, IETF or IETF
/// *    Trust, nor the names of specific contributors, may be
/// *    used to endorse or promote products derived from this
/// *    software without specific prior written permission.
/// *
/// *    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
/// *    AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
/// *    WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// *    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
/// *    FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
/// *    EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
/// *    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
/// *    EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
/// *    NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
/// *    SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// *    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
/// *    LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
/// *    OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
/// *    IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
/// *    ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
/// */
///
/// enum FedFsStatus {
///   FEDFS_OK                             = 0,
///   FEDFS_ERR_ACCESS                     = 1,
///   FEDFS_ERR_BADCHAR                    = 2,
///   FEDFS_ERR_BADNAME                    = 3,
///   FEDFS_ERR_NAMETOOLONG                = 4,
///   FEDFS_ERR_LOOP                       = 5,
///   FEDFS_ERR_BADXDR                     = 6,
///   FEDFS_ERR_EXIST                      = 7,
///   FEDFS_ERR_INVAL                      = 8,
///   FEDFS_ERR_IO                         = 9,
///   FEDFS_ERR_NOSPC                      = 10,
///   FEDFS_ERR_NOTJUNCT                   = 11,
///   FEDFS_ERR_NOTLOCAL                   = 12,
///   FEDFS_ERR_PERM                       = 13,
///   FEDFS_ERR_ROFS                       = 14,
///   FEDFS_ERR_SVRFAULT                   = 15,
///   FEDFS_ERR_NOTSUPP                    = 16,
///   FEDFS_ERR_NSDB_ROUTE                 = 17,
///   FEDFS_ERR_NSDB_DOWN                  = 18,
///   FEDFS_ERR_NSDB_CONN                  = 19,
///   FEDFS_ERR_NSDB_AUTH                  = 20,
///   FEDFS_ERR_NSDB_LDAP                  = 21,
///   FEDFS_ERR_NSDB_LDAP_VAL              = 22,
```

```
///   FEDFS_ERR_NSDB_NONCE                      = 23,
///   FEDFS_ERR_NSDB_NOFSN                      = 24,
///   FEDFS_ERR_NSDB_NOFSL                      = 25,
///   FEDFS_ERR_NSDB_RESPONSE                   = 26,
///   FEDFS_ERR_NSDB_FAULT                      = 27,
///   FEDFS_ERR_NSDB_PARAMS                     = 28,
///   FEDFS_ERR_NSDB_LDAP_REFERRAL              = 29,
///   FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL          = 30,
///   FEDFS_ERR_NSDB_LDAP_REFERRAL_NOTFOLLOWED  = 31,
///   FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL       = 32,
///   FEDFS_ERR_PATH_TYPE_UNSUPP                = 33,
///   FEDFS_ERR_DELAY                           = 34,
///   FEDFS_ERR_NO_CACHE                        = 35,
///   FEDFS_ERR_UNKNOWN_CACHE                   = 36,
///   FEDFS_ERR_NO_CACHE_UPDATE                 = 37
/// };
///
/// typedef opaque               utf8string<>;
/// typedef utf8string           ascii_REQUIRED4;
/// typedef utf8string           utf8val_REQUIRED4;
///
/// typedef opaque FedFsUuid[16];
///
/// struct FedFsNsdbName {
///         unsigned int         port;
///         utf8val_REQUIRED4     hostname;
/// };
///
/// typedef ascii_REQUIRED4 FedFsPathComponent;
/// typedef FedFsPathComponent FedFsPathName<>;
///
/// struct FedFsFsn {
///         FedFsUuid            fsnUuid;
///         FedFsNsdbName        nsdbName;
/// };
///
/// enum FedFsFslType {
///  FEDFS_NFS_FSL = 0
/// };
///
/// struct FedFsNfsFsl {
///         FedFsUuid            fslUuid;
///         unsigned int         port;
///         utf8val_REQUIRED4    hostname;
///         FedFsPathName        path;
/// };
///
/// union FedFsFsl switch(FedFsFslType type) {
```

```
///   case FEDFS_NFS_FSL:
///           FedFsNfsFsl            nfsFsl;
/// };
///
/// enum FedFsPathType {
///  FEDFS_PATH_SYS = 0,
///  FEDFS_PATH_NFS = 1
/// };
///
/// union FedFsPath switch(FedFsPathType type) {
///  case FEDFS_PATH_SYS: /* administrative path */
///           FedFsPathName          adminPath;
///  case FEDFS_PATH_NFS: /* NFS namespace path */
///           FedFsPathName          nfsPath;
/// };
///
/// struct FedFsCreateArgs {
///           FedFsPath              path;
///           FedFsFsn               fsn;
/// };
///
/// enum FedFsResolveType {
///  FEDFS_RESOLVE_NONE  = 0,
///  FEDFS_RESOLVE_CACHE = 1,
///  FEDFS_RESOLVE_NSDB  = 2
/// };
///
/// struct FedFsLookupArgs {
///           FedFsPath              path;
///           FedFsResolveType       resolve;
/// };
///
/// struct FedFsLookupResOk {
///           FedFsFsn               fsn;
///           FedFsFsl               fsl<>;
/// };
///
/// struct FedFsLookupResReferralVal {
///           FedFsNsdbName          targetNsdb;
///           unsigned int           ldapResultCode;
/// };
///
/// union FedFsLookupRes switch (FedFsStatus status) {
///  case FEDFS_OK:
///  case FEDFS_ERR_NO_CACHE_UPDATE:
///           FedFsLookupResOk          resok;
///  case FEDFS_ERR_NSDB_LDAP_VAL:
///           unsigned int              ldapResultCode;
```

```
///   case FEDFS_ERR_NSDB_LDAP_REFERRAL:
///   case FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL:
///         FedFsNsdbName              targetNsdb;
///   case FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL:
///         FedFsLookupResReferralVal  resReferralVal;
///   default:
///         void;
/// };
///
/// enum FedFsConnectionSec {
///  FEDFS_SEC_NONE = 0,
///  FEDFS_SEC_TLS = 1 /* StartTLS mechanism; RFC4513, Section 3 */
/// };
///
/// union FedFsNsdbParams switch (FedFsConnectionSec secType) {
///  case FEDFS_SEC_TLS:
///         opaque                    secData<>;
///  default:
///         void;
/// };
///
/// struct FedFsSetNsdbParamsArgs {
///         FedFsNsdbName             nsdbName;
///         FedFsNsdbParams           params;
/// };
///
/// union FedFsGetNsdbParamsRes switch (FedFsStatus status) {
///  case FEDFS_OK:
///         FedFsNsdbParams           params;
///  default:
///         void;
/// };
///
/// union FedFsGetLimitedNsdbParamsRes switch (FedFsStatus status) {
///  case FEDFS_OK:
///         FedFsConnectionSec        secType;
///  default:
///         void;
/// };
///
/// program FEDFS_PROG {
///  version FEDFS_V1 {
///       void FEDFS_NULL(void) = 0;
///       FedFsStatus FEDFS_CREATE_JUNCTION(
///                   FedFsCreateArgs) = 1;
///       FedFsStatus FEDFS_DELETE_JUNCTION(
///                   FedFsPath) = 2;
///       FedFsLookupRes FEDFS_LOOKUP_JUNCTION(
```

```
///                    FedFsLookupArgs) = 3;
///        FedFsStatus FEDFS_CREATE_REPLICATION(
///                    FedFsCreateArgs) = 7;
///        FedFsStatus FEDFS_DELETE_REPLICATION(
///                    FedFsPath) = 8;
///        FedFsLookupRes FEDFS_LOOKUP_REPLICATION(
///                    FedFsLookupArgs) = 9;
///        FedFsStatus FEDFS_SET_NSDB_PARAMS(
///                    FedFsSetNsdbParamsArgs) = 4;
///        FedFsGetNsdbParamsRes FEDFS_GET_NSDB_PARAMS(
///                    FedFsNsdbName) = 5;
///        FedFsGetLimitedNsdbParamsRes FEDFS_GET_LIMITED_NSDB_PARAMS(
///                    FedFsNsdbName) = 6;
///   } = 1;
/// } = 100418;
```

   <CODE ENDS>

3. Error Values

   The results of successful operations will consist of a status of
   FEDFS_OK. The results of unsuccessful operations will begin with a
   status, other than FEDFS_OK, that indicates the reason why the
   operation failed.

   Many of the error status names and meanings (and the prose for their
   descriptions) are taken from the specification for NFSv4 [3530bis].
   Note, however, that the numeric values for the status codes are
   different. For example, the name and meaning of FEDFS_ERR_ACCESS was
   inspired by NFSv4's NFS4ERR_ACCESS, but their numeric values are
   different.

   The status of an unsuccessful operation will generally only indicate
   the first error encountered during the attempt to execute the
   operation.

   FEDFS_OK: No errors were encountered. The operation was a success.

   FEDFS_ERR_ACCESS: Permission denied. The caller does not have the
      correct permission to perform the requested operation.

   FEDFS_ERR_BADCHAR: A UTF-8 string contains a character which is not
      supported by the server in the context in which it being used.

   FEDFS_ERR_BADNAME: A name string in a request consisted of valid
      UTF-8 characters supported by the server, but the name is not
      supported by the server as a valid name for the current operation.

   FEDFS_ERR_NAMETOOLONG: Returned when the pathname in an operation
      exceeds the server's implementation limit.

   FEDFS_ERR_LOOP: Returned when too many symbolic links were
      encountered in resolving pathname.

   FEDFS_ERR_BADXDR: The server encountered an XDR decoding error while
      processing an operation.

   FEDFS_ERR_EXIST: The junction specified already exists.

   FEDFS_ERR_INVAL: Invalid argument for an operation.

   FEDFS_ERR_IO: A hard error occurred while processing the requested
      operation.

   FEDFS_ERR_NOSPC: The requested operation would have caused the
      server's filesystem to exceed some limit (for example, if there is
      a fixed number of junctions per fileset or per server).

   FEDFS_ERR_NOTJUNCT: The caller specified a path that does not end in
      a junction as the operand for an operation that requires the last
      component of the path to be a junction.

   FEDFS_ERR_NOTLOCAL: The caller specified a path that contains a
      junction in any position other than the last component.

   FEDFS_ERR_PERM: The operation was not allowed because the caller is
      either not a privileged user or not the owner of an object that
      would be modified by the operation.

   FEDFS_ERR_ROFS: A modifying operation was attempted on a read-only
      filesystem.

   FEDFS_ERR_SVRFAULT: An unanticipated non-protocol error occurred on
      the server.

   FEDFS_ERR_NSDB_ROUTE: The fileserver was unable to find a route to
      the NSDB.

   FEDFS_ERR_NSDB_DOWN: The fileserver determined that the NSDB was
      down.

FEDFS_ERR_NSDB_CONN: The fileserver was unable to establish a
   connection with the NSDB.

FEDFS_ERR_NSDB_AUTH: The fileserver was unable to authenticate and
   establish a secure connection with the NSDB.

FEDFS_ERR_NSDB_LDAP: An LDAP error occurred on the connection
   between the fileserver and NSDB.

FEDFS_ERR_NSDB_LDAP_VAL: Indicates the same error as
   FEDFS_ERR_NSDB_LDAP, and allows the LDAP protocol error value to
   be returned back to an ADMIN protocol client.

FEDFS_ERR_NSDB_NONCE: The fileserver was unable to locate the NCE in
   the appropriate NSDB.

FEDFS_ERR_NSDB_NOFSN: The fileserver was unable to locate the given
   FSN in the appropriate NSDB.

FEDFS_ERR_NSDB_NOFSL: The fileserver was unable to locate any FSLs
   for the given FSN in the appropriate NSDB.

FEDFS_ERR_NSDB_RESPONSE: The fileserver received a malformed
   response from the NSDB. This includes situations when an NSDB
   entry (e.g., FSN or FSL) is missing a required attribute.

FEDFS_ERR_NSDB_FAULT: An unanticipated error related to the NSDB
   occurred.

FEDFS_ERR_NSDB_PARAMS: The fileserver does not have any connection
   parameters on record for the specified NSDB.

FEDFS_ERR_NSDB_LDAP_REFERRAL: The fileserver received an LDAP
   referral that it was unable to follow.

FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL: Indicates the same error as
   FEDFS_ERR_NSDB_LDAP_REFERRAL, and allows the LDAP protocol error
   value to be returned back to an ADMIN protocol client.

FEDFS_ERR_NSDB_LDAP_REFERRAL_NOTFOLLOWED: The fileserver received an
   LDAP referral that it chose not to follow, either because the
   fileserver does not support following LDAP referrals or LDAP
   referral following is disabled.

FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL: The fileserver received an LDAP
   referral that it chose not to follow because the fileserver had no
   NSDB parameters for the NSDB targeted by the LDAP referral.

FEDFS_ERR_PATH_TYPE_UNSUPP: The fileserver does not support the
   specified FedFsPathType value.

FEDFS_ERR_NOTSUPP: The fileserver does not support the specified
   procedure.

FEDFS_ERR_DELAY: The fileserver initiated the request, but was not
   able to complete it in a timely fashion. The ADMIN protocol
   client should wait and then try the request with a new RPC
   transaction ID.

FEDFS_ERR_NO_CACHE: The fileserver does not implement an FSN-to-FSL
   cache.

FEDFS_ERR_UNKNOWN_CACHE: The software receiving the ONC RPC request
   is unaware if the fileserver implements an FSN-to-FSL cache or
   unable to communicate with the FSN-to-FSL cache if it exists.

FEDFS_ERR_NO_CACHE_UPDATE: The fileserver was unable to update its
   FSN-to-FSL cache.

4. Data Types

   The basic data types defined above are formatted as follows:

   FedFsUuid: A universally unique identifier (UUID) as described in
      [RFC4122] as a version 4 UUID. The UUID MUST be formatted in
      network byte order.

   FedFsNsdbName: A (hostname, port) pair.

      The hostname is a variable length UTF-8 string that represents an
      NSDB's network location in DNS name notation. It SHOULD be
      prepared using the server4 rules defined in Chapter 12
      "Internationalization" of [3530bis]. The DNS name MUST be
      represented using a fully qualified domain name.

      The port value in the FedFsNsdbName indicates the LDAP port on the
      NSDB (see [RFC4511]). The value MUST be in the range 0 to 65535.
      A value of 0 indicates that the standard LDAP port number, 389,
      MUST be assumed.

      FSNs are immutable and invariant. The attributes of an FSN,
      including the fedfsNsdbName, are expected to remain constant.
      Therefore, a FedFsNsdbName MUST NOT contain a network address,
      such as an IPv4 or IPv6 address, as this would indefinitely assign
      the network address.

FedFsPathComponent: A case sensitive UTF-8 string containing a
   filesystem path component. It MUST be prepared using the
   component4 rules defined in Chapter 12 "Internationalization" of
   [3530bis].

FedFsPathName: A variable length array of FedFsPathComponent values
   representing a filesystem path. The path's first component is
   stored at the first position of the array, the second component is
   stored at the second position of the array, and so on.

   The path "/" MUST be encoded as an array with zero components.

   A FedFsPathName MUST NOT contain any zero-length components.

FedFsPath: A pathname container. The format and semantics of the
   pathname are defined by the FedFsPathType value.

FedFsPathType: The type specific description of a pathname.

   A FEDFS_PATH_SYS is an implementation dependent administrative
   pathname. For example, it could be a local file system path.

   A FEDFS_PATH_NFS is a pathname in the NFSv4 server's single-server
   namespace.

FedFsNsdbParams: A set of parameters for connecting to an NSDB.
   Conceptually the fileserver contains a data structure that maps an
   NSDB name (DNS name and port value) to these LDAP connection
   parameters.

   The secType field indicates the security mechanism that MUST be
   used to protect all connections to the NSDB with the connection
   parameters.

   A value of FEDFS_SEC_NONE indicates that a transport security
   mechanism MUST NOT be used when connecting to the NSDB. In this
   case, the secData array will have a length of zero.

   A value of FEDFS_SEC_TLS indicates that the StartTLS security
   mechanism [RFC4513] MUST be used to protect all connections to the
   NSDB. In this case, the secData array will contain an X.509v3
   root certificate in binary DER format [RFC5280] fulfilling the TLS
   requirement that root keys be distributed independently from the
   TLS protocol. The certificate MUST be used by the fileserver as a
   Trust Anchor to validate the NSDB's TLS server certificate list
   chain (see section 7.4.2 of [RFC5246]) and thus authenticate the
   identitiy of the NSDB. The certificate could be that of a
   certificate authority or a self-signed certificate. To ensure

that this security configuration information does not cause
vulnerabilities for other services, trust anchors provided through
secData MUST only be used for the NSDB service (as opposed to
being installed as system-wide trust anchors for other services).
Most popular TLS libraries provide ways in which this can be done
such as denoting a private file system location for the
certificates.

4.1. FedFsNsdbName Equality

Two FedFsNsdbNames are considered equal if their respective hostname
and port fields contain the same values. The only exception to this
rule is that a value of 0 in the port field always matches the
standard LDAP port number, 389.

Therefore, the FedFsNsdbName "(nsdb.example.com, 0)" is considered
equal to "(nsdb.example.com, 389)" but not equal to
"(nsdb.example.com, 1066)" since the port numbers are different, or
"(nsdb.foo.example.com, 389)" since the hostnames are different.

5. Procedures

The procedures defined in Section 2 are described in detail in the
following sections.

Fileservers that participate as "internal" nodes in the federated
namespace MUST implement the following procedures:

    FEDFS_NULL
    FEDFS_CREATE_JUNCTION
    FEDFS_DELETE_JUNCTION
    FEDFS_LOOKUP_JUNCTION
    FEDFS_SET_NSDB_PARAMS
    FEDFS_GET_NSDB_PARAMS
    FEDFS_GET_LIMITED_NSDB_PARAMS

and SHOULD implement the following procedures:

    FEDFS_CREATE_REPLICATION
    FEDFS_DELETE_REPLICATION
    FEDFS_LOOKUP_REPLICATION

Fileservers that participate as "leaf" nodes in the namespace (i.e.,
fileservers that host filesets that are the target of junctions, but
that do not contain any junctions) are not required to implement any
of these operations.

Operations that modify the state of a replicated fileset MUST result
in the update of all of the replicas in a consistent manner. Ideally
all of the replicas SHOULD be updated before any operation returns.
If one or more of the replicas are unavailable, the operation MAY
succeed, but the changes MUST be applied before the unavailable
replicas are brought back online. We assume that replicas are
updated via some protocol that permits state changes to be reflected
consistently across the set of replicas in such a manner that the
replicas will converge to a consistent state within a bounded number
of successful message exchanges between the servers hosting the
replicas.

5.1. FEDFS_NULL

5.1.1. Synopsis

The standard NULL procedure.

5.1.2. Description

The null RPC, which is included, by convention, in every ONC RPC
protocol. This procedure does not take any arguments and does not
produce a result.

5.1.3. Errors

None.

5.2. FEDFS_CREATE_JUNCTION

5.2.1. Synopsis

Create a new junction from some location on the server (defined as a
pathname) to an FSN.

5.2.2. Description

This operation creates a junction from a server-relative path to a
(potentially) remote fileset named by the given FSN.

The junction directory on the server is identified by a pathname in
the form of an array of one or more UTF-8 path component strings. It
is not required that this path be accessible in any other manner
(e.g., to a file-access client). This path does not appear in the
federated namespace, except by coincidence; there is no requirement
that the global namespace parallel the server namespace, nor is it
required that this path be relative to the server pseudo-root. It
does not need to be a path that is accessible via NFS (although the

junction will be of limited utility if the directory specified by the
path is not also accessible via NFS).

If the fileset is read-only, then this operation MUST indicate this
with a status of FEDFS_ERR_ROFS.

If the path contains a character that is not supported by the server,
then status FEDFS_ERR_BADCHAR MUST be returned.

The path is REQUIRED to exist and be completely local to the server.
It MUST NOT contain a junction. If the last component of the path is
a junction (i.e., this operation is attempting to create a junction
where one already exists), then this operation MUST return the error
FEDFS_ERR_EXIST (even if the requested junction is identical to the
current junction). If any other component of the path is a junction,
then this operation MUST fail with status FEDFS_ERR_NOTLOCAL. The
path might contain a symbolic link (if supported by the local
server), but the traversal of the path MUST remain within the server-
local namespace.

If any component of the path does not exist, then the operation MUST
fail with status FEDFS_ERR_INVAL.

The server MAY enforce the local permissions on the path, including
the final component. If a server wishes to report that a path cannot
be traversed because of insufficient permissions, or the final
component is an unexecutable or unwritable directory, then the
operation MUST fail with status FEDFS_ERR_ACCESS.

The operation SHOULD fail with status FEDFS_ERR_NSDB_PARAMS if the
fileserver does not have any connection parameters on record for the
specified NSDB, or the server may allow the operation to proceed
using some set of default NSDB connection parameters.

The association between the path and the FSN MUST be durable before
the operation returns successfully. If the operation return codes
indicates success, then the junction was successfully created and is
immediately accessible.

If successful, subsequent references via NFSv4.0 [3530bis] or NFSv4.1
[RFC5661] clients to the directory that has been replaced by the
junction will result in a referral to a current location of the
target fileset [FEDFS-NSDB].

The effective permissions of the directory that is converted, by this
operation, into a junction are the permissions of the root directory
of the target fileset. The original permissions of the directory
(and any other attributes it might have) are subsumed by the

junction.

This operation does not create a fileset at the location targeted by the junction. If the target fileset does not exist, the junction will still be created. An NFS client will discover the missing fileset when it traverses the junction.

5.2.3. Errors

       FEDFS_ERR_ACCESS
       FEDFS_ERR_BADCHAR
       FEDFS_ERR_BADNAME
       FEDFS_ERR_NAMETOOLONG
       FEDFS_ERR_LOOP
       FEDFS_ERR_BADXDR
       FEDFS_ERR_EXIST
       FEDFS_ERR_INVAL
       FEDFS_ERR_IO
       FEDFS_ERR_NOSPC
       FEDFS_ERR_NOTLOCAL
       FEDFS_ERR_PERM
       FEDFS_ERR_ROFS
       FEDFS_ERR_SVRFAULT
       FEDFS_ERR_PATH_TYPE_UNSUPP
       FEDFS_ERR_NOTSUPP
       FEDFS_ERR_DELAY

5.3.   FEDFS_DELETE_JUNCTION

5.3.1. Synopsis

Delete an existing junction from some location on the server (defined as a pathname).

5.3.2. Description

This operation removes a junction specified by a server-relative path.

As with FEDFS_CREATE_JUNCTION, the junction on the server is identified by a pathname in the form of an array of one or more UTF-8 path component strings. It is not required that this path be accessible in any other manner (e.g., to a file-access client). This path does not appear in the federated namespace, except by coincidence; there is no requirement that the global namespace reflect the server namespace, nor is it required that this path be relative to the server pseudo-root. It does not need to be a path that is accessible via NFS.

If the fileset is read-only, then this operation MUST indicate this
with a status of FEDFS_ERR_ROFS.

If the path contains a character that is not supported by the server,
then status FEDFS_ERR_BADCHAR MUST be returned.

The path used to delete a junction might not be the same path that
was used to create the junction. If the namespace on the server has
changed, then the junction might now appear at a different path than
where it was created. If there is more than one valid path to the
junction, any of them can be used.

The path is REQUIRED to exist and be completely local to the server.
It MUST NOT contain a junction, except as the final component, which
MUST be a junction. If any other component of the path is a
junction, then this operation MUST fail with status
FEDFS_ERR_NOTLOCAL. If the last component of the path is not a
junction then this operation MUST return status FEDFS_ERR_NOTJUNCT.
The path might contain a symbolic link (if supported by the local
server), but the traversal of the path MUST remain within the server-
local namespace.

The server MAY enforce the local permissions on the path, including
the final component. If a server wishes to report that a path cannot
be traversed because of insufficient permissions, or the final
component is an unexecutable or unwritable directory, then the
operation MUST fail with status FEDFS_ERR_ACCESS.

The removal of the association between the path and the FSN MUST be
durable before the operation returns successfully. If the operation
return codes indicates success, then the junction was successfully
destroyed.

The effective permissions and other attributes of the directory that
is restored by this operation SHOULD be identical to their value
prior to the creation of the junction.

After removal of the junction, the fileserver MAY check if any of its
existing junctions reference the NSDB specified in the removed
junction's FSN. If the NSDB is not referenced, the fileserver MAY
delete the connection parameters of the unreferenced NSDB.

5.3.3. Errors

FEDFS_ERR_ACCESS
FEDFS_ERR_BADCHAR

```
       FEDFS_ERR_BADNAME
       FEDFS_ERR_NAMETOOLONG
       FEDFS_ERR_LOOP
       FEDFS_ERR_BADXDR
       FEDFS_ERR_INVAL
       FEDFS_ERR_IO
       FEDFS_ERR_NOTJUNCT
       FEDFS_ERR_NOTLOCAL
       FEDFS_ERR_PERM
       FEDFS_ERR_ROFS
       FEDFS_ERR_SVRFAULT
       FEDFS_ERR_PATH_TYPE_UNSUPP
       FEDFS_ERR_NOTSUPP
       FEDFS_ERR_DELAY
```

5.4. FEDFS_LOOKUP_JUNCTION

5.4.1. Synopsis

   Query the server to discover the current value of the junction (if
   any) at a given path in the server namespace.

5.4.2. Description

   This operation queries a server to determine whether a given path
   ends in a junction, and if so, the FSN to which the junction refers
   and the filerserver's ability to resolve the junction.

   Ordinary NFSv4 operations do not provide any general mechanism to
   determine whether an object is a junction -- there is no encoding
   specified by the NFSv4 protocol that can represent this information.

   As with FEDFS_CREATE_JUNCTION, the pathname MUST be in the form of an
   array of one or more UTF-8 path component strings. It is not
   required that this path be accessible in any other manner (e.g., to a
   file-access client). This path does not appear in the federated
   namespace, except by coincidence; there is no requirement that the
   global namespace reflect the server namespace, nor is it required
   that this path be relative to the server pseudo-root. It does not
   need to be a path that is accessible via NFS.

   If the path contains a character that is not supported by the server,
   then status FEDFS_ERR_BADCHAR MUST be returned.

   The path used to lookup a junction might not be the same path that
   was used to create the junction. If the namespace on the server has
   changed, then a junction might now appear at a different path than
   where it was created. If there is more than one valid path to the

junction, any of them might be used.

The path is REQUIRED to exist and be completely local to the server.
It MUST NOT contain a junction, except as the final component. If
any other component of the path is a junction, then this operation
MUST fail with status FEDFS_ERR_NOTLOCAL. If the last component of
the path is not a junction then this operation MUST return the status
FEDFS_ERR_NOTJUNCT. The path might contain a symbolic link (if
supported by the local server), but the traversal of the path MUST
remain within the server-local namespace.

The server MAY enforce the local permissions on the path, including
the final component. If a server wishes to report that a path cannot
be traversed because of insufficient permissions, or the final
component is an unexecutable or unwritable directory, then the
operation MUST fail with status FEDFS_ERR_ACCESS.

If the junction exists, the resolve parameter allows for testing the
fileserver's ability to resolve the junction. If the junction does
not exist, the fileserver will ignore the resolve parameter.

If the junction exists and the resolve parameter is set to
FEDFS_RESOLVE_NONE, the fileserver MUST NOT attempt to resolve the
FSN. This will allow the administrator to obtain the junction's FSN
even if the resolution would fail. Therefore on success, the result
of a FEDFS_RESOLVE_NONE call will return a 0 length fsl list in the
FedFsLookupResOk structure.

If the junction exists and the resolve parameter is set to
FEDFS_RESOLVE_CACHE, the fileserver MUST attempt to resolve the FSN
using its FSL cache, if one exists. The fileserver MUST NOT resolve
the FSN by contacting the appropriate NSDB. If the fileserver's
cache does not have a mapping for the FSN in question, the result of
the operation MUST be FEDFS_OK with 0 elements in the
FedFsLookupResOk structure's fsl array. The operation MAY fail with
status FEDFS_ERR_NO_CACHE if the fileserver does not contain an FSN-
to-FSL cache or with status FEDFS_ERR_UNKNOWN_CACHE if the state of
the cache is unknown.

If the junction exists and the resolve parameter is set to
FEDFS_RESOLVE_NSDB, the fileserver MUST attempt to resolve the FSN by
contacting the appropriate NSDB. The FSN MUST NOT be resolved using
cached information. The resolution MAY fail with
FEDFS_ERR_NSDB_ROUTE, FEDFS_ERR_NSDB_DOWN, FEDFS_ERR_NSDB_CONN,
FEDFS_ERR_NSDB_AUTH, FEDFS_ERR_NSDB_LDAP, FEDFS_ERR_NSDB_LDAP_VAL,
FEDFS_ERR_NSDB_NOFSN, FEDFS_ERR_NSDB_NOFSL, FEDFS_ERR_NSDB_NONCE,
FEDFS_ERR_NSDB_RESPONSE, FEDFS_ERR_NSDB_FAULT,
FEDFS_ERR_NSDB_LDAP_REFERRAL, FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL,

FEDFS_ERR_NSDB_LDAP_REFERRAL_NOTFOLLOWED, or
FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL, depending on the nature of the
failure.

In the case of a LDAP failure, the fileserver MUST return either
FEDFS_ERR_NSDB_LDAP or FEDFS_ERR_NSDB_LDAP_VAL. FEDFS_ERR_NSDB_LDAP
indicates that an LDAP protocol error occurred during the resolution.
FEDFS_ERR_NSDB_LDAP_VAL also indicates that an LDAP protocol error
occurred during the resolution and allows the LDAP protocol error
value to be returned in the FedFsLookupRes's ldapResultCode field
(see the resultCode values in Section 4.1.9 of [RFC4511]).

If the NSDB responds with an LDAP referral, either the Referral type
defined in Section 4.1.10 of [RFC4511] or the SearchResultReference
type defined in Section 4.5.3 of [RFC4511], the fileserver SHOULD
process the LDAP referral using the same policies as the fileserver's
file-access protocol server. The fileserver MUST indicate a failure
while processing the LDAP referral using
FEDFS_ERR_NSDB_LDAP_REFERRAL, FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL,
FEDFS_ERR_NSDB_LDAP_REFERRAL_NOTFOLLOWED, or
FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL. The
FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL is analogous to the
FEDFS_ERR_NSDB_LDAP_VAL error and allows the LDAP protocol error
value to be returned in the FedFsLookupResReferralVal's
ldapResultCode field. The FEDFS_ERR_NSDB_LDAP_REFERRAL and
FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL errors errors allow the NSDB
targeted by the LDAP referral to be returned in the FedFsLookupRes's
targetNsdb field. Similarly, the FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL
error includes this information in the FedFsLookupResReferralVal's
targetNsdb.

If the fileserver has a cache of FSL records, the process of
resolving an FSN using an NSDB SHOULD result in the cache being
updated. A failure to update the cache MAY be indicated with the
FEDFS_ERR_NO_CACHE_UPDATE status value, or the operation may complete
successfully.

When updating the cache, new FSLs for the given FSN SHOULD be added
to the cache and deleted FSLs SHOULD be removed from the cache. This
behavior is desirable because it allows an administrator to
proactively request that the fileserver refresh its FSL cache. For
example, the administrator might like to refresh the fileserver's
cache when changes are made to an FSN's FSLs.

If the junction is resolved, the fileserver will include a list of
UUIDs for the FSN's FSLs in the FedFsLookupResOk structure's fsl
array.

5.4.3. Errors

        FEDFS_ERR_ACCESS
        FEDFS_ERR_BADCHAR
        FEDFS_ERR_BADNAME
        FEDFS_ERR_NAMETOOLONG
        FEDFS_ERR_LOOP
        FEDFS_ERR_BADXDR
        FEDFS_ERR_INVAL
        FEDFS_ERR_IO
        FEDFS_ERR_NOTJUNCT
        FEDFS_ERR_NOTLOCAL
        FEDFS_ERR_PERM
        FEDFS_ERR_SVRFAULT
        FEDFS_ERR_NSDB_ROUTE
        FEDFS_ERR_NSDB_DOWN
        FEDFS_ERR_NSDB_CONN
        FEDFS_ERR_NSDB_AUTH
        FEDFS_ERR_NSDB_LDAP
        FEDFS_ERR_NSDB_LDAP_VAL
        FEDFS_ERR_NSDB_NONCE
        FEDFS_ERR_NSDB_NOFSN
        FEDFS_ERR_NSDB_NOFSL
        FEDFS_ERR_NSDB_RESPONSE
        FEDFS_ERR_NSDB_FAULT
        FEDFS_ERR_NSDB_PARAMS
        FEDFS_ERR_NSDB_LDAP_REFERRAL
        FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL
        FEDFS_ERR_NSDB_LDAP_REFERRAL_NOTFOLLOWED
        FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL
        FEDFS_ERR_PATH_TYPE_UNSUPP
        FEDFS_ERR_NOTSUPP
        FEDFS_ERR_DELAY
        FEDFS_ERR_NO_CACHE
        FEDFS_ERR_UNKNOWN_CACHE
        FEDFS_ERR_NO_CACHE_UPDATE

5.5. FEDFS_CREATE_REPLICATION

5.5.1. Synopsis

   Set an FSN representing the replication information for the fileset
   containing the pathname.

5.5.2. Description

   This operation indicates the replication information to be returned
   for a particular fileset. An NFSv4 client might request fs_locations

or fs_locations_info at any time to detect other copies of this
fileset, and this operation supports this by supplying the FSN the
fileserver should use to respond. This FSN should be associated with
the entire fileset in which the path resides, and should be used to
satisfy fs_locations or fs_locations_info attribute requests whenever
no junction is being accessed; if a junction is being accessed, the
FSN specified by FEDFS_CREATE_JUNCTION will take precedence. Setting
the replication FSN on a fileset that already has a replication FSN
set is allowed.

This operation differs from FEDFS_CREATE_JUNCTION in that it controls
a fileset-wide attribute not associated with a junction.

The server SHOULD permit this operation even on read-only filesets,
but MUST return FEDFS_ERR_ROFS if this is not possible.

If the path contains a character that is not supported by the server,
then status FEDFS_ERR_BADCHAR MUST be returned.

The path is REQUIRED to exist and be completely local to the server.
It MUST NOT contain a junction. If any component of the path is a
junction, then this operation MUST fail with status
FEDFS_ERR_NOTLOCAL. The path might contain a symbolic link (if
supported by the local server), but the traversal of the path MUST
remain within the server-local namespace.

The server MAY enforce the local permissions on the path, including
the final component. If a server wishes to report that a path cannot
be traversed because of insufficient permissions, or the final
component is an unexecutable or unwritable directory, then the
operation MUST fail with status FEDFS_ERR_ACCESS.

The operation SHOULD fail with status FEDFS_ERR_NSDB_PARAMS if the
fileserver does not have any connection parameters on record for the
specified NSDB, or the server may allow the operation to proceed
using some set of default NSDB connection parameters.

The same FSN value SHOULD be associated with all replicas of a
filesystem. Depending on the underlying representation, the FSN
associated with a filesystem might or might not be replicated
automatically with the filesystem replication mechanism. Therefore
if FEDFS_CREATE_REPLICATION is used on one replica of a filesystem,
it SHOULD be used on all replicas.

5.5.3. Errors

```
FEDFS_ERR_ACCESS
FEDFS_ERR_BADCHAR
FEDFS_ERR_BADNAME
FEDFS_ERR_NAMETOOLONG
FEDFS_ERR_LOOP
FEDFS_ERR_BADXDR
FEDFS_ERR_EXIST
FEDFS_ERR_INVAL
FEDFS_ERR_IO
FEDFS_ERR_NOSPC
FEDFS_ERR_NOTLOCAL
FEDFS_ERR_PERM
FEDFS_ERR_ROFS
FEDFS_ERR_SVRFAULT
FEDFS_ERR_PATH_TYPE_UNSUPP
FEDFS_ERR_NOTSUPP
FEDFS_ERR_DELAY
```

5.6. FEDFS_DELETE_REPLICATION

5.6.1. Synopsis

Remove the replication information for the fileset containing the
pathname.

5.6.2. Description

This operation removes any replication information from the fileset
in which the path resides, such that NFSv4 client requests for
fs_locations or fs_locations_info in the absence of a junction will
not be satisfied.

This operation differs from FEDFS_DELETE_JUNCTION in that it controls
a fileset-wide attribute not associated with a junction.

The server SHOULD permit this operation even on read-only filesets,
but MUST return FEDFS_ERR_ROFS if this is not possible.

If the path contains a character that is not supported by the server,
then status FEDFS_ERR_BADCHAR MUST be returned.

The path is REQUIRED to exist and be completely local to the server.
It MUST NOT contain a junction. If any component of the path is a
junction, then this operation MUST fail with status
FEDFS_ERR_NOTLOCAL.

The server MAY enforce the local permissions on the path, including
the final component. If a server wishes to report that a path cannot

be traversed because of insufficient permissions, or the final
component is an unexecutable or unwritable directory, then the
operation MUST fail with status FEDFS_ERR_ACCESS.

5.6.3. Errors

       FEDFS_ERR_ACCESS
       FEDFS_ERR_BADCHAR
       FEDFS_ERR_BADNAME
       FEDFS_ERR_NAMETOOLONG
       FEDFS_ERR_LOOP
       FEDFS_ERR_BADXDR
       FEDFS_ERR_INVAL
       FEDFS_ERR_IO
       FEDFS_ERR_NOTJUNCT
       FEDFS_ERR_NOTLOCAL
       FEDFS_ERR_PERM
       FEDFS_ERR_ROFS
       FEDFS_ERR_SVRFAULT
       FEDFS_ERR_PATH_TYPE_UNSUPP
       FEDFS_ERR_NOTSUPP
       FEDFS_ERR_DELAY

5.7. FEDFS_LOOKUP_REPLICATION

5.7.1. Synopsis

   Query the server to discover the current replication information (if
   any) at the given path.

5.7.2. Description

   This operation queries a server to determine whether a fileset
   containing the given path has replication information associated with
   it, and if so, the FSN for that replication information.

   This operation differs from FEDFS_LOOKUP_JUNCTION in that it inquires
   about a fileset-wide attribute not associated with a junction.

   If the path contains a character that is not supported by the server,
   then status FEDFS_ERR_BADCHAR MUST be returned.

   The path is REQUIRED to exist and be completely local to the server.
   It MUST NOT contain a junction. If any component of the path is a
   junction, then this operation MUST fail with status
   FEDFS_ERR_NOTLOCAL.

   The server MAY enforce the local permissions on the path, including

the final component. If a server wishes to report that a path cannot be traversed because of insufficient permissions, or the final component is an unexecutable or unwritable directory, then the operation MUST fail with status FEDFS_ERR_ACCESS.

Interpretation of the 'resolve' parameter and the procedure's results shall be the same as specified in Section 5.4 for the FEDFS_LOOKUP_JUNCTION operation.

5.7.3. Errors

```
     FEDFS_ERR_ACCESS
     FEDFS_ERR_BADCHAR
     FEDFS_ERR_BADNAME
     FEDFS_ERR_NAMETOOLONG
     FEDFS_ERR_LOOP
     FEDFS_ERR_BADXDR
     FEDFS_ERR_INVAL
     FEDFS_ERR_IO
     FEDFS_ERR_NOTJUNCT
     FEDFS_ERR_NOTLOCAL
     FEDFS_ERR_PERM
     FEDFS_ERR_SVRFAULT
     FEDFS_ERR_NSDB_ROUTE
     FEDFS_ERR_NSDB_DOWN
     FEDFS_ERR_NSDB_CONN
     FEDFS_ERR_NSDB_AUTH
     FEDFS_ERR_NSDB_LDAP
     FEDFS_ERR_NSDB_LDAP_VAL
     FEDFS_ERR_NSDB_NONCE
     FEDFS_ERR_NSDB_NOFSN
     FEDFS_ERR_NSDB_NOFSL
     FEDFS_ERR_NSDB_RESPONSE
     FEDFS_ERR_NSDB_FAULT
     FEDFS_ERR_NSDB_PARAMS
     FEDFS_ERR_NSDB_LDAP_REFERRAL
     FEDFS_ERR_NSDB_LDAP_REFERRAL_VAL
     FEDFS_ERR_NSDB_LDAP_REFERRAL_NOTFOLLOWED
     FEDFS_ERR_NSDB_PARAMS_LDAP_REFERRAL
     FEDFS_ERR_PATH_TYPE_UNSUPP
     FEDFS_ERR_NOTSUPP
     FEDFS_ERR_DELAY
     FEDFS_ERR_NO_CACHE
     FEDFS_ERR_UNKNOWN_CACHE
```

5.8. FEDFS_SET_NSDB_PARAMS

5.8.1. Synopsis

   Set the connection parameters for the specified NSDB.

5.8.2. Description

   This operations allows the administrator to set the connection
   parameters for a given NSDB.

   If a record for the given NSDB does not exist, a new record is
   created with the specified connection parameters.

   If a record for the given NSDB does exist, the existing connection
   parameters are replaced with the specified connection parameters.

   An NSDB is specified using a FedFsNsdbName. The rules in Section 4.1
   define when two FedFsNsdbNames are considered equal.

   The given NSDB need not be referenced by any junctions on the
   fileserver. This situation will occur when connection parameters for
   a new NSDB are installed.

   The format of the connection parameters is described above.

   On success, this operation returns FEDFS_OK. When the operation
   returns, the new connection parameters SHOULD be used for all
   subsequent LDAP connections to the given NSDB. Existing connections
   MAY be terminated and re-established using the new connection
   parameters. The connection parameters SHOULD be durable across
   fileserver reboots.

   On failure, an error value indicating the type of error is returned.
   If the operation's associated user does not have sufficient
   permissions to create/modify NSDB connection parameters, the
   operation MUST return FEDFS_ERR_ACCESS.

5.8.3. Errors

      FEDFS_ERR_ACCESS
      FEDFS_ERR_BADCHAR
      FEDFS_ERR_BADNAME
      FEDFS_ERR_BADXDR
      FEDFS_ERR_INVAL
      FEDFS_ERR_IO

```
FEDFS_ERR_NOSPC
FEDFS_ERR_SVRFAULT
FEDFS_ERR_NOTSUPP
FEDFS_ERR_DELAY
```

5.9. FEDFS_GET_NSDB_PARAMS

5.9.1. Synopsis

   Get the connection parameters for the specified NSDB.

5.9.2. Description

   This operations allows the administrator to retrieve connection
   parameters, if they exist, for the given NSDB.

   An NSDB is specified using a FedFsNsdbName. The rules in Section 4.1
   define when two FedFsNsdbNames are considered equal.

   A set of connection parameters is considered a match if their
   associated NSDB is equal (as defined above) to the operation's NSDB
   argument. Therefore, there is at most one set of connection
   parameters that can match the query described by this operation.

   The format of the connection parameters is described above.

   On success, this operation returns FEDFS_OK and the connection
   parameters on record for the given NSDB.

   On failure, an error value indicating the type of error is returned.
   This operation MUST return FEDFS_ERR_NSDB_PARAMS to indicate that
   there are no connection parameters on record for the given NSDB. If
   the operation's associated user does not have sufficient permissions
   to view NSDB connection parameters, the operation MUST return
   FEDFS_ERR_ACCESS.

5.9.3. Errors

```
FEDFS_ERR_ACCESS
FEDFS_ERR_BADCHAR
FEDFS_ERR_BADNAME
FEDFS_ERR_BADXDR
FEDFS_ERR_INVAL
FEDFS_ERR_IO
FEDFS_ERR_SVRFAULT
FEDFS_ERR_NSDB_PARAMS
```

```
FEDFS_ERR_NOTSUPP
FEDFS_ERR_DELAY
```

5.10. FEDFS_GET_LIMITED_NSDB_PARAMS

5.10.1. Synopsis

Get a limited subset of the connection parameters for the specified
NSDB.

5.10.2. Description

This operation allows the administrator to retrieve a limited subset
of information on the connection parameters, if they exist, for the
given NSDB.

A NSDB is specified using a FedFsNsdbName. The rules in Section 4.1
define when two FedFsNsdbNames are considered equal.

A set of connection parameters is considered a match if their
associated NSDB is equal (as defined above) to the operation's NSDB
argument. Therefore, there is at most one set of connection
parameters that can match the query described by this operation.

This operation returns a limited subset of the connection parameters.
Only the FedFsConnectionSec mechanism that is used to protect
communication between the fileserver and NSDB is returned.

Viewing the limited subset of NSDB connection parameters returned by
FEDFS_GET_LIMITED_NSDB_PARAMS MAY be a less privileged operation than
viewing the entire set of NSDB connection parameters returned by
FEDFS_GET_NSDB_PARAMS. For example, the full contents of an NSDB's
connection parameters could contain sensitive information for some
security mechanisms. FEDFS_GET_LIMITED_NSDB_PARAMS allows the
fileserver to communicate a subset of the connection parameters (the
security mechanism) to users with sufficient permissions without
revealing more sensitive information.

On success, this operation returns FEDFS_OK and the
FedFsConnectionSec value on record for the given NSDB.

On failure, an error value indicating the type of error is returned.
This operation MUST return FEDFS_ERR_NSDB_PARAMS to indicate that
there are no connection parameters on record for the given NSDB. If
the operation's associated user does not have sufficient permissions
to view the subset of NSDB connection parameters returned by this
procedure, the operation MUST return FEDFS_ERR_ACCESS.

5.10.3. Errors

        FEDFS_ERR_ACCESS
        FEDFS_ERR_BADCHAR
        FEDFS_ERR_BADNAME
        FEDFS_ERR_BADXDR
        FEDFS_ERR_INVAL
        FEDFS_ERR_IO
        FEDFS_ERR_SVRFAULT
        FEDFS_ERR_NSDB_PARAMS
        FEDFS_ERR_NOTSUPP
        FEDFS_ERR_DELAY

6. Security Considerations

   The Security Considerations of [RFC5531] apply to the protocol
   described in this document. The ONC RPC protocol supports
   authentication, integrity and privacy via the RPCSEC_GSS framework
   [RFC2203]. Fileservers which support the FedFS administration
   protocol described above MUST support RPCSEC_GSS.

   As with NFSv4.1 (see Section 2.2.1.1.1.1 of [RFC5661]), FedFS
   administration protocol clients and servers MUST support RPCSEC_GSS's
   integrity and authentication services. FedFS administration protocol
   servers MUST support RPCSEC_GSS's privacy service. FedFS
   administration protocol clients SHOULD support RPCSEC_GSS's privacy
   service. When RPCSEC_GSS is employed on behalf of the FedFS
   administration protocol, RPCSEC_GSS data integrity SHOULD be used.

   It is strongly RECOMMENDED that an Access Control Service be employed
   to restrict access to a fileserver's FedFS administration
   configuration data via the FedFS administrative protocol to prevent
   FedFS namespace corruption, and protect NSDB communication
   parameters.

   For example, when the FedFsNsdbParams secType field value
   FEDFS_SEC_TLS is chosen, the payload is used to provision the trust
   anchor root certificate for TLS secure communication between the
   fileserver and the NSDB. In this case, RPCSEC_GSS with data
   integrity SHOULD be employed along with an Access Control Service to
   restrict access to domain adminstrators

   FEDFS_GET_LIMITED_NSDB_PARAMS's interaction with the NSDB's
   connection parameters is discussed in Section 5.10.2.

7. IANA Considerations

   A range of ONC RPC program numbers were assigned for use by FedFS
   using the procedure described in Section 7.3 "Program Number
   Assignment" of [RFC5531]. The FedFS range is:

      IETF NFSv4 Working Group - FedFS 100418 - 100421

   This document describes version 1 of the ONC RPC program 100418 with
   the short name "fedfs_admin", a Description of "FedFS
   Administration", and a reference of [RFCTBD10]. Program 100418 will
   be removed from the reserved FedFS range and assigned these new
   values.

8. References

8.1. Normative References

   [3530bis]   Haynes, T. and D. Noveck, "NFS Version 4 Protocol",
               draft-ietf-nfsv4-rfc3530bis (Work In Progress), 2010.

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC2203]   Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol
               Specification", RFC 2203, September 1997.

   [RFC4122]   Leach, P., Mealling, M., and R. Salz, "A Universally
               Unique IDentifier (UUID) URN Namespace", RFC 4122,
               July 2005.

   [RFC4506]   Eisler, M., "XDR: External Data Representation Standard",
               STD 67, RFC 4506, May 2006.

   [RFC4511]   Sermersheim, J., "Lightweight Directory Access Protocol
               (LDAP): The Protocol", RFC 4511, June 2006.

   [RFC4513]   Harrison, R., "Lightweight Directory Access Protocol
               (LDAP): Authentication Methods and Security Mechanisms",
               RFC 4513, June 2006.

   [RFC5246]   Dierks, T. and E. Rescorla, "The Transport Layer Security
               (TLS) Protocol Version 1.2", RFC 5246, August 2008.

   [RFC5280]   Cooper, D., Santesson, S., Farrell, S., Boeyen, S.,
               Housley, R., and W. Polk, "Internet X.509 Public Key
               Infrastructure Certificate and Certificate Revocation List

                (CRL) Profile", RFC 5280, May 2008.

   [RFC5531]   Thurlow, R., "RPC: Remote Procedure Call Protocol
               Specification Version 2", RFC 5531, May 2009.

8.2. Informative References

   [FEDFS-NSDB]
               Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M.
               Naik, "NSDB Protocol for Federated Filesystems",
               draft-ietf-nfsv4-federated-fs-protocol (Work In Progress),
               2010.

   [MS-CIFS]   Microsoft Corporation, "Common Internet File System (CIFS)
               Protocol Specification", MS-CIFS 2.0, November 2009.

   [MS-SMB]   Microsoft Corporation, "Server Message Block (SMB)
               Protocol Specification", MS-SMB 17.0, November 2009.

   [MS-SMB2]   Microsoft Corporation, "Server Message Block (SMB) Version
               2 Protocol Specification", MS-SMB2 19.0, November 2009.

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

   [RFC5661]   Shepler, S., Eisler, M., and D. Noveck, "Network File
               System (NFS) Version 4 Minor Version 1 Protocol",
               RFC 5661, January 2010.

   [RFC5662]   Shepler, S., Eisler, M., and D. Noveck, "Network File
               System (NFS) Version 4 Minor Version 1 External Data
               Representation Standard (XDR) Description", RFC 5662,
               January 2010.

   [RFC5716]   Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M.
               Naik, "Requirements for Federated File Systems", RFC 5716,
               January 2010.

Appendix A. Acknowledgments

   The authors and editor would like to thank Craig Everhart and Manoj
   Naik, who were co-authors of an earlier version of this document. In
   addition, we would like to thank Paul Lemahieu, Mario Wurzl, and
   Robert Thurlow for helping to author this document.

   We would like to thank Trond Myklebust for suggesting improvements to
   the FSL pathname format, David Noveck for his suggestions on

internationalization and path encoding rules, and Nicolas Williams for his suggestions.

The editor gratefully acknowledges the IESG reviewers, whose constructive comments helped make this a much stronger document.

Finally, we would like to thank Andy Adamson, Rob Thurlow, and Tom Haynes for helping to get this document out the door.

The extract.sh shell script and formatting conventions were first described by the authors of the NFSv4.1 XDR specification [RFC5662].

Appendix B. RFC Editor Notes

[RFC Editor: please remove this section prior to publishing this document as an RFC]

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCTBD10 with RFCxxxx where xxxx is the RFC number of this document]

Authors' Addresses

James Lentini
NetApp
1601 Trapelo Rd, Suite 16
Waltham, MA 02451
US

Phone: +1 781-768-5359
Email: jlentini@netapp.com

Daniel Ellard
Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
US

Phone: +1 617-873-8004
Email: dellard@bbn.com

      Renu Tewari
      IBM Almaden
      650 Harry Rd
      San Jose, CA 95120
      US

      Email: tewarir@us.ibm.com

      Charles Lever (editor)
      Oracle Corporation
      1015 Granger Avenue
      Ann Arbor, MI 48104
      US

      Phone: +1 248-614-5091
      Email: chuck.lever@oracle.com

NFSv4 Working Group   J. Lentini
Internet-Draft   NetApp
Intended status: Standards Track   D. Ellard
Expires: June 15, 2013   Raytheon BBN Technologies
                                                             R. Tewari
                                                           IBM Almaden
                                                        C. Lever, Ed.
                                                   Oracle Corporation
                                                    December 12, 2012

                    NSDB Protocol for Federated Filesystems
                    draft-ietf-nfsv4-federated-fs-protocol-15

Abstract

   This document describes a filesystem federation protocol that enables
   file access and namespace traversal across collections of
   independently administered fileservers. The protocol specifies a set
   of interfaces by which fileservers with different administrators can
   form a fileserver federation that provides a namespace composed of
   the filesystems physically hosted on and exported by the constituent
   fileservers.

Requirements Language

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

Table of Contents

1. Introduction

   A federated filesystem enables file access and namespace traversal in
   a uniform, secure and consistent manner across multiple independent
   fileservers within an enterprise or across multiple enterprises.

   This document specifies a set of protocols that allow fileservers,
   possibly from different vendors and with different administrators, to
   cooperatively form a federation containing one or more federated
   filesystems. Each federated filesystem's namespace is composed of
   the filesystems physically hosted on and exported by the federation's
   fileservers. A federation comprises a common namespace across all
   its fileservers. A federation can project multiple namespaces and
   enable clients to traverse each one. A federation can contain an
   arbitrary number of namespace repositories, each belonging to a
   different administrative entity, and each rendering a part of the
   namespace. A federation might also have an arbitrary number of
   administrative entities responsible for administering disjoint
   subsets of the fileservers.

   Traditionally, building a namespace that spans multiple fileservers
   has been difficult for two reasons. First, the fileservers that
   export pieces of the namespace are often not in the same
   administrative domain. Second, there is no standard mechanism for
   the fileservers to cooperatively present the namespace. Fileservers
   may provide proprietary management tools and in some cases an
   administrator may be able to use the proprietary tools to build a
   shared namespace out of the exported filesystems. However, relying
   on vendor-specific proprietary tools does not work in larger
   enterprises or when collaborating across enterprises because the
   fileservers are likely to be from multiple vendors or use different
   software versions, each with their own namespace protocols, with no
   common mechanism to manage the namespace or exchange namespace
   information.

   The federated filesystem protocols in this document define how to
   construct a namespace accessible by an NFSv4.0 [3530bis], NFSv4.1
   [RFC5661] or newer client and have been designed to accommodate other
   file access protocols in the future.

   The requirements for federated filesystems are described in
   [RFC5716]. A protocol for administering a fileserver's namespace is
   described in [FEDFS-ADMIN]. The mechanism for discovering the root
   of a federated namespace is described in [RFC6641].

   In the rest of the document, the term fileserver denotes a fileserver
   that is part of a federation.

2. Overview of Features and Concepts

2.1. File-access Protocol

   A file-access protocol is a network protocol for accessing data. The
   NFSv4.0 protocol [3530bis] is an example of a file-access protocol.

2.2. File-access Client

   File-access clients are standard off-the-shelf network attached
   storage (NAS) clients that communicate with fileservers using a
   standard file-access protocol.

2.3. Fileserver

   Fileservers are servers that store physical fileset data, or refer
   file-access clients to other fileservers. A fileserver provides
   access to its shared filesystem data via a file-access protocol. A
   fileserver may be implemented in a number of different ways,
   including a single system, a cluster of systems, or some other
   configuration.

2.4. Referral

   A referral is a mechanism by which a fileserver redirects a file-
   access client to a different fileserver or export. The exact
   information contained in a referral varies from one file-access
   protocol to another. The NFSv4.0 protocol, for example, defines the
   fs_locations attribute for returning referral information to NFSv4.0
   clients. The NFSv4.1 protocol introduces the fs_locations_info
   attribute that can return richer referral information to its clients.
   NFSv4.1 fileservers may use either attribute during a referral. Both
   attributes are defined in [RFC5661].

2.5. Namespace

   The goal of a unified namespace is to make all managed data available
   to any file-access client via the same path in a common filesystem
   namespace. This should be achieved with minimal or zero
   configuration on file-access clients. In particular, updates to the
   common namespace should not require configuration changes to any
   file-access client.

   Filesets, which are the unit of data management, are a set of files
   and directories. From the perspective of file-access clients, the
   common namespace is constructed by mounting filesets that are
   physically located on different fileservers. The namespace, which is
   defined in terms of fileset names and locations, is stored in a set

   of namespace repositories, each managed by an administrative entity.

   The namespace schema defines the model used for populating,
   modifying, and querying the namespace repositories. It is not
   required by the federation that the namespace be common across all
   fileservers. It should be possible to have several independently
   rooted namespaces.

2.6. Fileset

   A fileset is loosely defined as a set of files and the directory tree
   that contains them. The fileset abstraction is the basic unit of
   data management. Depending on the configuration, a fileset may be
   anything from an individual directory of an exported filesystem to an
   entire exported filesystem on a fileserver.

2.7. Fileset Name (FSN)

   A fileset is uniquely represented by its fileset name (FSN). An FSN
   is considered unique across a federation. After an FSN is created,
   it is associated with one or more fileset locations (FSLs) on one or
   more fileservers.

   An FSN consists of:

      NsdbName: the network location of the Namespace Database (NSDB)
         node that contains authoritative information for this FSN.

      FsnUuid: a UUID (universally unique identifier), conforming to
         [RFC4122], that is used to uniquely identify an FSN.

      FsnTTL: the time-to-live of the FSN's FSL information, in
         seconds. Fileservers MUST NOT use cached FSL records after the
         parent FSN's FsnTTL has expired. An FsnTTL value of zero
         indicates that fileservers MUST NOT cache the results of
         resolving this FSN.

   The NsdbName is not physically stored as an attribute of the record.
   The NsdbName is obvious to any client that accesses an NSDB, and is
   indeed authenticated in cases where TLS security is in effect.

   The FsnUuid and NsdbName values never change during an FSN's
   lifetime. However, an FSN's FSL information can change over time,
   and is typically cached on fileservers for performance. More detail
   on FSL caching is provided in Section 2.8.3.

   An FSN record may also contain:

Annotations: name/value pairs that can be interpreted by a
      fileserver. The semantics of this field are not defined by
      this document. These tuples are intended to be used by higher-
      level protocols.

Descriptions: text descriptions. The semantics of this field are
      not defined by this document.

2.8. Fileset Location (FSL)

An FSL describes one physical location where a complete copy of the
fileset's data resides. An FSL contains generic and type specific
information which together describe how to access the fileset data at
this location. An FSL's attributes can be used by a fileserver to
decide which locations it will return to a file-access client.

An FSL consists of:

FslUuid: a UUID, conforming to [RFC4122], that is used to
      uniquely identify an FSL.

FsnUuid: the UUID of the FSL's FSN.

NsdbName: the network location of the NSDB node that contains
      authoritative information for this FSL.

The NsdbName is not stored as an attribute of an FSL record for the
same reason it is not stored in FSN records.

An FSL record may also contain:

Annotations: name/value pairs that can be interpreted by a
      fileserver. The semantics of this field are not defined by
      this document. These tuples are intended to be used by higher-
      level protocols.

Descriptions: text descriptions. The semantics of this field are
      not defined by this document.

In addition to the attributes defined above, an FSL record contains
attributes that allow a fileserver to construct referrals. For each
file-access protocol, a corresponding FSL record subtype is defined.

This document defines an FSL subtype for NFS. An NFS FSL contains
information suitable for use in one of the NFSv4 referral attributes
(e.g., fs_locations or fs_locations_info, described in [RFC5661]).
Section 4.2.2.4 describes the contents of an NFS FSL record.

A fileset also may be accessible by file-access protocols other than NFS. The contents and format of such FSL subtypes are not defined in this document.

2.8.1. The NFS URI scheme

To capture the location of an NFSv4 fileset, we extend the NFS URL scheme specified in [RFC2224]. This extension follows rules for defining Uniform Resource Identifier schemes (see [RFC3986]). In the following text, we refer to this extended NFS URL scheme as an NFS URI.

An NFS URI MUST contain both an authority and a path component. It MUST NOT contain a query component or a fragment component. Use of the familiar "nfs" scheme name is retained.

2.8.1.1. The NFS URI authority component

The rules for encoding the authority component of a generic URI are specified in section 3.2 of [RFC3986]. The authority component of an NFS URI MUST contain the host subcomponent. For globally-scoped NFS URIs, a hostname used in such URIs SHOULD be a fully qualified domain name. See section 3.2.2 of [RFC3986] for rules on encoding non-ASCII characters in hostnames.

An NFS URI MAY contain a port subcomponent as described in section 3.2.3 of [RFC3986]. If this subcomponent is missing, a port value of 2049 is assumed, as specified in [3530bis], Section 3.1.

2.8.1.2. The NFS URI path component

The rules for encoding the path component of a generic URI are specified in section 3.3 of [RFC3986].

According to sections 5 and 6 of [RFC2224], NFS URLs specify a pathname relative to an NFS fileserver's "public filehandle." However, NFSv4 fileservers do not expose a "public filehandle." Instead, NFSv4 pathnames contained in an NFS URI are evaluated relative to the pseudoroot of the fileserver identified in the URI's authority component.

Each component of an NFSv4 pathname is represented as a component4 string (see Section 3.2, "Basic Data Types" of [RFC5661]). The component4 elements of an NFSv4 pathname are encoded as path segments in an NFS URI. NFSv4 pathnames MUST be expressed in an NFS URI as an absolute path. An NFS URI path component MUST NOT be empty. The NFS URI path component starts with a slash ("/") character, followed by one or more path segments which each start with a slash ("/")

character [RFC3986].

Therefore, a double slash always follows the authority component of
an NFS URI. For example, the NFSv4 pathname "/" is represented by
two slash ("/") characters following an NFS URI's authority
component.

The component4 elements of an NFSv4 pathname MUST be prepared using
the component4 rules defined in Chapter 12 "Internationalization" of
[3530bis] prior to encoding the path component of an NFS URI. As
specified in [RFC3986], any non-ASCII characters and any URI-reserved
characters, such as the slash ("/") character, contained in a
component4 element MUST be represented by URI percent encoding.

2.8.1.3. Encoding an NFS location in an FSL

The path component of an NFS URI encodes the "rootpath" field of the
NFSv4 fs_location4 data type or the "fli_rootpath" of the NFSv4
fs_locations_item4 data type (see [RFC5661]).

In its "server" field, the NFSv4 fs_location4 data type contains a
list of universal addresses and DNS labels. Each may optionally
include a port number. The exact encoding requirements for this
information is found in Section 12.6 of [3530bis]. The NFSv4
fs_locations_item4 data type encodes the same data in its
"fli_entries" field (see [RFC5661]). This information is encoded in
the authority component of an NFS URI.

The "server" and "fli_entries" fields can encode multiple server
hostnames that share the same pathname. An NFS URI, and hence an FSL
record, represents only a single hostname and pathname pair. An NFS
fileserver MUST NOT combine a set of FSL records into a single
fs_location4 or fs_locations_item4 unless each FSL record in the set
contains the same rootpath value and extended filesystem information.

2.8.2. Mutual Consistency across Fileset Locations

All of the FSLs that have the same FSN (and thereby reference the
same fileset) are equivalent from the point of view of access by a
file-access client. Different fileset locations for an FSN represent
the same data, though potentially at different points in time.
Fileset locations are equivalent but not identical. Locations may
either be read-only or read-write. Typically, multiple read-write
locations are backed by a clustered filesystem while read-only
locations are replicas created by a federation-initiated or external
replication operation. Read-only locations may represent consistent
point-in-time copies of a read-write location. The federation
protocols, however, cannot prevent subsequent changes to a read-only

location nor guarantee point-in-time consistency of a read-only
location if the read-write location is changing.

Regardless of the type, one file-access client may be referred to a
location described by one FSL while another client chooses to use a
location described by another FSL. Since updates to each fileset
location are not controlled by the federation protocol, it is the
responsibility of administrators to guarantee the functional
equivalence of the data.

The federation protocols do not guarantee that different fileset
locations are mutually consistent in terms of the currency of their
data. However, they provide a means to publish currency information
so that all fileservers in a federation can convey the same
information to file-access clients during referrals. Clients use
this information to ensure they do not revert to an out-of-date
version of a fileset's data when switching between fileset locations.
NFSv4.1 provides guidance on how replication can be handled in such a
manner. In particular see Section 11.7 of [RFC5661].

2.8.3. Caching of Fileset Locations

To resolve an FSN to a set of FSL records, a fileserver queries the
NSDB node named in the FSN for FSL records associated with this FSN.
The parent FSN's FsnTTL attribute (see Section 2.7) specifies the
period of time during which a fileserver may cache these FSL records.

The combination of FSL caching and FSL migration presents a
challenge. For example, suppose there are three fileservers named A,
B, and C. Suppose further that fileserver A contains a junction J to
fileset X stored on fileserver B (see Section 2.10 for a description
of junctions).

Now suppose that fileset X is migrated from fileserver B to
fileserver C, and the corresponding FSL information for fileset X in
the authoritative NSDB is updated.

If fileserver A has cached FSLs for fileset X, a file-access client
traversing junction J on fileserver A will be referred to fileserver
B, even though fileset X has migrated to fileserver C. If fileserver
A had not cached the FSL records, it would have queried the NSDB and
obtained the correct location of fileset X.

Typically, the process of fileset migration leaves a redirection on
the source fileserver in place of a migrated fileset (without such a
redirection, file-access clients would find an empty space where the
migrated fileset was, which defeats the purpose of a managed
migration).

This redirection might be a new junction that targets the same FSN as other junctions referring to the migrated fileset, or it might be some other kind of directive, depending on the fileserver implementation, that simply refers file-access clients to the new location of the migrated fileset.

Back to our example. Suppose, as part of the migration process, a junction replaces fileset X on fileserver B. Later, either:

o New file-access clients are referred to fileserver B by stale FSL information cached on fileserver A, or

o File-access clients continue to access fileserver B because they cache stale location data for fileset X.

In either case, thanks to the redirection, file-access clients are informed by fileserver B that fileset X has moved to fileserver C.

Such redirecting junctions (here, on fileserver B) would not be required to be in place forever. They need to stay in place at least until FSL entries cached on fileservers and locations cached on file-access clients for the target fileset are invalidated.

The FsnTTL field in the FSL's parent FSN (see Section 2.7) specifies an upper bound for the lifetime of cached FSL information, and thus can act as a lower bound for the lifetime of redirecting junctions.

For example, suppose the FsnTTL field contains the value 3600 seconds (one hour). In such a case, administrators SHOULD keep the redirection in place for at least one hour after a fileset migration has taken place, because a referring fileserver might cache the FSL data during that time before refreshing it.

To get file-access clients to access the destination fileserver more quickly, administrators SHOULD set the FsnTTL field of the migrated fileset to a low number or zero before migration begins. It can be reset to a more reasonable number at a later point.

Note that some file-access protocols do not communicate location cache expiry information to file-access clients. In some cases it may be difficult to determine an appropriate lifetime for redirecting junctions because file-access clients may cache location information indefinitely.

2.8.4. Generating A Referral from Fileset Locations

After resolving an FSN to a set of FSL records, the fileserver generates a referral to redirect a file-access client to one or more

of the FSN's FSLs. The fileserver converts the FSL records to a
referral format understood by a particular file-access client, such
as an NFSv4 fs_locations or fs_locations_info attribute.

To give file-access clients as many options as possible, the
fileserver SHOULD include the maximum possible number of FSL records
in a referral. However, the fileserver MAY omit some of the FSL
records from the referral. For example, the fileserver might omit an
FSL record because of limitations in the file access protocol's
referral format.

For a given FSL record, the fileserver MAY convert or reduce the FSL
record's contents in a manner appropriate to the referral format.
For example, an NFS FSL record contains all the data necessary to
construct an fs_locations_info attribute, but an fs_locations_info
attribute contains several pieces of information that are not found
in the simpler fs_locations attribute. A fileserver constructs
entries in an fs_locations attribute using the relevant contents of
an NFS FSL record.

Whenever the fileserver converts or reduces FSL data, the fileserver
SHOULD attempt to maintain the original meaning where possible. For
example, an NFS FSL record contains the rank and order information
that is included in an fs_locations_info attribute (see NFSv4.1's
FSLI4BX_READRANK, FSLI4BX_READORDER, FSLI4BX_WRITERANK, and
FSLI4BX_WRITEORDER). While this rank and order information is not
explicitly expressible in an fs_locations attribute, the fileserver
can arrange the fs_locations attribute's locations list based on the
rank and order values.

Another example: A single NFS FSL record contains the hostname of one
fileserver. A single fs_locations attribute can contain a list of
fileserver names. An NFS fileserver MAY combine two or more FSL
records into a single entry in an fs_locations or fs_locations_info
array only if each FSL record contains the same pathname and extended
filesystem information.

Refer to the NFSv4.1 protocol specification [RFC5661], sections 11.9
and 11.10, for further details.

2.9. Namespace Database (NSDB)

The NSDB service is a federation-wide service that provides
interfaces to define, update, and query FSN information, FSL
information, and FSN to FSL mapping information.

An individual repository of namespace information is called an NSDB
node. The difference between the NSDB service and an NSDB node is

analogous to that between the DNS service and a particular DNS
server.

Each NSDB node is managed by a single administrative entity. A
single administrative entity can manage multiple NSDB nodes.

Each NSDB node stores the definition of the FSNs for which it is
authoritative. It also stores the definitions of the FSLs associated
with those FSNs. An NSDB node is authoritative for the filesets that
it defines.

An NSDB MAY be replicated throughout the federation. If an NSDB is
replicated, the NSDB MUST exhibit loose, converging consistency as
defined in [RFC3254]. The mechanism by which this is achieved is
outside the scope of this document. Many LDAP implementations
support replication. These features MAY be used to replicate the
NSDB.

2.9.1. NSDB Client

Each NSDB node supports an LDAP [RFC4510] interface. An NSDB client
is software that uses the LDAP protocol to access or update namespace
information stored on an NSDB node. Details of these transactions
are discussed in Section 4.

A domain's administrative entity uses NSDB client software to manage
information stored on NSDB nodes.

Fileservers act as an NSDB client when contacting a particular NSDB
node to resolve an FSN to a set of FSL records. The resulting
location information is then transferred to file-access clients via
referrals. Therefore file-access clients never have need to access
NSDBs directly.

2.10. Junctions and Referrals

A junction is a point in a particular fileset namespace where a
specific target fileset may be attached. If a file-access client
traverses the path leading from the root of a federated namespace to
the junction referring to a target fileset, it should be able to
mount and access the data in that target fileset (assuming
appropriate permissions). In other words, a junction can be viewed
as a reference from a directory in one fileset to the root of the
target fileset.

A junction can be implemented as a special marker on a directory, or
by some other mechanism in the fileserver's underlying filesystem.
What data is used by the fileserver to represent junctions is not

defined by this document. The essential property is that given a
junction, a fileserver must be able to find the FSN for the target
fileset.

When a file-access client reaches a junction, the fileserver refers
the client to a list of FSLs associated with the FSN targeted by the
junction. The client can then mount one of the associated FSLs.

The federation protocols do not limit where and how many times a
fileset is mounted in the namespace. Filesets can be nested; a
fileset can be mounted under another fileset.

2.11. Unified Namespace and the Root Fileset

The root fileset, when defined, is the top-level fileset of the
federation-wide namespace. The root of the unified namespace is the
top level directory of this fileset. A set of designated fileservers
in the federation can export the root fileset to render the
federation-wide unified namespace. When a file-access client mounts
the root fileset from any of these designated fileservers it can view
a common federation-wide namespace.

2.12. UUID Considerations

To ensure FSN and FSL records are unique across a domain, FedFS
employs UUIDs conforming to [RFC4122] to form the distinguished names
of LDAP records containing FedFS data (see Section 4.2.2.2).

Because junctions store a tuple containing an FSN UUID and the name
and port of an NSDB node, an FSN UUID must be unique only on a single
NSDB node. An FSN UUID collision can be detected immediately when an
administrator attempts to publish an FSN or FSL by storing it under a
specific NSDB Container Entry (NCE) on an authoritative NSDB host.

Note that one NSDB node may store multiple NCEs, each under a
different namingContext. If an NSDB node must contain more than one
NCE, the federation's admin entity SHOULD provide a robust method for
preventing FSN UUID collisions between FSNs that reside on the same
NSDB node but under different NCEs.

Because FSLs are children of FSNs, FSL UUIDs must be unique for just
a single FSN. As with FSNs, as soon as an FSL is published, its
uniqueness is guaranteed.

A fileserver performs the operations described in Section 5.2 as an
unauthenticated user. Thus distinguished names of FSN and FSL
records, as well as the FSN and FSL records themselves, are required
to be readable by anyone who can bind anonymously to an NSDB node.

Therefore FSN and FSL UUIDs should be considered public information.

Version 1 UUIDs contain a host's MAC address and a time stamp in the clear. This gives provenance to each UUID, but attackers can use such details to guess information about the host where the UUID was generated. Security-sensitive installations should be aware that on externally-facing NSDBs, UUIDs can reveal information about the hosts where they are generated.

In addition, version 1 UUIDs depend on the notion that a hardware MAC address is unique across machines. As virtual machines do not depend on unique physical MAC addresses and in any event an administrator can modify the physical MAC address, version 1 UUIDs are no longer considered sufficient.

To minimize the probability of UUIDs colliding, a consistent procedure for generating UUIDs should be used throughout a federation. Within a federation, UUIDs SHOULD be generated using the procedure described for version 4 of the UUID variant specified in [RFC4122].

3. Examples

In this section we provide examples and discussion of the basic operations facilitated by the federated filesystem protocol: creating a fileset, adding a replica of a fileset, resolving a junction, and creating a junction.

3.1. Creating a Fileset and its FSL(s)

A fileset is the abstraction of a set of files and the directory tree that contains them. The fileset abstraction is the fundamental unit of data management in the federation. This abstraction is implemented by an actual directory tree whose root location is specified by a fileset location (FSL).

In this section, we describe the basic requirements for starting with a directory tree and creating a fileset that can be used in the federation protocols. Note that we do not assume that the process of creating a fileset requires any transformation of the files or the directory hierarchy. The only thing that is required by this process is assigning the fileset a fileset name (FSN) and expressing the location of the implementation of the fileset as an FSL.

There are many possible variations to this procedure, depending on how the FSN that binds the FSL is created, and whether other replicas of the fileset exist, are known to the federation, and need to be

bound to the same FSN.

It is easiest to describe this in terms of how to create the initial implementation of the fileset, and then describe how to add replicas.

3.1.1. Creating a Fileset and an FSN

1. Choose the NSDB node that will keep track of the FSL(s) and related information for the fileset.

2. Create an FSN in the NSDB node.

    The FSN UUID is chosen by the administrator or generated automatically by administration software. The former case is used if the fileset is being restored, perhaps as part of disaster recovery, and the administrator wishes to specify the FSN UUID in order to permit existing junctions that reference that FSN to work again.

    At this point, the FSN exists, but its fileset locations are unspecified.

3. For the FSN created above, create an FSL in the NSDB node that describes the physical location of the fileset data.

3.1.2. Adding a Replica of a Fileset

Adding a replica is straightforward: the NSDB node and the FSN are already known. The only remaining step is to add another FSL.

Note that the federation protocols provide only the mechanisms to register and unregister replicas of a fileset. Fileserver-to-fileserver replication protocols are not defined.

3.2. Junction Resolution

A fileset may contain references to other filesets. These references are represented by junctions. If a file-access client requests access to a fileset object that is a junction, the fileserver resolves the junction to discover one or more FSLs that implement the referenced fileset.

There are many possible variations to this procedure, depending on how the junctions are represented by the fileserver and how the fileserver performs junction resolution.

Step 4 is the only step that interacts directly with the federation protocols. The rest of the steps may use platform-specific

interfaces.

1. The fileserver determines that the object being accessed is a
    junction.

2. The fileserver does a local lookup to find the FSN of the target
    fileset.

3. Using the FSN, the fileserver finds the NSDB node responsible for
    the target FSN.

4. The fileserver contacts that NSDB node and asks for the set of
    FSLs that implement the target FSN. The NSDB node responds with
    a (possibly empty) set of FSLs.

5. The fileserver converts one or more of the FSLs to the location
    type used by the file-access client (e.g., an NFSv4 fs_locations
    attribute as described in [RFC5661]).

6. The fileserver redirects (in whatever manner is appropriate for
    the client) the client to the location(s).

3.3. Example Use Cases for Fileset Annotations

Fileset annotations can convey additional attributes of a fileset.
For example, fileset annotations can be used to define relationships
between filesets that can be used by an auxiliary replication
protocol. Consider the scenario where a fileset is created and
mounted at some point in the namespace. A snapshot of the read-write
FSL of that fileset is taken periodically at different frequencies
(say, a daily or weekly snapshot). The different snapshots are
mounted at different locations in the namespace.

The daily snapshots are considered as different filesets from the
weekly ones, but both are related to the source fileset. We can
define an annotation labeling the filesets as source and replica.
The replication protocol can use this information to copy data from
one or more FSLs of the source fileset to all the FSLs of the replica
fileset. The replica filesets are read-only while the source fileset
is read-write.

This follows the traditional Andrew File System (AFS) model of
mounting the read-only volume at a path in the namespace different
from that of the read-write volume [AFS].

The federation protocol does not control or manage the relationship
among filesets. It merely enables annotating the filesets with user-
defined relationships.

Another potential use for annotations is recording references to an
FSN. A single annotation containing the number of references could
be defined; or multiple annotations, one per reference, could be used
to store detailed information on the location of each reference.

As with the replication annotation described above, the maintenance
of reference information would not be controlled by the federation
protocol. The information would most likely be non-authoritative
because the ability to create a junction does not require the
authority to update the FSN record. In any event, such annotations
could be useful to administrators for determining if an FSN is
referenced by a junction.

4. NSDB Configuration and Schema

This section describes how an NSDB is constructed using an LDAP
Version 3 [RFC4510] Directory. Section 4.1 describes the basic
properties of the LDAP configuration that MUST be used in order to
ensure compatibility between different implementations. Section 4.2
defines the new LDAP attribute types, the new object types, and
specifies how the distinguished name (DN) of each object instance
MUST be constructed.

4.1. LDAP Configuration

An NSDB is constructed using an LDAP Directory. This LDAP Directory
MAY have multiple naming contexts. The LDAP Directory's DSA-specific
entry (its rootDSE) has a multi-valued namingContext attribute. Each
value of the namingContext attribute is the DN of a naming context's
root entry (see [RFC4512]).

For each naming context that contains federation entries (e.g., FSNs
and FSLs):

1. There MUST be an LDAP entry that is superior to all of the naming
    context's federation entries in the Directory Information Tree
    (DIT). This entry is termed the NSDB Container Entry (NCE). The
    NCE's children are FSNs. An FSN's children are FSLs.

2. The naming context's root entry MUST include the
    fedfsNsdbContainerInfo (defined below) as one of its object
    classes. The fedfsNsdbContainerInfo's fedfsNceDN attribute is
    used to locate the naming context's NCE.

If a naming context does not contain federation entries, it will not
contain an NCE and its root entry will not include a
fedfsNsdbContainerInfo as one of its object classes.

A fedfsNsdbContainerInfo's fedfsNceDN attribute contains the
Distinguished Name (DN) of the NSDB Container Entry residing under
this naming context. The fedfsNceDN attribute MUST NOT be empty.

For example, an LDAP directory might have the following entries:

```
        -+ [root DSE]
         |  namingContext: o=fedfs
         |  namingContext: dc=example,dc=com
         |  namingContext: ou=system
         |
         |
         |
        +---- [o=fedfs]
         |        fedfsNceDN: o=fedfs
         |
         |
         |
        +---- [dc=example,dc=com]
         |        fedfsNceDN: ou=fedfs,ou=corp-it,dc=example,dc=com
         |
         |
         |
        +---- [ou=system]
```

In this case, the o=fedfs namingContext has an NSDB Container Entry
at o=fedfs, the dc=example,dc=com namingContext has an NSDB Container
Entry at ou=fedfs,ou=corp-it,dc=example,dc=com, and the ou=system
namingContext has no NSDB Container Entry.

The NSDB SHOULD be configured with one or more privileged LDAP users.
These users are able to modify the contents of the LDAP database. An
administrator that performs the operations described in Section 5.1
SHOULD authenticate using the DN of a privileged LDAP user.

It MUST be possible for an unprivileged (unauthenticated) user to
perform LDAP queries that access the NSDB data. A fileserver
performs the operations described in Section 5.2 as an unprivileged
user.

All implementations SHOULD use the same schema. At minimum, each
MUST use a schema that includes all objects named in the following
sections, with all associated attributes. If it is necessary for an
implementation to extend the schema defined here, consider using one
of the following ways to extend the schema:

o Define a fedfsAnnotation key and values (see Section 4.2.1.6).
   Register the new key and values with IANA (see Section 7.1).

o Define additional attribute types and object classes, then have
   entries inherit from a class defined in this document and from the

implementation-defined ones.

Given the above configuration guidelines, an NSDB SHOULD be
constructed using a dedicated LDAP server. If LDAP directories are
needed for other purposes, such as to store user account information,
use of a separate LDAP server for those is RECOMMENDED. By using an
LDAP server dedicated to storing NSDB records, there is no need to
disturb the configuration of any other LDAP directories that store
information unrelated to an NSDB.

4.2. LDAP Schema

The schema definitions provided in this document use the LDAP schema
syntax defined in [RFC4512]. The definitions are formatted to allow
the reader to easily extract them from the document. The reader can
use the following shell script to extract the definitions:

```
<CODE BEGINS>

#!/bin/sh
grep '^ *///' | sed 's?^ */// ??' | sed 's?^ *///$??'

<CODE ENDS>
```

If the above script is stored in a file called "extract.sh", and this
document is in a file called "spec.txt", then the reader can do:

```
<CODE BEGINS>

sh extract.sh < spec.txt > fedfs.schema

<CODE ENDS>
```

The effect of the script is to remove leading white space from each
line, plus a sentinel sequence of "///".

Code components extracted from this document must include the
following license:

<CODE BEGINS>

```
<CODE ENDS>
```

4.2.1. LDAP Attributes

   The following definitions are used below:

   o The "name" attribute described in [RFC4519].

   o The Integer syntax (1.3.6.1.4.1.1466.115.121.1.27) described in
      [RFC4517].

   o The "integerMatch" rule described in [RFC4517].

   o The Octet String syntax (1.3.6.1.4.1.1466.115.121.1.40) described
      in [RFC4517].

   o The "octetStringMatch" rule described in [RFC4517].

   o The Boolean syntax (1.3.6.1.4.1.1466.115.121.1.7) described in
      [RFC4517].

   o The "booleanMatch" rule described in [RFC4517].

   o The "distinguishedNameMatch" rule described in [RFC4517].

   o The DN syntax (1.3.6.1.4.1.1466.115.121.1.12) described in
      [RFC4517].

   o The "labeledURI" attribute described in [RFC2079].

   o The UUID syntax (1.3.6.1.1.16.1) described in [RFC4530].

   o The UuidMatch rule described in [RFC4530].

   o The UuidOrderingMatch rule described in [RFC4530].

4.2.1.1. fedfsUuid

   A fedfsUuid is the base type for all of the universally unique
   identifiers (UUIDs) used by the federated filesystem protocols.

   The fedfsUuid type is based on rules and syntax defined in [RFC4530].

   A fedfsUuid is a single-valued LDAP attribute.

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.1 NAME 'fedfsUuid'
        ///     DESC 'A UUID used by NSDB'
        ///     EQUALITY uuidMatch
        ///     ORDERING uuidOrderingMatch
        ///     SYNTAX 1.3.6.1.1.16.1
        ///     SINGLE-VALUE
        ///     )
        ///
```

    <CODE ENDS>

4.2.1.2. fedfsFsnUuid

   A fedfsFsnUuid represents the UUID component of an FSN. An NSDB
   SHOULD ensure that no two FSNs it stores have the same fedfsFsnUuid.

   This attribute is single-valued.

    <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.4 NAME 'fedfsFsnUuid'
        ///     DESC 'The FSN UUID component of an FSN'
        ///     SUP fedfsUuid
        ///     SINGLE-VALUE
        ///     )
        ///
```

    <CODE ENDS>

4.2.1.3. fedfsFsnTTL

   A fedfsFsnTTL is the time-to-live in seconds of a cached FSN and its
   child FSL records. It corresponds to the FsnTTL as defined in
   Section 2.7. See also Section Section 2.8.3 for information about
   caching FSLs. A fedfsFsnTTL MUST be encoded as an Integer syntax
   value [RFC4517] in the range [0, 4294967295].

   This attribute is single-valued.

    <CODE BEGINS>

```
        ///
        /// attributetype (
        ///      1.3.6.1.4.1.31103.1.11 NAME 'fedfsFsnTTL'
        ///      DESC 'Time to live of an FSN tree'
        ///      EQUALITY integerMatch
        ///      SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///      SINGLE-VALUE
        ///      )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.4. fedfsNceDN

   A fedfsNceDN stores a distinguished name (DN).

   This attribute is single-valued.

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///      1.3.6.1.4.1.31103.1.14 NAME 'fedfsNceDN'
        ///      DESC 'NCE Distinguished Name'
        ///      EQUALITY distinguishedNameMatch
        ///      SYNTAX 1.3.6.1.4.1.1466.115.121.1.12
        ///      SINGLE-VALUE
        ///      )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.12 is the DN syntax [RFC4517].

4.2.1.5. fedfsFslUuid

   A fedfsFslUuid represents the UUID of an FSL. An NSDB SHOULD ensure
   that no two FSLs it stores have the same fedfsFslUuid.

   This attribute is single-valued.

   <CODE BEGINS>

```
///
/// attributetype (
///      1.3.6.1.4.1.31103.1.8 NAME 'fedfsFslUuid'
///      DESC 'UUID of an FSL'
///      SUP fedfsUuid
///      SINGLE-VALUE
///      )
///
```

<CODE ENDS>

4.2.1.6. fedfsAnnotation

   A fedfsAnnotation contains an object annotation formatted as a key/
   value pair.

   This attribute is multi-valued; an object type that permits
   annotations may have any number of annotations per instance.

   A fedfsAnnotation attribute is a human-readable sequence of UTF-8
   characters with no non-terminal NUL characters. The value MUST be
   formatted according to the following ABNF [RFC5234] rules:

```
ANNOTATION = KEY "=" VALUE
KEY = ITEM
VALUE = ITEM
ITEM = *WSP DQUOTE UTF8-octets DQUOTE *WSP
```

   DQUOTE and WSP are defined in [RFC5234], and UTF8-octets is defined
   in [RFC3629].

   The following escape sequences are allowed:

```
      +-----------------+-------------+
      | escape sequence | replacement |
      +-----------------+-------------+
      |       \\        |      \      |
      |       \"        |      "      |
      +-----------------+-------------+
```

   A fedfsAnnotation value might be processed as follows:

   1. Parse the attribute value according to the ANNOTATION rule,
       ignoring the escape sequences above.

   2. Scan through results of the previous step and replace the escape
       sequences above.

A fedfsAnnotation attribute that does not adhere to this format SHOULD be ignored in its entirety. It MUST NOT prevent further processing of its containing entry.

The following are examples of valid fedfsAnnotation attributes:

```
"key1" = "foo"
"another key" = "x=3"
"key-2" = "A string with \" and \\ characters."
"key3"="bar"
```

which correspond to the following key/value pairs:

```
+-------------+--------------------------------+
|     key     |              value             |
+-------------+--------------------------------+
|     key1    |               foo              |
| another key |               x=3              |
|    key-2    | A string with " and \ characters. |
|     key3    |               bar              |
+-------------+--------------------------------+
```

<CODE BEGINS>

```
///
/// attributetype (
///      1.3.6.1.4.1.31103.1.12 NAME 'fedfsAnnotation'
///      DESC 'Annotation of an object'
///      SUP name
///      )
///
```

<CODE ENDS>

4.2.1.7. fedfsDescr

A fedfsDescr stores an object description. The description MUST be encoded as a UTF-8 string.

This attribute is multi-valued which permits any number of descriptions per entry.

<CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.13 NAME 'fedfsDescr'
        ///     DESC 'Description of an object'
        ///     SUP name
        ///     )
        ///
```

    <CODE ENDS>

4.2.1.8. fedfsNfsURI

    A fedfsNfsURI stores the host and pathname components of an FSL. A
    fedfsNfsURI MUST be encoded as an NFS URI (see Section 2.8.1).

    The fedfsNfsURI is a subtype of the labeledURI type [RFC2079], with
    the same encoding rules.

    This attribute is single-valued.

    <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.120 NAME 'fedfsNfsURI'
        ///     DESC 'Location of fileset'
        ///     SUP labeledURI
        ///     SINGLE-VALUE
        ///     )
        ///
```

    <CODE ENDS>

4.2.1.9. fedfsNfsCurrency

    A fedfsNfsCurrency stores the NFSv4.1 fs_locations_server's
    fls_currency value [RFC5661]. A fedfsNfsCurrency MUST be encoded as
    an Integer syntax value [RFC4517] in the range [-2147483648,
    2147483647].

    This attribute is single-valued.

    <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.103 NAME 'fedfsNfsCurrency'
        ///     DESC 'up-to-date measure of the data'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.10. fedfsNfsGenFlagWritable

   A fedfsNfsGenFlagWritable stores the value of an FSL's NFSv4.1
   FSLI4GF_WRITABLE bit [RFC5661]. A value of "TRUE" indicates the bit
   is set. A value of "FALSE" indicates the bit is not set.

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.104 NAME 'fedfsNfsGenFlagWritable'
        ///     DESC 'Indicates if the filesystem is writable'
        ///     EQUALITY booleanMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.7 is the Boolean syntax [RFC4517].

4.2.1.11. fedfsNfsGenFlagGoing

   A fedfsNfsGenFlagGoing stores the value of an FSL's NFSv4.1
   FSLI4GF_GOING bit [RFC5661]. A value of "TRUE" indicates the bit is
   set. A value of "FALSE" indicates the bit is not set.

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.105 NAME 'fedfsNfsGenFlagGoing'
        ///     DESC 'Indicates if the filesystem is going'
        ///     EQUALITY booleanMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.7 is the Boolean syntax [RFC4517].

4.2.1.12. fedfsNfsGenFlagSplit

   A fedfsNfsGenFlagSplit stores the value of an FSL's NFSv4.1
   FSLI4GF_SPLIT bit [RFC5661]. A value of "TRUE" indicates the bit is
   set. A value of "FALSE" indicates the bit is not set.

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.106 NAME 'fedfsNfsGenFlagSplit'
        ///     DESC 'Indicates if there are multiple filesystems'
        ///     EQUALITY booleanMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.7 is the Boolean syntax [RFC4517].

4.2.1.13. fedfsNfsTransFlagRdma

   A fedfsNfsTransFlagRdma stores the value of an FSL's NFSv4.1
   FSLI4TF_RDMA bit [RFC5661]. A value of "TRUE" indicates the bit is
   set. A value of "FALSE" indicates the bit is not set.

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.107 NAME 'fedfsNfsTransFlagRdma'
        ///     DESC 'Indicates if the transport supports RDMA'
        ///     EQUALITY booleanMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
        ///     SINGLE-VALUE
        ///     )
        ///
```

    <CODE ENDS>

    OID 1.3.6.1.4.1.1466.115.121.1.7 is the Boolean syntax [RFC4517].

4.2.1.14. fedfsNfsClassSimul

    A fedfsNfsClassSimul contains the FSL's NFSv4.1 FSLI4BX_CLSIMUL
    [RFC5661] value. A fedfsNfsClassSimul MUST be encoded as an Integer
    syntax value [RFC4517] in the range [0, 255].

    <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.108 NAME 'fedfsNfsClassSimul'
        ///     DESC 'The simultaneous-use class of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

    <CODE ENDS>

    OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.15. fedfsNfsClassHandle

    A fedfsNfsClassHandle contains the FSL's NFSv4.1 FSLI4BX_CLHANDLE
    [RFC5661] value. A fedfsNfsClassHandle MUST be encoded as an Integer
    syntax value [RFC4517] in the range [0, 255].

    <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.109 NAME 'fedfsNfsClassHandle'
        ///     DESC 'The handle class of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.16. fedfsNfsClassFileid

   A fedfsNfsClassFileid contains the FSL's NFSv4.1 FSLI4BX_CLFILEID
   [RFC5661] value. A fedfsNfsClassFileid MUST be encoded as an Integer
   syntax value [RFC4517] in the range [0, 255].

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.110 NAME 'fedfsNfsClassFileid'
        ///     DESC 'The fileid class of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.17. fedfsNfsClassWritever

   A fedfsNfsClassWritever contains the FSL's NFSv4.1 FSLI4BX_CLWRITEVER
   [RFC5661] value. A fedfsNfsClassWritever MUST be encoded as an
   Integer syntax value [RFC4517] in the range [0, 255].

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.111 NAME 'fedfsNfsClassWritever'
        ///     DESC 'The write-verifier class of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.18. fedfsNfsClassChange

   A fedfsNfsClassChange contains the FSL's NFSv4.1 FSLI4BX_CLCHANGE
   [RFC5661] value. A fedfsNfsClassChange MUST be encoded as an Integer
   syntax value [RFC4517] in the range [0, 255].

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.112 NAME 'fedfsNfsClassChange'
        ///     DESC 'The change class of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.19. fedfsNfsClassReaddir

   A fedfsNfsClassReaddir contains the FSL's NFSv4.1 FSLI4BX_CLREADDIR
   [RFC5661] value. A fedfsNfsClassReaddir MUST be encoded as an
   Integer syntax value [RFC4517] in the range [0, 255].

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.113 NAME 'fedfsNfsClassReaddir'
        ///     DESC 'The readdir class of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

<CODE ENDS>

OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.20. fedfsNfsReadRank

A fedfsNfsReadRank contains the FSL's NFSv4.1 FSLI4BX_READRANK
[RFC5661] value. A fedfsNfsReadRank MUST be encoded as an Integer
syntax value [RFC4517] in the range [0, 255].

<CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.114 NAME 'fedfsNfsReadRank'
        ///     DESC 'The read rank of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

<CODE ENDS>

OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.21. fedfsNfsReadOrder

A fedfsNfsReadOrder contains the FSL's NFSv4.1 FSLI4BX_READORDER
[RFC5661] value. A fedfsNfsReadOrder MUST be encoded as an Integer
syntax value [RFC4517] in the range [0, 255].

<CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.115 NAME 'fedfsNfsReadOrder'
        ///     DESC 'The read order of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.22. fedfsNfsWriteRank

   A fedfsNfsWriteRank contains the FSL's FSLI4BX_WRITERANK [RFC5661]
   value. A fedfsNfsWriteRank MUST be encoded as an Integer syntax
   value [RFC4517] in the range [0, 255].

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.116 NAME 'fedfsNfsWriteRank'
        ///     DESC 'The write rank of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.23. fedfsNfsWriteOrder

   A fedfsNfsWriteOrder contains the FSL's FSLI4BX_WRITEORDER [RFC5661]
   value. A fedfsNfsWriteOrder MUST be encoded as an Integer syntax
   value [RFC4517] in the range [0, 255].

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.117 NAME 'fedfsNfsWriteOrder'
        ///     DESC 'The write order of the filesystem'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

4.2.1.24. fedfsNfsVarSub

   A fedfsNfsVarSub stores the value of an FSL's NFSv4.1 FSLI4IF_VAR_SUB
   bit [RFC5661]. A value of "TRUE" indicates the bit is set. A value
   of "FALSE" indicates the bit is not set.

   <CODE BEGINS>

```
        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.118 NAME 'fedfsNfsVarSub'
        ///     DESC 'Indicates if variable substitution is present'
        ///     EQUALITY booleanMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
        ///     SINGLE-VALUE
        ///     )
        ///
```

   <CODE ENDS>

   OID 1.3.6.1.4.1.1466.115.121.1.7 is the Boolean syntax [RFC4517].

4.2.1.25. fedfsNfsValidFor

   A fedfsNfsValidFor stores an FSL's NFSv4.1 fs_locations_info
   fli_valid_for value [RFC5661]. A fedfsNfsValidFor MUST be encoded as
   an Integer syntax value [RFC4517] in the range [-2147483648,
   2147483647].

   An FSL's parent's fedfsFsnTTL value and its fedfsNfsValidFor value
   MAY be different.

   This attribute is single-valued.

```
<CODE BEGINS>

        ///
        /// attributetype (
        ///     1.3.6.1.4.1.31103.1.19 NAME 'fedfsNfsValidFor'
        ///     DESC 'Valid for time'
        ///     EQUALITY integerMatch
        ///     SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        ///     SINGLE-VALUE
        ///     )
        ///
```

   OID 1.3.6.1.4.1.1466.115.121.1.27 is the Integer syntax [RFC4517].

   <CODE ENDS>

4.2.2. LDAP Object Classes

4.2.2.1. fedfsNsdbContainerInfo

   A fedfsNsdbContainerInfo describes the location of the NCE.

   A fedfsNsdbContainerInfo's fedfsNceDN attribute is REQUIRED.

   A fedfsNsdbContainerInfo's fedfsAnnotation and fedfsDescr attributes
   are OPTIONAL.

```
<CODE BEGINS>

        ///
        /// objectclass (
        ///     1.3.6.1.4.1.31103.1.1001 NAME 'fedfsNsdbContainerInfo'
        ///     DESC 'Describes NCE location'
        ///     SUP top AUXILIARY
        ///     MUST (
        ///             fedfsNceDN
        ///     )
        ///     MAY (
        ///             fedfsAnnotation
        ///             $ fedfsDescr
        ///     ))
        ///
```

   <CODE ENDS>

4.2.2.2. fedfsFsn

   A fedfsFsn represents an FSN.

   A fedfsFsn's fedfsFsnUuid and fedfsFsnTTL attributes are REQUIRED.

   A fedfsFsn's fedfsAnnotation and fedfsDescr attributes are OPTIONAL.

   The DN of an FSN is REQUIRED to take the following form:
   "fedfsFsnUuid=$FSNUUID,$NCE", where $FSNUUID is the UUID of the FSN
   and $NCE is the DN of the NCE. Since LDAP requires a DN to be
   unique, this ensures that each FSN entry has a unique UUID value
   within the LDAP directory.

   <CODE BEGINS>

```
        ///
        /// objectclass (
        ///     1.3.6.1.4.1.31103.1.1002 NAME 'fedfsFsn'
        ///     DESC 'Represents a fileset'
        ///     SUP top STRUCTURAL
        ///     MUST (
        ///             fedfsFsnUuid
        ///             $ fedfsFsnTTL
        ///     )
        ///     MAY (
        ///             fedfsAnnotation
        ///             $ fedfsDescr
        ///     ))
        ///
```

   <CODE ENDS>

4.2.2.3. fedfsFsl

   The fedfsFsl object class represents an FSL.

   The fedfsFsl is an abstract object class. Protocol specific subtypes
   of this object class are used to store FSL information. The
   fedfsNfsFsl object class defined below is used to record an NFS FSL's
   location. Other subtypes MAY be defined for other protocols (e.g.,
   CIFS).

   A fedfsFsl's fedfsFslUuid and fedfsFsnUuid attributes are REQUIRED.

   A fedfsFsl's fedfsAnnotation, and fedfsDescr attributes are OPTIONAL.

   The DN of an FSL is REQUIRED to take the following form:

"fedfsFslUuid=$FSLUUID,fedfsFsnUuid=$FSNUUID,$NCE" where $FSLUUID is
the FSL's UUID, $FSNUUID is the FSN's UUID, and $NCE is the DN of the
NCE. Since LDAP requires a DN to be unique, this ensures that each
FSL entry has a unique UUID value within the LDAP directory.

<CODE BEGINS>

```
        ///
        /// objectclass (
        ///     1.3.6.1.4.1.31103.1.1003 NAME 'fedfsFsl'
        ///     DESC 'A physical location of a fileset'
        ///     SUP top ABSTRACT
        ///     MUST (
        ///             fedfsFslUuid
        ///             $ fedfsFsnUuid
        ///     )
        ///     MAY (
        ///             fedfsAnnotation
        ///             $ fedfsDescr
        ///     ))
        ///
```

<CODE ENDS>

4.2.2.4. fedfsNfsFsl

A fedfsNfsFsl is used to represent an NFS FSL. The fedfsNfsFsl
inherits all of the attributes of the fedfsFsl and extends the
fedfsFsl with information specific to the NFS protocol.

The DN of an NFS FSL is REQUIRED to take the following form:
"fedfsFslUuid=$FSLUUID,fedfsFsnUuid=$FSNUUID,$NCE" where $FSLUUID is
the FSL's UUID, $FSNUUID is the FSN's UUID, and $NCE is the DN of the
NCE. Since LDAP requires a DN to be unique, this ensures that each
NFS FSL entry has a unique UUID value within the LDAP directory.

<CODE BEGINS>

```
        ///
        /// objectclass (
        ///     1.3.6.1.4.1.31103.1.1004 NAME 'fedfsNfsFsl'
        ///     DESC 'An NFS location of a fileset'
        ///     SUP fedfsFsl STRUCTURAL
        ///     MUST (
        ///             fedfsNfsURI
        ///           $ fedfsNfsCurrency
        ///           $ fedfsNfsGenFlagWritable
        ///           $ fedfsNfsGenFlagGoing
        ///           $ fedfsNfsGenFlagSplit
        ///           $ fedfsNfsTransFlagRdma
        ///           $ fedfsNfsClassSimul
        ///           $ fedfsNfsClassHandle
        ///           $ fedfsNfsClassFileid
        ///           $ fedfsNfsClassWritever
        ///           $ fedfsNfsClassChange
        ///           $ fedfsNfsClassReaddir
        ///           $ fedfsNfsReadRank
        ///           $ fedfsNfsReadOrder
        ///           $ fedfsNfsWriteRank
        ///           $ fedfsNfsWriteOrder
        ///           $ fedfsNfsVarSub
        ///           $ fedfsNfsValidFor
        ///     ))
        ///
```

   <CODE ENDS>

5. NSDB Operations

   The operations defined by the protocol can be described as several
   sub-protocols that are used by entities within a federation to
   perform different roles.

   The first of these sub-protocols defines how the state of an NSDB
   node can be initialized and updated. The primary use of this sub-
   protocol is by an administrator to add, edit, or delete filesets,
   their properties, and their fileset locations.

   The second of these sub-protocols defines the queries that are sent
   to an NSDB node in order to perform resolution (or to find other
   information about the data stored within that NSDB node) and the
   responses returned by the NSDB node. The primary use of this sub-
   protocol is by a fileserver in order to perform resolution, but it
   may also be used by an administrator to query the state of the
   system.

The first and second sub-protocols are defined as LDAP operations, using the schema defined in the previous section. If each NSDB node is a standard LDAP server, then, in theory, it is unnecessary to describe the LDAP operations in detail, because the operations are ordinary LDAP operations to query and update records. However, we do not require that an NSDB node implement a complete LDAP service, and therefore we define in these sections the minimum level of LDAP functionality required to implement an NSDB node.

The NSDB sub-protocols are defined in Section 5.1 and Section 5.2. The descriptions of LDAP messages in these sections use the LDAP Data Interchange Format (LDIF) [RFC2849]. In order to differentiate constant and variable strings in the LDIF specifications, variables are prefixed by a $ character and use all upper case characters. For example, a variable named FOO would be specified as $FOO.

This document uses the term NSDB client to refer to an LDAP client that uses either of the NSDB sub-protocols.

The third sub-protocol defines the queries and other requests that are sent to a fileserver in order to get information from it or to modify the state of the fileserver in a manner related to the federation protocols. The primary purpose of this protocol is for an administrator to create or delete a junction or discover related information about a particular fileserver.

The third sub-protocol is defined as an ONC RPC protocol. The reason for using ONC RPC instead of LDAP is that all fileservers support ONC RPC but some do not support an LDAP Directory server.

The ONC RPC administration protocol is defined in [FEDFS-ADMIN].

5.1. NSDB Operations for Administrators

The admin entity initiates and controls the commands to manage fileset and namespace information. The protocol used for communicating between the admin entity and each NSDB node MUST be the LDAPv3 [RFC4510] protocol.

The names we assign to these operations are entirely for the purpose of exposition in this document, and are not part of the LDAP dialogs.

5.1.1. Create an FSN

This operation creates a new FSN in the NSDB by adding a new fedfsFsn entry in the NSDB's LDAP directory.

A fedfsFsn entry contains a fedfsFsnUuid. The administrator chooses

the fedfsFsnUuid by a process described in Section 2.12). A fedfsFsn
entry also contains a fedfsFsnTTL. The fedfsFsnTTL is chosen by the
administrator as described in Section 2.8.3.

5.1.1.1. LDAP Request

This operation is implemented using the LDAP ADD request described by
the LDIF below.

```
dn: fedfsFsnUuid=$FSNUUID,$NCE
changeType: add
objectClass: fedfsFsn
fedfsFsnUuid: $FSNUUID
fedfsFsnTTL: $TTL
```

For example, if the $FSNUUID is "e8c4761c-eb3b-4307-86fc-
f702da197966", the $TTL is "300" seconds, and the $NCE is "o=fedfs"
the operation would be:

```
dn: fedfsFsnUuid=e8c4761c-eb3b-4307-86fc-f702da197966,o=fedfs
changeType: add
objectClass: fedfsFsn
fedfsFsnUuid: e8c4761c-eb3b-4307-86fc-f702da197966
fedfsFsnTTL: 300
```

5.1.2. Delete an FSN

This operation deletes an FSN by removing a fedfsFsn entry in the
NSDB's LDAP directory.

If the FSN entry being deleted has child FSL entries, this function
MUST return an error. This ensures that the NSDB will not contain
any orphaned FSL entries. A compliant LDAP implementation will meet
this requirement since Section 4.8 of [RFC4511] defines the LDAP
delete operation to only be capable of removing leaf entries.

Note that the FSN delete function removes the fileset only from a
federation namespace (by removing the records for that FSN from the
NSDB node that receives this request). The fileset and its data are
not deleted. Any junction that has this FSN as its target may
continue to point to this non-existent FSN. A dangling reference may
be detected when a fileserver tries to resolve a junction that refers
to the deleted FSN.

5.1.2.1. LDAP Request

This operation is implemented using the LDAP DELETE request described
by the LDIF below.

```
dn: fedfsFsnUuid=$FSNUUID,$NCE
changeType: delete
```

For example, if the $FSNUUID is "e8c4761c-eb3b-4307-86fc-
f702da197966" and $NCE is "o=fedfs", the operation would be:

```
dn: fedfsFsnUuid=e8c4761c-eb3b-4307-86fc-f702da197966,o=fedfs
changeType: delete
```

5.1.3. Create an FSL

This operation creates a new FSL for the given FSN by adding a new
fedfsFsl entry in the NSDB's LDAP directory.

A fedfsFsl entry contains a fedfsFslUuid and fedfsFsnUuid. The
administrator chooses the fedfsFslUuid. The process for choosing the
fedfsFslUuid is described in Section 2.12. The fedfsFsnUuid is the
UUID of the FSL's FSN.

The administrator will also set additional attributes depending on
the FSL type.

5.1.3.1. LDAP Request

This operation is implemented using the LDAP ADD request described by
the LDIF below (NOTE: the LDIF shows the creation of an NFS FSL)

```
dn: fedfsFslUuid=$FSLUUID,fedfsFsnUuid=$FSNUUID,$NCE
changeType: add
objectClass: fedfsNfsFsl
fedfsFslUuid: $FSLUUID
fedfsFsnUuid: $FSNUUID
fedfsNfsURI: nfs://$HOST:$PORT//$PATH
fedfsNfsCurrency: $CURRENCY
fedfsNfsGenFlagWritable: $WRITABLE
fedfsNfsGenFlagGoing: $GOING
fedfsNfsGenFlagSplit: $SPLIT
fedfsNfsTransFlagRdma: $RDMA
fedfsNfsClassSimul: $CLASS_SIMUL
fedfsNfsClassHandle:$CLASS_HANDLE
fedfsNfsClassFileid:$CLASS_FILEID
fedfsNfsClassWritever:$CLASS_WRITEVER
fedfsNfsClassChange: $CLASS_CHANGE
fedfsNfsClassReaddir: $CLASS_READDIR
fedfsNfsReadRank: $READ_RANK
fedfsNfsReadOrder: $READ_ORDER
fedfsNfsWriteRank: $WRITE_RANK
fedfsNfsWriteOrder: $WRITE_ORDER
fedfsNfsVarSub: $VAR_SUB
fedfsNfsValidFor: $TIME
fedfsAnnotation: $ANNOTATION
fedfsDescr: $DESCR
```

For example, if the $FSNUUID is "e8c4761c-eb3b-4307-86fc-
f702da197966", the $FSLUUID is "ba89a802-41a9-44cf-8447-
dda367590eb3", the $HOST is "server.example.com", $PORT is "20049",
the $PATH is stored in the file "/tmp/fsl_path", $CURRENCY is "0" (an
up-to-date copy), the FSL is writable, but not going, split, or
accessible via RDMA, the simultaneous-use class is "1", the handle
class is "0", the fileid class is "1", the write-verifier class is
"1", the change class is "1", the readdir class is "9", the read rank
is "7", the read order is "8", the write rank is "5", the write order
is "6", variable substitution is false, $TIME is "300" seconds,
$ANNOTATION is ""foo" = "bar"", $DESC is "This is a description.",
and the $NCE is "o=fedfs", the operation would be (for readability
the DN is split into two lines):

```
dn: fedfsFslUuid=ba89a802-41a9-44cf-8447-dda367590eb3,
 fedfsFsnUuid=e8c4761c-eb3b-4307-86fc-f702da197966,o=fedfs
changeType: add
objectClass: fedfsNfsFsl
fedfsFslUuid: ba89a802-41a9-44cf-8447-dda367590eb3
fedfsFsnUuid: e8c4761c-eb3b-4307-86fc-f702da197966
fedfsNfsURI: nfs://server.example.com:20049//tmp/fsl_path
fedfsNfsCurrency: 0
fedfsNfsGenFlagWritable: TRUE
fedfsNfsGenFlagGoing: FALSE
fedfsNfsGenFlagSplit: FALSE
fedfsNfsTransFlagRdma: FALSE
fedfsNfsClassSimul: 1
fedfsNfsClassHandle: 0
fedfsNfsClassFileid: 1
fedfsNfsClassWritever: 1
fedfsNfsClassChange: 1
fedfsNfsClassReaddir: 9
fedfsNfsReadRank: 7
fedfsNfsReadOrder: 8
fedfsNfsWriteRank: 5
fedfsNfsWriteOrder: 6
fedfsNfsVarSub: FALSE
fedfsNfsValidFor: 300
fedfsAnnotation: "foo" = "bar"
fedfsDescr: This is a description.
```

5.1.3.2. Selecting fedfsNfsFsl Values

   The fedfsNfsFSl object class is used to describe NFSv4 accessible
   filesets. For the reasons described in Section 2.8.4, administrators
   SHOULD choose reasonable values for all LDAP attributes of an NFSv4
   accessible fedfsNfsFsl even though some of these LDAP attributes are
   not explicitly contained in an NFSv4 fs_locations attribute.

   When the administrator is unable to choose reasonable values for the
   LDAP attributes not explicitly contained in an NFSv4 fs_locations
   attribute, the values in the following table are RECOMMENDED.

+------------------------+----------+------------------------------+
| LDAP attribute | LDAP | Notes |
| | value | |
+------------------------+----------+------------------------------+
fedfsNfsCurrency	negative	Indicates that the server
	value	does not know the currency
		(see 11.10.1 of [RFC5661]).
fedfsNfsGenFlagWritable	FALSE	Leaving unset is not harmful
		(see 11.10.1 of [RFC5661]).
fedfsNfsGenFlagGoing	FALSE	NFS client will detect a
		migration event if the FSL
		becomes unavailable.
fedfsNfsGenFlagSplit	TRUE	Safe to assume that the FSL
		is split.
fedfsNfsTransFlagRdma	TRUE	NFS client will detect if
		RDMA access is available.
fedfsNfsClassSimul	0	0 is treated as non-matching
		(see 11.10.1 of [RFC5661]).
fedfsNfsClassHandle	0	See fedfsNfsClassSimul note.
fedfsNfsClassFileid	0	See fedfsNfsClassSimul note.
fedfsNfsClassWritever	0	See fedfsNfsClassSimul note.
fedfsNfsClassChange	0	See fedfsNfsClassSimul note.
fedfsNfsClassReaddir	0	See fedfsNfsClassSimul note.
fedfsNfsReadRank	0	Highest value ensures FSL
		will be tried.
fedfsNfsReadOrder	0	See fedfsNfsReadRank note.
fedfsNfsWriteRank	0	See fedfsNfsReadRank note.
fedfsNfsWriteOrder	0	See fedfsNfsReadRank note.
fedfsNfsVarSub	FALSE	NFSv4 does not define
		variable substitution in
		paths.
fedfsNfsValidFor	0	Indicates no appropriate
		refetch interval (see
		11.10.2 of [RFC5661]).
+------------------------+----------+------------------------------+

5.1.4. Delete an FSL

   This operation deletes an FSL record. The admin requests the NSDB
   node storing the fedfsFsl to delete it from its database. This
   operation does not result in fileset data being deleted on any
   fileserver.

5.1.4.1. LDAP Request

   The admin sends an LDAP DELETE request to the NSDB node to remove the
   FSL.

```
        dn: fedfsFslUuid=$FSLUUID,fedfsFsnUuid=$FSNUUID,$NCE
        changeType: delete
```

For example, if the $FSNUUID is "e8c4761c-eb3b-4307-86fc-
f702da197966", the $FSLUUID is "ba89a802-41a9-44cf-8447-
dda367590eb3", and the $NCE is "o=fedfs", the operation would be (for
readability the DN is split into two lines):

```
        dn: fedfsFslUuid=ba89a802-41a9-44cf-8447-dda367590eb3,
         fedfsFsnUuid=e8c4761c-eb3b-4307-86fc-f702da197966,o=fedfs
        changeType: delete
```

5.1.5. Update an FSL

This operation updates the attributes of a given FSL. This command
results in a change in the attributes of the fedfsFsl at the NSDB
node maintaining this FSL. The values of the fedfsFslUuid and
fedfsFsnUuid attributes MUST NOT change during an FSL update.

5.1.5.1. LDAP Request

The admin sends an LDAP MODIFY request to the NSDB node to update the
FSL.

```
        dn: fedfsFslUuid=$FSLUUID,fedfsFsnUuid=$FSNUUID,$NCE
        changeType: modify
        replace: $ATTRIBUTE-TYPE
```

For example, if the $FSNUUID is "e8c4761c-eb3b-4307-86fc-
f702da197966", the $FSLUUID is "ba89a802-41a9-44cf-8447-
dda367590eb3", the $NCE is "o=fedfs", and the administrator wished to
change the NFS read rank to 10, the operation would be (for
readability the DN is split into two lines):

```
        dn: fedfsFslUuid=ba89a802-41a9-44cf-8447-dda367590eb3,
         fedfsFsnUuid=e8c4761c-eb3b-4307-86fc-f702da197966,o=fedfs
        changeType: modify
        replace: fedfsNfsReadClass
        fedfsNfsReadRank: 10
```

5.2. NSDB Operations for Fileservers

5.2.1. NSDB Container Entry (NCE) Enumeration

   To find the NCEs for the NSDB nsdb.example.com, a fileserver would do
   the following:

```
    nce_list = empty
    connect to the LDAP directory at nsdb.example.com
    for each namingContext value $BAR in the root DSE
        /* $BAR is a DN */
        query for a fedfsNceDN value at $BAR
        /*
         * The RFC 4516 LDAP URL for this search would be
         *
         *  ldap://nsdb.example.com:389/$BAR?fedfsNceDN??
         *                   (objectClass=fedfsNsdbContainerInfo)
         *
         */
        if a fedfsNceDN value is found
            add the value to the nce_list
```

5.2.2. Lookup FSLs for an FSN

   Using an LDAP search, the fileserver can obtain all of the FSLs for a
   given FSN. The FSN's fedfsFsnUuid is used as the search key. The
   following examples use the LDAP Universal Resource Identifier (URI)
   format defined in [RFC4516].

   To obtain a list of all FSLs for $FSNUUID on the NSDB named
   $NSDBNAME, the following search can be used (for readability the URI
   is split into two lines):

```
        for each $NCE in nce_list
            ldap://$NSDBNAME/fsnUuid=$FSNUUID,$NCE??one?
                (objectClass=fedfsFsl)
```

   This search is for the children of the object with DN
   "fedfsFsnUuid=$FSNUUID,$NCE" with a filter for
   "objectClass=fedfsFsl". The scope value of "one" restricts the
   search to the entry's children (rather than the entire subtree below
   the entry) and the filter ensures that only FSL entries are returned.

   For example if $NSDBNAME is "nsdb.example.com", $FSNUUID is
   "e8c4761c-eb3b-4307-86fc-f702da197966", and $NCE is "o=fedfs", the
   search would be (for readability the URI is split into three lines):

```
ldap://nsdb.example.com/
        fsnUuid=e8c4761c-eb3b-4307-86fc-f702da197966,o=fedfs
        ??one?(objectClass=fedfsFsl)
```

The following search can be used to obtain only the NFS FSLs for $FSNUUID on the NSDB named $NSDBNAME (for readability the URI is split into two lines):

```
for each $NCE in nce_list
    ldap://$NSDBNAME/fsnUuid=$FSNUUID,$NCE??one?
        (objectClass=fedfsNfsFsl)
```

This also searches for the children of the object with DN "fedfsFsnUuid=$FSNUUID,$NCE", but the filter for "objectClass = fedfsNfsFsl" restricts the results to only NFS FSLs.

For example if $NSDBNAME is nsdb.example.com, $FSNUUID is "e8c4761c-eb3b-4307-86fc-f702da197966", and $NCE is "o=fedfs",the search would be (for readability the URI is split into three lines):

```
ldap://nsdb.example.com/
        fsnUuid=e8c4761c-eb3b-4307-86fc-f702da197966,o=fedfs
        ??one?(objectClass=fedfsNfsFsl)
```

The fileserver will generate a referral based on the set of FSLs returned by these queries using the process described in Section 2.8.4.

5.3. NSDB Operations and LDAP Referrals

The LDAPv3 protocol defines an LDAP referral mechanism that allows an LDAP server to redirect an LDAP client. LDAPv3 defines two types of LDAP referrals: the Referral type defined in Section 4.1.10 of [RFC4511] and the SearchResultReference type defined in Section 4.5.3 of [RFC4511]. In both cases, the LDAP referral lists one or more URIs for services that can be used to complete the operation. In the remainder of this document, the term LDAP referral is used to indicate either of these types.

If an NSDB operation results in an LDAP referral, the NSDB client MAY follow the LDAP referral. An NSDB client's decision to follow an LDAP referral is implementation and configuration dependent. For example, an NSDB client might be configured to follow only those LDAP

referrals that were received over a secure channel, or only those
that target an NSDB that supports encrypted communication. If an
NSDB client chooses to follow an LDAP referral, the NSDB client MUST
process the LDAP referral and prevent looping as described in Section
4.1.10 of [RFC4511].

6. Security Considerations

Both the NFSv4 and LDAPv3 protocols provide security mechanisms.
When used in conjunction with the federated filesystem protocols
described in this document, the use of these mechanisms is
RECOMMENDED. Specifically, the use of RPCSEC_GSS [RFC2203], which is
built on the GSS-API [RFC2743], is RECOMMENDED on all NFS connections
between a file-access client and fileserver. The "Security
Considerations" sections of the NFSv4.0 [3530bis] and NFSv4.1
[RFC5661] specifications contain special considerations for the
handling of GETATTR operations for the fs_locations and
fs_locations_info attributes.

NSDB nodes and NSDB clients MUST implement support for TLS [RFC5246],
as described in [RFC4513]. For all LDAP connections established by
the federated filesystem protocols, the use of TLS is RECOMMENDED.

If an NSDB client chooses to follow an LDAP referral, the NSDB client
SHOULD authenticate the LDAP referral's target NSDB using the target
NSDB's credentials (not the credentials of the NSDB that generated
the LDAP referral). The NSDB client SHOULD NOT follow an LDAP
referral that targets an NSDB for which it does not know the NSDB's
credentials.

Within a federation, there are two types of components an attacker
may compromise: a fileserver and an NSDB.

If an attacker compromises a fileserver, the attacker can interfere
with a file-access client's filesystem I/O operations (e.g., by
returning fictitious data in the response to a read request) or
fabricating a referral. The attacker's abilities are the same
regardless of whether or not the federation protocols are in use.
While the federation protocols do not give the attacker additional
capabilities, they are additional targets for attack. The LDAP
protocol described in Section 5.2 SHOULD be secured using the methods
described above to defeat attacks on a fileserver via this channel.

If an attacker compromises an NSDB, the attacker will be able to
forge FSL information and thus poison the fileserver's referral
information. Therefore an NSDB should be as secure as the
fileservers which query it. The LDAP operations described in

Section 5 SHOULD be secured using the methods described above to
defeat attacks on an NSDB via this channel.

A fileserver binds anonymously when performing NSDB operations. Thus
the contents and distinguished names of FSN and FSL records are
required to be readable by anyone who can bind anonymously to an NSDB
service. Section 2.12 presents the security considerations in the
choice of the type of UUID used in these records.

It should be noted that the federation protocols do not directly
provide access to filesystem data. The federation protocols only
provide a mechanism for building a namespace. All data transfers
occur between a file-access client and fileserver just as they would
if the federation protocols were not in use. As a result, the
federation protocols do not require new user authentication and
authorization mechanisms or require a fileserver to act as a proxy
for a client.

7. IANA Considerations

7.1. Registry for the fedfsAnnotation Key Namespace

This document defines the fedfsAnnotation key in Section 4.2.1.6.
The fedfsAnnotation key namespace is to be managed by IANA. IANA is
to create and maintain a new registry entitled "FedFS Annotation
Keys". The location of this registry should be under a new heading
called "Federated File System (FedFS) Parameters". The URL address
can be based off of the new heading name, for example:
http://www.iana.org/assignments/fedfs-parameters/ ...

Future registrations are to be administered by IANA using the "First
Come First Served" policy defined in [RFC5226]. Registration
requests MUST include the key (a valid UTF-8 string of any length), a
brief description of the key's purpose, and an email contact for the
registration. For viewing, the registry should be sorted
lexicographically by key. There are no initial assignments for this
registry.

7.2. Registry for FedFS Object Identifiers

Using the process described in [RFC2578], one of the authors was
assigned the Internet Private Enterprise Numbers range
1.3.6.1.4.1.31103.x. Within this range, the subrange
1.3.6.1.4.1.31103.1.x is permanently dedicated for use by the
federated file system protocols. Unassigned OIDs in this range MAY
be used for Private Use or Experimental Use as defined in [RFC5226].
New permanent FedFS OID assignments MUST NOT be made using OIDs in

this range.

IANA is to create and maintain a new registry entitled "FedFS Object Identifiers" for the purpose of recording the allocations of FedFS Object Identifiers (OIDs) specified by this document. No future allocations in this registry are allowed.

The location of this registry should be under the heading "Federated File System (FedFS) Parameters", created in Section 7.1. The URL address can be based off of the new heading name, for example: http://www.iana.org/assignments/fedfs-parameters/ ...

For viewing, the registry should be sorted numerically by OID value. The contents of the FedFS Object Identifiers registry are given in Table 1.

Note: A descriptor designated below as "historic" reserves an OID used in a past version of the NSDB protocol. Registering such OIDs retains compatibility among existing implementations of the NSDB protocol. This document does not otherwise refer to historic OIDs.

```
+-------------------------+-------------------------+-----------+
| OID                     | Description             | Reference |
+-------------------------+-------------------------+-----------+
| 1.3.6.1.4.1.31103.1.1   | fedfsUuid               | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.2   | fedfsNetAddr            | historic  |
| 1.3.6.1.4.1.31103.1.3   | fedfsNetPort            | historic  |
| 1.3.6.1.4.1.31103.1.4   | fedfsFsnUuid            | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.5   | fedfsNsdbName           | historic  |
| 1.3.6.1.4.1.31103.1.6   | fedfsNsdbPort           | historic  |
| 1.3.6.1.4.1.31103.1.7   | fedfsNcePrefix          | historic  |
| 1.3.6.1.4.1.31103.1.8   | fedfsFslUuid            | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.9   | fedfsFslHost            | historic  |
| 1.3.6.1.4.1.31103.1.10  | fedfsFslPort            | historic  |
| 1.3.6.1.4.1.31103.1.11  | fedfsFslTTL             | historic  |
| 1.3.6.1.4.1.31103.1.12  | fedfsAnnotation         | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.13  | fedfsDescr              | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.14  | fedfsNceDN              | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.15  | fedfsFsnTTL             | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.100 | fedfsNfsPath            | historic  |
| 1.3.6.1.4.1.31103.1.101 | fedfsNfsMajorVer        | historic  |
| 1.3.6.1.4.1.31103.1.102 | fedfsNfsMinorVer        | historic  |
| 1.3.6.1.4.1.31103.1.103 | fedfsNfsCurrency        | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.104 | fedfsNfsGenFlagWritable | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.105 | fedfsNfsGenFlagGoing    | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.106 | fedfsNfsGenFlagSplit    | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.107 | fedfsNfsTransFlagRdma   | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.108 | fedfsNfsClassSimul      | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.109 | fedfsNfsClassHandle     | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.110 | fedfsNfsClassFileid     | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.111 | fedfsNfsClassWritever   | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.112 | fedfsNfsClassChange     | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.113 | fedfsNfsClassReaddir    | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.114 | fedfsNfsReadRank        | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.115 | fedfsNfsReadOrder       | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.116 | fedfsNfsWriteRank       | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.117 | fedfsNfsWriteOrder      | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.118 | fedfsNfsVarSub          | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.119 | fedfsNfsValidFor        | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.120 | fedfsNfsURI             | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.1001| fedfsNsdbContainerInfo  | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.1002| fedfsFsn                | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.1003| fedfsFsl                | RFC-TBD1  |
| 1.3.6.1.4.1.31103.1.1004| fedfsNfsFsl             | RFC-TBD1  |
+-------------------------+-------------------------+-----------+
```

Table 1

7.3. LDAP Descriptor Registration

   In accordance with Section 3.4 and Section 4 of [RFC4520], the object
   identifier descriptors defined in this document (listed below) will
   be registered via the Expert Review process.

   Subject: Request for LDAP Descriptor Registration
   Person & email address to contact for further information: See
      "Author/Change Controller"
   Specification: draft-ietf-nfsv4-federated-fs-protocol
   Author/Change Controller: IESG (iesg@ietf.org)

   Object Identifier: 1.3.6.1.4.1.31103.1.1
   Descriptor (short name): fedfsUuid
   Usage: attribute type

   Object Identifier: 1.3.6.1.4.1.31103.1.2
   Descriptor (short name): fedfsNetAddr
   Usage: attribute type (historic)

   Object Identifier: 1.3.6.1.4.1.31103.1.3
   Descriptor (short name): fedfsNetPort
   Usage: attribute type (historic)

   Object Identifier: 1.3.6.1.4.1.31103.1.4
   Descriptor (short name): fedfsFsnUuid
   Usage: attribute type

   Object Identifier: 1.3.6.1.4.1.31103.1.5
   Descriptor (short name): fedfsNsdbName
   Usage: attribute type (historic)

   Object Identifier: 1.3.6.1.4.1.31103.1.6
   Descriptor (short name): fedfsNsdbPort
   Usage: attribute type (historic)

   Object Identifier: 1.3.6.1.4.1.31103.1.7
   Descriptor (short name): fedfsNcePrefix
   Usage: attribute type (historic)

   Object Identifier: 1.3.6.1.4.1.31103.1.8
   Descriptor (short name): fedfsFslUuid
   Usage: attribute type

Object Identifier: 1.3.6.1.4.1.31103.1.9
Descriptor (short name): fedfsFslHost
Usage: attribute type (historic)

Object Identifier: 1.3.6.1.4.1.31103.1.10
Descriptor (short name): fedfsFslPort
Usage: attribute type (historic)

Object Identifier: 1.3.6.1.4.1.31103.1.11
Descriptor (short name): fedfsFslTTL
Usage: attribute type (historic)

Object Identifier: 1.3.6.1.4.1.31103.1.12
Descriptor (short name): fedfsAnnotation
Usage: attribute type

Object Identifier: 1.3.6.1.4.1.31103.1.13
Descriptor (short name): fedfsDescr
Usage: attribute type

Object Identifier: 1.3.6.1.4.1.31103.1.14
Descriptor (short name): fedfsNceDN
Usage: attribute type

Object Identifier: 1.3.6.1.4.1.31103.1.15
Descriptor (short name): fedfsFsnTTL
Usage: attribute type

Object Identifier: 1.3.6.1.4.1.31103.1.100
Descriptor (short name): fedfsNfsPath
Usage: attribute type (historic)

Object Identifier: 1.3.6.1.4.1.31103.1.101
Descriptor (short name): fedfsNfsMajorVer
Usage: attribute type (historic)

Object Identifier: 1.3.6.1.4.1.31103.1.102
Descriptor (short name): fedfsNfsMinorVer
Usage: attribute type (historic)

Object Identifier: 1.3.6.1.4.1.31103.1.103
Descriptor (short name): fedfsNfsCurrency
Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.104
      Descriptor (short name): fedfsNfsGenFlagWritable
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.105
      Descriptor (short name): fedfsNfsGenFlagGoing
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.106
      Descriptor (short name): fedfsNfsGenFlagSplit
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.107
      Descriptor (short name): fedfsNfsTransFlagRdma
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.108
      Descriptor (short name): fedfsNfsClassSimul
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.109
      Descriptor (short name): fedfsNfsClassHandle
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.110
      Descriptor (short name): fedfsNfsClassFileid
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.111
      Descriptor (short name): fedfsNfsClassWritever
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.112
      Descriptor (short name): fedfsNfsClassChange
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.113
      Descriptor (short name): fedfsNfsClassReaddir
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.114
      Descriptor (short name): fedfsNfsReadRank
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.115
      Descriptor (short name): fedfsNfsReadOrder
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.116
      Descriptor (short name): fedfsNfsWriteRank
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.117
      Descriptor (short name): fedfsNfsWriteOrder
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.118
      Descriptor (short name): fedfsNfsVarSub
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.119
      Descriptor (short name): fedfsNfsValidFor
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.120
      Descriptor (short name): fedfsNfsURI
      Usage: attribute type

      Object Identifier: 1.3.6.1.4.1.31103.1.1001
      Descriptor (short name): fedfsNsdbContainerInfo
      Usage: object class

      Object Identifier: 1.3.6.1.4.1.31103.1.1002
      Descriptor (short name): fedfsFsn
      Usage: object class

      Object Identifier: 1.3.6.1.4.1.31103.1.1003
      Descriptor (short name): fedfsFsl
      Usage: object class

      Object Identifier: 1.3.6.1.4.1.31103.1.1004
      Descriptor (short name): fedfsNfsFsl
      Usage: object class

8. Glossary

   Administrator: A user with the necessary authority to initiate
      administrative tasks on one or more servers.

Admin Entity: A server or agent that administers a collection of
   fileservers and persistently stores the namespace information.

File-access Client: Standard off-the-shelf network attached storage
   (NAS) client software that communicates with fileservers using a
   standard file-access protocol.

Federation: A set of fileserver collections and singleton
   fileservers that use a common set of interfaces and protocols in
   order to provide to file-access clients a federated namespace
   accessible through a filesystem access protocol.

Fileserver: A server that stores physical fileset data, or refers
   file-access clients to other fileservers. A fileserver provides
   access to its shared filesystem data via a file-access protocol.

Fileset: The abstraction of a set of files and the directory tree
   that contains them. A fileset is the fundamental unit of data
   management in the federation.

   Note that all files within a fileset are descendants of one
   directory, and that filesets do not span filesystems.

Filesystem: A self-contained unit of export for a fileserver, and
   the mechanism used to implement filesets. The fileset does not
   need to be rooted at the root of the filesystem, nor at the export
   point for the filesystem.

   A single filesystem MAY implement more than one fileset, if the
   file-access protocol and the fileserver permit this.

File-access Protocol: A network filesystem access protocol such as
   NFSv3 [RFC1813], NFSv4 [3530bis], or CIFS (Common Internet File
   System) [MS-SMB] [MS-SMB2] [MS-CIFS].

FSL (Fileset Location): The location of the implementation of a
   fileset at a particular moment in time. An FSL MUST be something
   that can be translated into a protocol-specific description of a
   resource that a file-access client can access directly, such as an
   fs_locations attribute (for NFSv4), or a share name (for CIFS).

FSN (Fileset Name): A platform-independent and globally unique name
   for a fileset. Two FSLs that implement replicas of the same
   fileset MUST have the same FSN, and if a fileset is migrated from
   one location to another, the FSN of that fileset MUST remain the
   same.

Junction: A filesystem object used to link a directory name in the
   current fileset with an object within another fileset. The
   server-side "link" from a leaf node in one fileset to the root of
   another fileset.

Namespace: A filename/directory tree that a sufficiently authorized
   file-access client can observe.

NSDB (Namespace Database) Service: A service that maps FSNs to FSLs.
   The NSDB may also be used to store other information, such as
   annotations for these mappings and their components.

NSDB Node: The name or location of a server that implements part of
   the NSDB service and is responsible for keeping track of the FSLs
   (and related info) that implement a given partition of the FSNs.

Referral: A server response to a file-access client access that
   directs the client to evaluate the current object as a reference
   to an object at a different location (specified by an FSL) in
   another fileset, and possibly hosted on another fileserver. The
   client re-attempts the access to the object at the new location.

Replica: A replica is a redundant implementation of a fileset. Each
   replica shares the same FSN, but has a different FSL.

   Replicas may be used to increase availability or performance.
   Updates to replicas of the same fileset MUST appear to occur in
   the same order, and therefore each replica is self-consistent at
   any moment.

   We do not assume that updates to each replica occur
   simultaneously. If a replica is offline or unreachable, the other
   replicas may be updated.

Server Collection: A set of fileservers administered as a unit. A
   server collection may be administered with vendor-specific
   software.

   The namespace provided by a server collection could be part of the
   federated namespace.

Singleton Server: A server collection containing only one server; a
   stand-alone fileserver.

9. References

9.1. Normative References

    [3530bis] Haynes, T. and D. Noveck, "NFS Version 4 Protocol",
               draft-ietf-nfsv4-rfc3530bis (Work In Progress), 2010.

    [RFC2079] Smith, M., "Definition of an X.500 Attribute Type and an
               Object Class to Hold Uniform Resource Identifiers (URIs)",
               RFC 2079, January 1997.

    [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119, March 1997.

    [RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol
               Specification", RFC 2203, September 1997.

    [RFC2578] McCloghrie, K., Ed., Perkins, D., Ed., and J.
               Schoenwaelder, Ed., "Structure of Management Information
               Version 2 (SMIv2)", STD 58, RFC 2578, April 1999.

    [RFC2743] Linn, J., "Generic Security Service Application Program
               Interface Version 2, Update 1", RFC 2743, January 2000.

    [RFC2849] Good, G., "The LDAP Data Interchange Format (LDIF) -
               Technical Specification", RFC 2849, June 2000.

    [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO
               10646", STD 63, RFC 3629, November 2003.

    [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
               Resource Identifier (URI): Generic Syntax", STD 66,
               RFC 3986, January 2005.

    [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally
               Unique IDentifier (UUID) URN Namespace", RFC 4122,
               July 2005.

    [RFC4510] Zeilenga, K., "Lightweight Directory Access Protocol
               (LDAP): Technical Specification Road Map", RFC 4510,
               June 2006.

    [RFC4511] Sermersheim, J., "Lightweight Directory Access Protocol
               (LDAP): The Protocol", RFC 4511, June 2006.

    [RFC4512] Zeilenga, K., "Lightweight Directory Access Protocol
               (LDAP): Directory Information Models", RFC 4512,
               June 2006.

    [RFC4513] Harrison, R., "Lightweight Directory Access Protocol

                (LDAP): Authentication Methods and Security Mechanisms",
                RFC 4513, June 2006.

    [RFC4516]   Smith, M. and T. Howes, "Lightweight Directory Access
                Protocol (LDAP): Uniform Resource Locator", RFC 4516,
                June 2006.

    [RFC4517]   Legg, S., "Lightweight Directory Access Protocol (LDAP):
                Syntaxes and Matching Rules", RFC 4517, June 2006.

    [RFC4519]   Sciberras, A., "Lightweight Directory Access Protocol
                (LDAP): Schema for User Applications", RFC 4519,
                June 2006.

    [RFC4520]   Zeilenga, K., "Internet Assigned Numbers Authority (IANA)
                Considerations for the Lightweight Directory Access
                Protocol (LDAP)", BCP 64, RFC 4520, June 2006.

    [RFC4530]   Zeilenga, K., "Lightweight Directory Access Protocol
                (LDAP) entryUUID Operational Attribute", RFC 4530,
                June 2006.

    [RFC5226]   Narten, T. and H. Alvestrand, "Guidelines for Writing an
                IANA Considerations Section in RFCs", BCP 26, RFC 5226,
                May 2008.

    [RFC5234]   Crocker, D. and P. Overell, "Augmented BNF for Syntax
                Specifications: ABNF", STD 68, RFC 5234, January 2008.

    [RFC5246]   Dierks, T. and E. Rescorla, "The Transport Layer Security
                (TLS) Protocol Version 1.2", RFC 5246, August 2008.

    [RFC5661]   Shepler, S., Eisler, M., and D. Noveck, "Network File
                System (NFS) Version 4 Minor Version 1 Protocol",
                RFC 5661, January 2010.

9.2. Informative References

    [AFS]   Howard, J., "An Overview of the Andrew File System",
                Proceeding of the USENIX Winter Technical Conference ,
                1988.

    [FEDFS-ADMIN]
                Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M.
                Naik, "Administration Protocol for Federated Filesystems",
                draft-ietf-nfsv4-federated-fs-admin (Work In Progress),
                2010.

   [MS-CIFS]   Microsoft Corporation, "Common Internet File System (CIFS)
               Protocol Specification", MS-CIFS 2.0, November 2009.

   [MS-SMB]   Microsoft Corporation, "Server Message Block (SMB)
               Protocol Specification", MS-SMB 17.0, November 2009.

   [MS-SMB2]   Microsoft Corporation, "Server Message Block (SMB) Version
               2 Protocol Specification", MS-SMB2 19.0, November 2009.

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

   [RFC2224]   Callaghan, B., "NFS URL Scheme", RFC 2224, October 1997.

   [RFC3254]   Alvestrand, H., "Definitions for talking about
               directories", RFC 3254, April 2002.

   [RFC5662]   Shepler, S., Eisler, M., and D. Noveck, "Network File
               System (NFS) Version 4 Minor Version 1 External Data
               Representation Standard (XDR) Description", RFC 5662,
               January 2010.

   [RFC5716]   Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M.
               Naik, "Requirements for Federated File Systems", RFC 5716,
               January 2010.

   [RFC6641]   Everhart, C., Adamson, W., and J. Zhang, "Using DNS SRV to
               Specify a Global File Namespace with NFS Version 4",
               RFC 6641, June 2012.

Appendix A. Acknowledgments

   The authors and editor would like to thank Craig Everhart and Manoj
   Naik, who were co-authors of an earlier version of this document. In
   addition, we would like to thank Andy Adamson, Paul Lemahieu, Mario
   Wurzl, and Robert Thurlow for helping to author this document.

   We would like to thank George Amvrosiadis, Trond Myklebust, Howard
   Chu, and Nicolas Williams for their comments and review.

   The editor gratefully acknowledges the IESG reviewers, whose
   constructive comments helped make this a much stronger document.

   Finally, we would like to thank Andy Adamson, Rob Thurlow, and Tom
   Haynes for helping to get this document out the door.

   The extract.sh shell script and formatting conventions were first

   described by the authors of the NFSv4.1 XDR specification [RFC5662].

Authors' Addresses

   James Lentini
   NetApp
   1601 Trapelo Rd, Suite 16
   Waltham, MA 02451
   US

   Phone: +1 781-768-5359
   Email: jlentini@netapp.com

   Daniel Ellard
   Raytheon BBN Technologies
   10 Moulton Street
   Cambridge, MA 02138
   US

   Phone: +1 617-873-8004
   Email: dellard@bbn.com

   Renu Tewari
   IBM Almaden
   650 Harry Rd
   San Jose, CA 95120
   US

   Email: tewarir@us.ibm.com

   Charles Lever (editor)
   Oracle Corporation
   1015 Granger Avenue
   Ann Arbor, MI 48104
   US

   Phone: +1 248-614-5091
   Email: chuck.lever@oracle.com

                     NFS operation over IPv4 and IPv6
                     draft-ietf-nfsv4-ipv4v6-00.txt

   Abstract

   This Internet-Draft provides the description of problem set faced by
   NFS and its various side band protocols when implemented over IPv4
   and IPv6 networks in various deployment scenarios. Solution to the
   various problems are also given in the draft and are sought for
   approval in the respective NFS and side band protocol versions.

   Foreword

   This "forward" section is an unnumbered section that is not included
   in the table of contents. It is primarily used for the IESG to make
   comments about the document. It can also be used for comments about
   the status of the document and sometimes is used for the RFC2119
   requirements language statement.

   Requirements Language

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

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 21, 2011.

Copyright Notice

Table of Contents

1. Terminology

   Host: Used to refer to the client or the server where the specific(s)
   of client or the server does not matter.

   IPv4: Internet Protocol Version 4.

   IPv6: Internet Protocol Version 6.

   NFS: Used to refer to Network File System irrespective of the
   version.

   NFSv2: Network File System Protocol Version 2.

   NFSv3: Network File System Protocol version 3.

   NFSv4: Network File System Protocol version 4.

   NFSv4.1: Network File System Protocol version 4.1.

   NLM: Network Lock Manager Protocol.

   NSM: Network Status Monitor Protocol.

   Operation: Refers to the NFS operation when its mode of request or
   response is inconsequential.

2. Introduction

   This draft addresses problems associated with operating NFS in an
   environment that has a mix of IPv4 and IPv6.

3. Various Deployment Scenarios

   The various deployment scenarios involving a mix of IPv4 and IPv6,
   are as follows:

   (a) Client in IPv4-only network and Server in IPv6-only network.
   (b) Client in IPv6-only network and Server in IPv4-only network.
   (c) Client in IPv4 and IPv6 capable network and Server too in IPv4
        and IPv6 capable network.

   The first two scenarios can be called asymmetric single stack mode.
   The third scenario can be called dual stack mode.

   Note: These scenarios -

(a) Client in IPv4-only network and Server in IPv4-only network.
(b) Client in IPv6 only network and Server in IPv6-only network.

can be referred to as symmetric single stack mode. They are not
discussed in this document, as the focus of this document is
scenarios that have a mix of IPv4 and IPv6.

The problems discussed below are primarily related to sharing of NFS
state across different protocol address families. State come into
picture in NFS, in case of NLM/NSM, and NFSv4.

4. PORTMAP and RPCBIND

Clients SHOULD use PORTMAP (version 2) while querying IPv4 server
addresses, and RPCBIND (version 3/4) while querying IPv6 server
addresses.

Similarly, servers should use PORTMAP (version 2) while querying
clients for making callbacks to IPv4 client addresses and RPCBIND
(version 3/4) while querying clients making callbacks to IPv6 client
addresses.

Callbacks from server to client are needed in case of port
information verification (NFSv4), asynchronous lock requests (NLM),
and delegation recalls (NFSv4).

5. NLM and NSM

Clients and servers should use the "caller_name" (in the NLM_LOCK
call), and the "mon_name" (in the SM_NOTIFY call) as the identity of
the caller. This will make the identify of the caller independent of
the protocol address family, and will help in proper operation in the
situations described below in this section.

A dual stack NSM server implementation with persistent recording of
source IP address, SHOULD record at least one IPv4 and one IPv6
address for the client (from the caller_name in the NLM_LOCK
request), so that in case of a reboot, it can send out NOTIFY
messages to the client via either/both protocol address families.

This will ensure proper operation in scenarios like these :

(a) Client1 connects to the server using IPv6 address.
(b) Client2 connects to the server using IPv4 address.
(c) The server switches from dual stack to single stack mode of
     operation.
(d) Server restarts.

Step 'c' can happen due to a network or interface disruption, or it
could also happen as part of step 'd' (due to administrative action
during step 'd'). Either way, it will result in loss of ability of
the server to communicate with the clients via one of the protocol
address families.

To handle such scenarios, the server SHOULD associate one IP address
for each protocol address family, with the client (caller_name from
the NLM_LOCK call). Otherwise, after step 'd', the server will not
be able to send a SM_NOTIFY to some of the clients. This will result
in those clients incorrectly assuming that the server is holding
their locks, when infact the server is not.

When clients receive a SM_NOTIFY from a server via one address
family, they SHOULD try to determine whether they hold locks on that
server (mon_name in the SM_NOTIFY call) via the other address family,
and if so, they SHOULD reclaim those locks too from the server.

Similarly, to handle the scenario where a dual stack client switches
to a single stack mode, and restarts, a server, when it receives a
SM_NOTIFY from a client on one address family, should try to
determine whether it holds any lock for that client (mon_name in the
SM_NOTIFY call), on another address family, and if so, it should
clear those locks too.

6. Client Identification

In the case of NFSv4.1, the short hand clientid is very similar to
NFSv4.0 clientid. Since states are tied to clientid, state sharing
across and within sessions are immune to individual connection
failures. The sessions from individual connections of an address
family can be failed over to another address family if available.

For NFSv4 however, RFC3530 [RFC3530] says - that a client MUST send a
different client string in SETCLIENTID to a different destination
addresses(s)/family of address(s). Even if the same server is
servicing on a different network address/address family the server
MUST return a different clientid to the client. This is to prevent
confusion on the client side as there is no way of determining
whether the server to which the client is connecting again is the
same or not.

The client using different client strings for different network
addresses / address families might result in a case that the multiple
requests from the same client conflict with each other on a
multihomed server, and result in revocation of delegation. This can
happen in this scenario:

(a) Client establishes a connection to server on address X and then
     opens the file Z in write mode.
(b) Server grants the client a write delegation.
(c) The connection the client had established with server address X
     in step a), breaks for some reason. Client establishes another
     connection with server address Y, and then tries to open the
     file Z.

In step c) as client is trying to connect to a different server
address/address family, it would send the SETCLIENTID with different
client string than in step a). Since servers generate clientid based
on client string, the clientid generated by the server in step c)
will be different than the clientid generated by the server in step
a). The server will then end up revoking the delegation granted in
step b).

Step c) can happen if the client side faced a disruption on one of
its address families and then connected on a different address family
to the server. Example would be client connected using IPv6 in step
a) and then client IPv6 stack or interfaces faced disruption after
step b). Client then used IPv4 to connect to the server in step c).

To handle such scenarios, the implementations should do the following
-

(a) The client SHOULD use the same client string irrespective of
     which server address or address family it is communicating with.
(b) For generating the clientid, the server SHOULD use a combination
     of the client string with its own server identifier. The server
     identifier should be generated in a unique way on similar lines
     as that of the client identifier. Specifically the server
     identifier should be such that no two servers should use the
     same server identifier. An example of well generated server
     identifier can be the one that includes the following:
(c)
     (a) a) MAC address
     (b) b) Machine serial number
(d) The client SHOULD always send the SETCLIENTID as the first
     request on the connection; even if the client is retransmitting
     the request. If the clientid returned by server is the same as
     a clientid that the client has received from some server in the
     past, the client SHOULD conclude that both the connections are

          to the same server. To prevent the server from expunging the
          client due to non renewal, the client should send a RENEW even
          if it does not have a lease after a SETCLIENTID to the server.

   With the above mechanism, in the preceding example, the client string
   in step c) would have been the same as step a) and therefore the
   server would not revoke the delegation granted in step b).
   Additionally, the clientid returned by the server in setp c) would be
   the same as that in step a), and so the client would know that it is
   communicating to the same server as in step a).

7. Dual to single stack mode transition

   Dual stack implementations of NFS over IPv4 and IPv6 should ensure
   that the shutdown of one stack implementation does not leave any data
   in indeterminate state. This means that state like locks that is
   shared between both IPv4 and IPv6 paths, should be handled carefully.
   A shutdown of one path could result in a partial or complete
   unreachability to the client, temporarily or permanently. To allow
   for possible reconnects after reachability condition are restored,
   the states SHOULD be left intact. To handle scenarios where
   reachability is not expected to be restored within any reasonable
   period of time, administrative action SHOULD be used for clearing the
   appropriate states (removal, cleanup etc).

   Shutting down of one address family stack, or loss of all interfaces
   of one address family, SHOULD NOT lead to NFSv4 client states being
   removed upon lease period expiry. This is required so that server
   does not grant conflicting access to other clients via a different
   address family; otherwise, they may find data file to be in some
   inconsistent state, leading to corruption.

   Consider this scenario -

   a) An IPv6 client A is connected to the server and is accessing file
   X, and has some state on the server, for that file.

   b) The partial reachability condition happens for IPv6.

   c) An IPv4 client B connects to the server and tries to access file X
   on which the client A had states.

   If after step b), the server had removed the state, then client B
   might find the file to be in unusable state and so the state for
   client A should be maintained unless the disruption due to step b) is
   permanent, in which case the administrator needs to take some steps
   to check / restore file X to some proper state, and then clear the

state the server had for client A, for file X.

For NFSv4, one of the ways to implement the above recommendation is
that the server should mark the client and the states associated with
it as temporarily unusable but should not remove the state associated
with the clients in such case. After the complete reachability is
restored the server should go into partial restart case for only the
clients that had their state marked as temporarily unusable and thus
should allow such clients to regain their state. The server should
identify the clientid/states that are marked as temporarily unusable
and should send those client the NFSERR_STALE_CLIENTID/
NFSERR_STALE_STATEID errors, which will start the state recovery
procedure on the client side. The server can remove the client state
if the clients have not recovered the state in the grace period after
the complete reachability condition has been restored.

For example, if the partial reachability condition affected only the
clients accessing the server over IPv6, then after the reachability
is restored, the grace period should be started only for the clients
coming over IPv6. Till the time the grace period completes, clients
coming over IPv4, trying to take locks that conflict with ones being
held by IPv6 clients, should be denied.

In such cases, for NLM, the SM_NOTIFYs should be sent only to the
IPv6 clients (the ones that were affected due to partial reachability
condition).

8. NFSv4 Callback Information

   The NFSv4 server implementation SHOULD verify the netid information
   in the callbacks corresponds to respective address families. The
   netid used for IPv6 address SHOULD be tcp6 and for IPv4 addresses, it
   SHOULD be tcp. Otherwise, the callback information SHOULD be
   rejected as incorrect.

9. Reply cache tuples for NFSv4

   Reply cache implementations usually use some combination of elements
   like client address, client port, server address, protocol, RPC XID,
   etc., to index into the reply cache. Use of the client IP usually
   has a drawback; for example, a client using a different source IP
   address + port while retransmitting a request, might result in a
   reply cache miss on the server.

   In environments where there is a mix of IPv4 and IPv6, there are
   greater chances of a server seeing a different source IP + port for a

client retransmission. For example, one address family being
completely disabled on a client, might cause the client to send
retransmissions from a different address family (and therefore
different source IP), as compared to the original request.

Therefore, reply cache indexing mechanisms, that don't rely on client
IP address, will add considerable value in environments having a mix
of IPv4 and IPv6. One alternative could be using the client string
instead of the client IP address, for the indexing, as explained here
-

As mentioned in Client Identification (Section 6) -

The client SHOULD always send the same client string, irrespective of
the server address that it is communicating with. Also, the NFSv4.0
client should always send the SETCLIENTID procedure as the first
request on any connection to the server. If a request is to be
retransmitted on a different connection, the first procedure sent out
should be a SETCLIENTID with no change in the callback address or
client string or verifier. This will help the server to associate
the new connection with the clientid.

Once a connection is associated with an existing clientid (and
therefore, an existing client string), any request restransmission on
the new connection can then successfully be indexed to it's match in
the reply cache, in a NFSv4 reply cache implementation that uses the
client string instead of the client IP address, for indexing into the
reply cache.

10. Other optimizations

10.1. Address Persistence

   There are scenarios where NFS implementations need to store IP
   addresses in persistent storage, like -

   NSM monitor/notify database.

   persistent reply cache.

   In such scenarios, to support dual stack mode, or a switch to/from
   it, implementations should store the protocol address family
   information explicitly, along with the IP address. This information
   can be used during upgrades and downgrades, across versions which
   have may or may not have turned on support for NFS over IPv6.

10.2. IP addresses as keys

   Functionalities like export checks require information to be indexed
   based on client IP, for efficient insertion / updation, and lookup.

   When using IP addresses as keys in these scenarios, the variability
   of the bits in the IP addresses SHOULD be considered. In IPv6, for
   the same interface, the different addresses might differ mostly in
   the subnet part (the lower order bits are often generated from the
   MAC address of the interface and are therefore mostly static). In
   IPv4 however, that may not be the case. Depending on the
   implementation specifics, different indexing algorithms might be
   needed for IPv4 and IPv6 addresses.

10.3. NFSv4 Id Mapping

   The "dns_domain" in "user@dns_domain" as referred to in section 5.8
   [RFC3530], used to map owners, groups, users and user groups string
   principals, to internal representations (usually numeric id) at the
   client and server SHOULD be the same for the same client accessing an
   NFSv4 server simultaneously over IPv4 and IPv6.

11. Acknowledgments

   The authors would like to acknowledge Mike Eisler for reviews of the
   various versions of the draft.

12. IANA Considerations

   This memo includes no request to IANA.

13. Security Considerations

   All considerations from RFC 3530 Section 16 [RFC3530]

14. References

14.1. Normative References

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

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

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997,
              <http://xml.resource.org/public/rfc/html/rfc2119.html>.

   [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6
              (IPv6) Specification", RFC 2460, December 1998.

   [RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
              Beame, C., Eisler, M., and D. Noveck, "Network File System
              (NFS) version 4 Protocol", RFC 3530, April 2003.

   [RFC4007] Deering, S., Haberman, B., Jinmei, T., Nordmark, E., and
              B. Zill, "IPv6 Scoped Address Architecture", RFC 4007,
              March 2005.

   [RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing
              Architecture", RFC 4291, February 2006.

   [RFC4506] Eisler, M., "XDR: External Data Representation Standard",
              STD 67, RFC 4506, May 2006.

   [RFC5378] Bradner, S. and J. Contreras, "Rights Contributors Provide
              to the IETF Trust", BCP 78, RFC 5378, November 2008.

   [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol
              Specification Version 2", RFC 5531, May 2009.

   [netid_ID]
              Eisler, M., "IANA Considerations for RPC Net Identifiers
              and Universal Address Formats",
              draft-ietf-nfsv4-rpc-netid-04 (work in progress),
              December 2008.

14.2. Informative References

   [RFC1094] Nowicki, B., "NFS: Network File System Protocol
              specification", RFC 1094, March 1989.

   [RFC2624] Shepler, S., "NFS Version 4 Design Considerations",
              RFC 2624, June 1999.

   [RFC2663] Srisuresh, P. and M. Holdrege, "IP Network Address
              Translator (NAT) Terminology and Considerations",
              RFC 2663, August 1999.

   [RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W.
              Stevens, "Basic Socket Interface Extensions for IPv6",
              RFC 3493, February 2003.

   [RFC3542] Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei,
              "Advanced Sockets Application Program Interface (API) for
              IPv6", RFC 3542, May 2003.

   [RFC3593] Tesink, K., "Textual Conventions for MIB Modules Using
              Performance History Based on 15 Minute Intervals",
              RFC 3593, September 2003.

   [RFC4620] Crawford, M. and B. Haberman, "IPv6 Node Information
              Queries", RFC 4620, August 2006.

   [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax
              Specifications: ABNF", STD 68, RFC 5234, January 2008.

Authors' Addresses

   Alex RN (editor)
   NetApp
   3rd Floor, Fair Winds Block, EGL Software Park,
   Bangalore, Karnataka 560071
   IN

   Phone: +91-80-41843352
   Email: rnalex@netapp.com

   Bhargo Sunil (editor)
   NetApp
   3rd Floor, Fair Winds Block, EGL Software Park,
   Bangalore, Karnataka 560071
   IN

   Phone: +91-80-41843963
   Email: bhargo@netapp.com

   Dhawal Bhagwat
   NetApp
   3rd Floor, Fair Winds Block, EGL Software Park,
   Bangalore, Karnataka 560071
   IN

   Phone: +91-80-41843134
   Email: dhawal@netapp.com

Dipankar Roy
NetApp
3rd Floor, Fair Winds Block, EGL Software Park,
Bangalore, Karnataka 560071
IN

Phone: +91-80-41843303
Email: dipankar@netapp.com

Rishikesh Barooah
NetApp
3rd Floor, Fair Winds Block, EGL Software Park,
Bangalore, Karnataka 560071
IN

Email: rbarooah@netapp.com

                        NFS operation over IPv6
                     draft-ietf-nfsv4-ipv6-00.txt

   Abstract

      This Internet-Draft provides the description of problems faced by NFS
      and its various side band protocols, when implemented over IPv6 in
      various deployment scenarios. Solutions to the various problems are
      also given in the draft and are sought for approval.

   Foreword

      This "forward" section is an unnumbered section that is not included
      in the table of contents. It is primarily used for the IESG to make
      comments about the document. It can also be used for comments about
      the status of the document and sometimes is used for the RFC2119
      requirements language statement.

   Requirements Language

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

Copyright Notice

Table of Contents

1. Terminology

   Host: Used to refer to the client or the server where the specific(s)
   of client or the server does not matter.

   IPv4: Internet Protocol Version 4.

   IPv6: Internet Protocol Version 6.

   NFS: Used to refer to Network File System irrespective of the
   version.

   NFSv2: Network File System Protocol Version 2.

   NFSv3: Network File System Protocol version 3.

   NFSv4: Network File System Protocol version 4.

   NFSv4.1: Network File System Protocol version 4.1.

   NLM: Network Lock Manager Protocol.

   NSM: Network Status Monitor Protocol.

   Operation: Refers to the NFS operation when its mode of request or
   response is inconsequential.

2. Introduction

   NFS being a application layer protocol can operate over several
   network layer protocols. This draft addresses problems associated
   with NFS operation over an IPv6 only network.

3. RPCBIND

   NFS servers supporting IPv6 MUST support RPCBINDv3 as defined in
   [RFC1833], over IPv6. Additionally, RPCBINDv4 SHOULD be supported,
   as noted later in this section.

   RPCBINDv3/4 protocols 'use a transport-independent format for the
   transport address'. Using RPCBINDv3/4, a client can clearly
   communicate to the server which transport (IPv4/v6, TCP/UDP) it is
   interested in for contacting a service. The server can communicate
   clearly to the client, the various transports on which a service is
   available. RPCBINDv2 (aka PORTMAP) provides limited support in this
   area.

RPCBINDv4 SHOULD be supported because it introduces useful procedures --

o RPCBPROC_GETVERSADDR - to query the server for the address of a specific version of an RPC service.

o RPCBPROC_GETADDRLIST - to query the server for a list of all addresses / transports on which an RPC service is available.

Clients SHOULD use those procedures wherever those procedures enable them to get the information of interest in one go, instead of making multiple RPCBPROC_GETADDR calls.

The netid and address formats in the RPCBINDv3/4 procedures, MUST be as per those defined for netid and universal addresses, in netid_ID draft [netid_ID]. The implementation MUST NOT use IPv4 embedded IPv6 addresses defined in Section 2.5.5 [RFC4291], for the RPCBINDv3/4 procedures.

An NFS client SHOULD specify a proper universal address in a RPCBPROC_GETADDR call; specifically, it SHOULD match the server's IP address on which the client made the call.

While processing the RPCBPROC_GETADDR call, the NFS server needs to know which local address the client is querying on; the server SHOULD pull that address from the network layer instead (the local address on which the RPCBPROC_GETADDR call was received; similar to what [RFC1833] recommends for the "r_netid" parameter -

The "r_netid" field of the argument is ignored and the "r_netid" is inferred from the network identifier of the transport on which the request came in.)

4. NFSv4 Callback Information

In the case of NFSv4.0 procedure SETCLIENTID, the netid and address formats in the callback information MUST be as per those defined for netid and universal addresses, in netid_ID draft [netid_ID]. The implementation MUST NOT use IPv4 embedded IPv6 addresses defined in Section 2.5.5 [RFC4291].

5. Handling of link-local addresses in multi-homed hosts

[RFC4007] describes link-local IPv6 addresses.

There may be environments where hosts operate only with auto-

configured (link-local) addresses. NFS implementations SHOULD
support link-local addresses, so they can operate in such
environments. For example, hosts booting over the network, via NFS.
However, since link-local addresses are link-scoped, they can cause
ambiguity on multi-homed hosts.

An NFS implementation on a multi-homed host MUST keep track of the
local interface (zone) when communicating with a link-local address
of another host. Alternately, such hosts can support a default zone,
which the network layer can use when no interface info is specified
explicitly. See the 'Scope Zones' section of RFC 4007 [RFC4007] for
more on (scope) zones and their implementation.

While making a callback to an address received in a NLM LOCK call or
a NFSv4 SETCLIENTID call, a server MUST specify the local interface
via which the call needs to be made (or let the default zone be
selected, if supported).

An NFS implementation on multi-homed hosts MUST also make sure that a
link-local address of any one of it's (local) interfaces is not
advertised out in any way, via any of it's other (local) interfaces.
For instance, the address list that a NFS server returns in a
RPCBPROC_GETADDRLIST response, MUST NOT contain a link-local address
any interface other than the one on which the request was received
(which will be same as the one which the response is being sent out).

6. Acknowledgments

   The authors would like to acknowledge Mike Eisler for reviews of the
   various early versions of the draft.

7. IANA Considerations

   This memo includes no request to IANA.

8. Security Considerations

   All considerations from RFC 3530 Section 16 [RFC3530]

9. References

9.1. Normative References

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

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

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997,
              <http://xml.resource.org/public/rfc/html/rfc2119.html>.

   [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6
              (IPv6) Specification", RFC 2460, December 1998.

   [RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
              Beame, C., Eisler, M., and D. Noveck, "Network File System
              (NFS) version 4 Protocol", RFC 3530, April 2003.

   [RFC4007] Deering, S., Haberman, B., Jinmei, T., Nordmark, E., and
              B. Zill, "IPv6 Scoped Address Architecture", RFC 4007,
              March 2005.

   [RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing
              Architecture", RFC 4291, February 2006.

   [RFC4506] Eisler, M., "XDR: External Data Representation Standard",
              STD 67, RFC 4506, May 2006.

   [RFC5378] Bradner, S. and J. Contreras, "Rights Contributors Provide
              to the IETF Trust", BCP 78, RFC 5378, November 2008.

   [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol
              Specification Version 2", RFC 5531, May 2009.

   [netid_ID]
              Eisler, M., "IANA Considerations for RPC Net Identifiers
              and Universal Address Formats",
              draft-ietf-nfsv4-rpc-netid-04 (work in progress),
              December 2008.

9.2. Informative References

   [RFC1094] Nowicki, B., "NFS: Network File System Protocol
              specification", RFC 1094, March 1989.

   [RFC2624] Shepler, S., "NFS Version 4 Design Considerations",
              RFC 2624, June 1999.

   [RFC2663]   Srisuresh, P. and M. Holdrege, "IP Network Address
               Translator (NAT) Terminology and Considerations",
               RFC 2663, August 1999.

   [RFC3493]   Gilligan, R., Thomson, S., Bound, J., McCann, J., and W.
               Stevens, "Basic Socket Interface Extensions for IPv6",
               RFC 3493, February 2003.

   [RFC3542]   Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei,
               "Advanced Sockets Application Program Interface (API) for
               IPv6", RFC 3542, May 2003.

   [RFC3593]   Tesink, K., "Textual Conventions for MIB Modules Using
               Performance History Based on 15 Minute Intervals",
               RFC 3593, September 2003.

   [RFC4620]   Crawford, M. and B. Haberman, "IPv6 Node Information
               Queries", RFC 4620, August 2006.

   [RFC5234]   Crocker, D. and P. Overell, "Augmented BNF for Syntax
               Specifications: ABNF", STD 68, RFC 5234, January 2008.

Authors' Addresses

   Alex RN (editor)
   NetApp
   3rd Floor, Fair Winds Block, EGL Software Park,
   Bangalore, Karnataka 560071
   IN

   Phone: +91-80-41843352
   Email: rnalex@netapp.com

   Dhawal Bhagwat (editor)
   NetApp
   3rd Floor, Fair Winds Block, EGL Software Park,
   Bangalore, Karnataka 560071
   IN

   Phone: +91-80-41843134
   Email: dhawal@netapp.com

Dipankar Roy
NetApp
3rd Floor, Fair Winds Block, EGL Software Park,
Bangalore, Karnataka 560071
IN

Phone: +91-80-41843303
Email: dipankar@netapp.com

Rishikesh Barooah
NetApp
3rd Floor, Fair Winds Block, EGL Software Park,
Bangalore, Karnataka 560071
IN

Email: rbarooah@netapp.com

NFSv4 Working Group   S. Faibish
Internet-Draft   EMC Corporation
Intended status: Proposed Standard   D. Black
Expires: April 14, 2011   EMC Corporation
Updates: 5661, 5662   M. Eisler
                                                                 NetApp
                                                             J. Glasgow
                                                                 Google
                                                       October 14, 2010

                        pNFS Access Permissions Check
                 draft-ietf-nfsv4-pnfs-access-permissions-check-00

Status of this Memo

Copyright Notice

Abstract

   This document extends the pNFS protocol to communicate errors caused
   by inability to access data servers referenced by layouts, including
   checks performed by both clients and the MDS. The extension provides
   means for clients to communicate client-detected access denial errors
   to the MDS, including the case in which a client requests direct NFS
   access via the MDS that the MDS cannot perform.

Table of Contents

1. Introduction

   Figure 1 shows the overall architecture of a Parallel NFS (pNFS)
   system:

```
         +-----------+
         |+-----------+
         ||+-----------+                                 +-----------+
         |||           |      NFSv4.1 + pNFS             |           |
         +|| Clients   |<----------------------------->|   MDS     |
          +|           |                                 |           |
           +-----------+                                 |           |
             |||                                         +-----------+
             |||                                               |
             |||                                               |
             |||  Storage                                      |
             |||  Protocol        +-----------+                |
             ||+---------------||+-----------+  Control        |
             |+---------------|||           | Protocol  |
         +-----------------+||  Storage  |-----------+
                           +| Devices   |
                            +-----------+
```
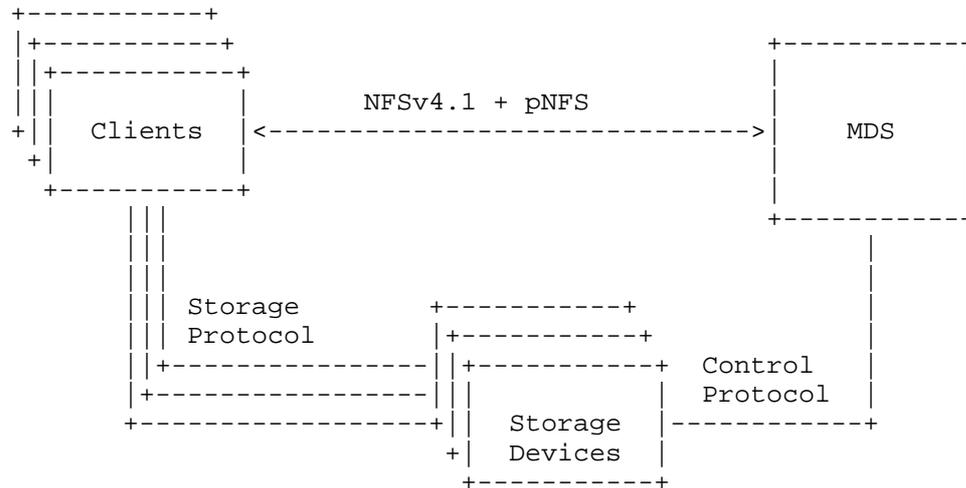
                     Figure 1 pNFS Architecture

   In this document, "storage device" is used as a general term for a
   data server and/or storage server for the file, block or object pNFS
   layouts.

   The current pNFS protocol [RFC5661] assumes that a client can access
   every storage device (SD) included in a valid layout sent by the MDS
   server, and provides no means to communicate client access failures
   to the MDS. Access failures can impair pNFS performance scaling and
   allow significant errors to go unreported. If the MDS can access all
   the storage devices involved, but the client doesn't have sufficient
   access rights to some storage devices, the client may choose to fall
   back to accessing the file system using NFSV4.1 without pNFS support;
   there are environments in which this behavior is undesirable,
   especially if it occurs silently. An important example is addition of
   a new storage device to which a large population of pNFS clients
   (e.g., 1000s) lacks access permission. Layouts granted that use this
   new device, result in client errors, requiring that all I/Os to that
   new storage device be served by the MDS server. This creates a
   performance and scalability bottleneck that may be difficult to
   detect based on I/O behavior because the other storage devices are
   functioning correctly.

The preferable approach to this scenario is to report the access
failures before any client attempts to issue any I/Os that can only
be serviced by the MDS server. This makes the problem explicit,
rather than forcing the MDS, or a system administrator, to diagnose
the performance problem caused by client I/O using NFS instead of
pNFS. There are limits to this approach because complex mount
structures may prevent a client from detecting this situation at
mount time, but at a minimum, access problems involving the root of
the mount structure can be detected.

The most suitable time for the client to report inability to access a
storage device is at mount time, but this is not always possible.
If the application uses a special tag or a switch to the mount
command (e.g., -pnfs) and syscall to declare its intention to use
pNFS, at the client, the client can check for both pNFS support and
device accessibility.

This document introduces an error reporting mechanism that is an
extension to the return of a pNFS layout; a pNFS client MAY use this
mechanism to inform the MDS that the layout is being returned because
one or more data servers are not accessible to the client. Error
reporting at I/O time is not affected because the result of an
inaccessible data server may not be an I/O error if a subsequent
retry of the operation via the MDS is successful.

There is a related problem scenario involving an MDS that cannot
access some storage devices and hence cannot perform I/Os on behalf
of a client. In the case of the block layout [RFC5663] if the MDS
lacks access to a storage device (e.g., LUN), MDS implementations
generally do not export any filesystem using that storage device. In
contrast to the block layout, MDSs for the file [RFC5661] and object
[RFC5664] layouts may be unable to access the storage devices that
store data for an exported filesystem. This enables a file or object
layout MDS to provide layouts that contain client-inaccessible
devices. For the specific case of adding a new storage device to a
filesystem, MDS issuance of test I/Os to the newly added device
before using it in layouts avoids this problem scenario, but does not
cover loss of access to existing storage devices at a later time.

In addition, [RFC5661] states that a client can write through or read
from the MDS, even if it has a layout; this assumes that the MDS can
access all the storage devices. This document makes that assumed
access an explicit requirement.

2. Conventions used in this document

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

3. Changes to Operation 51: LAYOUTRETURN (RFC 5661)

   The existing LAYOUTRETURN operation is extended by introducing three
   new layout return types that correspond to the existing types:

   o LAYOUT4_RET_REC_FILE_NO_ACCESS at file scope;

   o LAYOUT4_RET_REC_FSID_NO_ACCESS at fsid scope; and

   o LAYOUT4_RET_REC_ALL_NO_ACCESS at client scope.

   The first return type returns the layout for an individual file and
   informs the server that the reason for the return is a storage device
   connectivity problem. The second return type performs that function
   for all layouts held by the client for the filesystem that
   corresponds to the current filehandle used for the LAYOUTRETURN
   operation. The third return type performs that function for all
   layouts held by the client; it is intended for situations in which a
   device is shared across all or most of the filesystems from a server
   for which the client has layouts.

3.1. ARGUMENT (18.44.1)

   The ARGUMENT specification of the LAYOUTRETURN operation in section
   18.44.1 of [RFC5661] is replaced by the following XDR code [XDR]:

```
   /* Constants used for new LAYOUTRETURN and CB_LAYOUTRECALL */
   const LAYOUT4_RET_REC_FILE       = 1;
   const LAYOUT4_RET_REC_FSID       = 2;
   const LAYOUT4_RET_REC_ALL        = 3;
   const LAYOUT4_RET_REC_FILE_NO_ACCESS    = 4;
   const LAYOUT4_RET_REC_FSID_NO_ACESSS    = 5;
   const LAYOUT4_RET_REC_ALL_NO_ACCESS     = 6;

   enum layoutreturn_type4 {
        LAYOUTRETURN4_FILE = LAYOUT4_RET_REC_FILE,
        LAYOUTRETURN4_FSID = LAYOUT4_RET_REC_FSID,
        LAYOUTRETURN4_ALL  = LAYOUT4_RET_REC_ALL,
```

```
        LAYOUTRETURN4_FILE_NO_ACCESS = LAYOUT4_RET_REC_FILE_NO_ACCESS,
        LAYOUTRETURN4_FSID_NO_ACCESS = LAYOUT4_RET_REC_FSID_NO_ACCESS,
        LAYOUTRETURN4_ALL_NO_ACCESS  = LAYOUT4_RET_REC_ALL_NO_ACCESS
};

struct layoutreturn_file4 {
        offset4         lrf_offset;
        length4         lrf_length;
        stateid4        lrf_stateid;
        /* layouttype4 specific data */
        opaque          lrf_body<>;
};

struct layoutreturn_device_no_access4 {
        deviceid4       lrdna_deviceid;
        nfsstat4        lrdna_status;
};

struct layoutreturn_file_no_access4 {
        offset4         lrfna_offset;
        length4         lrfna_length;
        stateid4        lrfna_stateid;
        deviceid4       lrfna_deviceid;
        nfsstat4        lrfna_status;
        /* layouttype4 specific data */
        opaque          lrfna_body<>;
};

union layoutreturn4 switch(layoutreturn_type4 lr_returntype) {
        case LAYOUTRETURN4_FILE:
                layoutreturn_file4              lr_layout;
        case LAYOUTRETURN4_FILE_NO_ACCESS:
                layoutreturn_file_no_access4    lr_layout_na;
        case LAYOUTRETURN4_FSID_NO_ACCESS:
        case LAYOUTRETURN4_ALL_NO_ACCESS:
                layoutreturn_device_no_access4      lr_device<>;
        default:
                void;
};
```

3.2. RESULT (18.44.2)

   The RESULT of the LAYOUTRETURN operation is unchanged; see section
   18.44.2 of [RFC5661].

3.3. DESCRIPTION (18.44.3)

   The following text is added to the end of the LAYOUTRETURN operation
   DESCRIPTION in section 18.44.3 of [RFC5661]:

   There are three NO_ACCESS layoutreturn_type4 values that indicate a
   persistent lack of client ability to access storage device(s),
   LAYOUT4_RET_REC_FILE_NO_ACCESS, LAYOUT4_RET_REC_FSID_NO_ACCESS and
   LAYOUT4_RET_REC_ALL_NO_ACCESS. A client uses these return types to
   return a layout (or portion thereof) for a file, return all layouts
   for an FSID or all layouts from that server held by the client, and
   in all cases to inform the server that the reason for the return is
   the client's inability to access one or more storage devices. The
   same stateid may be used or the client MAY force use of a new stateid
   in order to report a new error.

   An NFS error value (nfsstat4) is included for each device for these
   three NO_ACCESS return types to provide additional information on the
   cause. The allowed NFS errors are those that are valid for an NFS
   READ or WRITE operation, and NFS4ERR_NXIO is also allowed to report
   an inaccessible device. The server SHOULD log the received NFS error
   value, but that error value does not affect server processing of the
   LAYOUTRETURN operation. All uses of the NO_ACCESS layout return types
   that report NFS errors SHOULD be logged by the client.

   The client MAY use the new LAYOUT4_RET_REC_FILE_NO_ACCESS when only
   one file, or a small number of files are affected. If the access
   problem affects multiple devices, the client may use multiple file
   layout return operations; each return operation SHOULD return a
   layout extent obtained from the device for which an error is being
   reported. In contrast, both LAYOUT4_RET_REC_FSID_NO_ACCESS and
   LAYOUT4_RET_REC_ALL_NO_ACCESS include an array of <device, status>
   pairs to enable a single operation to report errors for multiple
   devices in a single operation.

3.4. IMPLEMENTATION (18.44.4)

   The following text is added to the end of the LAYOUTRETURN operation
   IMPLEMENTATION in section 18.4.4 of [RFC5661]:

A client that expects to use pNFS for a mounted filesystem SHOULD
check for pNFS support at mount time. This check SHOULD be performed
by sending a GETDEVICELIST operation, followed by layout-type-
specific checks for accessibility of each storage device returned by
GETDEVICELIST. If the NFS server does not support pNFS, the
GETDEVICELIST operation will be rejected with an NFS4ERR_NOTSUPP
error; in this situation it is up to the client to determine whether
it is acceptable to proceed with NFS-only access.

Clients are expected to tolerate transient storage device errors, and
hence clients SHOULD NOT use the NO_ACCESS layout return types for
device access problems that may be transient. The methods by which a
client decides whether an access problem is transient vs. persistent
are implementation-specific, but may include retrying I/Os to a data
server under appropriate conditions.

When an I/O fails because a storage device is inaccessible, the
client SHOULD retry the failed I/O via the MDS. In this situation,
before retrying the I/O, the client SHOULD return the layout, or
inaccessible portion thereof, and SHOULD indicate which storage
device or devices was or were inaccessible. If the client does not do
this, the MDS may issue a layout recall callback in order to perform
the retried I/O.

Backwards compatibility may require a client to perform two layout
return operations to deal with servers that don't implement the
NO_ACCESS layoutreturn_type4 values and hence respond to them with
NFS4ERR_INVAL. In this situation, the client SHOULD perform an
ordinary layout return operation and remember that the new layout
NO_ACCESS return types are not to be used with that server.

The metadata server (MDS) SHOULD NOT use storage devices in pNFS
layouts that are not accessible to the MDS. At a minimum, the server
SHOULD check its own storage device accessibility before exporting a
filesystem that supports pNFS and when the device configuration for
such an exported filesystem is changed (e.g., to add a storage
device).

If an MDS is aware that a storage device is inaccessible to a client,
the MDS SHOULD NOT include that storage device in any pNFS layouts
sent to that client. An MDS SHOULD react to a client return of
inaccessible layouts by not using the inaccessible storage devices in
layouts for that client, but the MDS is not required to indefinitely
retain per-client storage device inaccessibility information. An MDS
is also not required to automatically reinstate use of a previously
inaccessible storage device; administrative intervention may be
required instead.

   A client MAY perform I/O via the MDS even when the client holds a
   layout that covers the I/O; servers MUST support this client
   behavior, and MAY recall layouts as needed to complete I/Os.

3.4.1. Storage Device Error Mapping (18.44.4.1, new)

   The following text is added as new subsection 18.44.4.1 of [RFC5661]:

   An NFS error value is sent for each device that the client reports as
   inaccessible via a NO_ACCESS layout return type. In general:

   o If the client is unable to access the storage device, NFS4ERR_NXIO
      SHOULD be used.

   o If the client is able to access the storage device, but permission
      is denied, NFS4ERR_ACCESS SHOULD be used.

   Beyond these two rules, error code usage is layout-type specific:

   o For the pNFS file layout, an indicative NFS error from a failed
      read or write operation on the inaccessible device SHOULD be used.

   o For the pNFS block layout, other errors from the Storage Protocol
      SHOULD be mapped to NFS4ERR_IO. In addition, the client SHOULD log
      information about the actual storage protocol error (e.g., SCSI
      status and sense data), but that information is not sent to the
      pNFS server.

   o For the pNFS object layout, occurrences of the object error types
      specified in [RFC5664] SHOULD be mapped to the following NFS
      errors for use in LAYOUTRETURN:

         o   PNFS_OSD_ERR_EIO   -> NFS4ERR_IO

         o   PNFS_OSD_ERR_NOT_FOUND   -> NFS4ERR_STALE

         o   PNFS_OSD_ERR_NO_SPACE   -> NFS4ERR_NOSPC

         o   PNFS_OSD_ERR_BAD_CRED   -> NFS4ERR_INVAL

         o   PNFS_OSD_ERR_NO_ACCESS   -> NFS4ERR_ACCESS

         o   PNFS_OSD_ERR_UNREACHABLE   -> NFS4ERR_NXIO

         o   PNFS_OSD_ERR_RESOURCE   -> NFS4ERR_SERVERFAULT

The LAYOUTRETURN NO_ACCESS return types are used for persistent
device errors; they do not replace other error reporting mechanisms
that also apply to transient errors (e.g., as specified for the
object layout in [RFC5664]).

4. Change to NFS4ERR_NXIO Usage

   This document specifies that the NFS4ERR_NXIO error SHOULD be used to
   report an inaccessible storage device. To enable that usage, this
   document updates [RFC5661] to allow use of the currently obsolete
   NFS4ERR_NXIO error in the ARGUMENT of LAYOUTRETURN; NFS4ERR_NXIO
   remains obsolete for all other uses of NFS errors.

5. Security Considerations

   This document adds a small extension to the NFSv4 LAYOUTRETURN
   operation. The NFS and pNFS security considerations in [RFC5661],
   [RFC5663] and [RFC5664] apply to the extended LAYOUTRETURN operation.

6. IANA Considerations

   There are no additional IANA considerations in this document beyond
   the IANA Considerations covered in [RFC5661].

7. Conclusions

   This draft specifies additions to the pNFS protocol addressing
   inability to access storage devices used in pNFS layouts.

8. References

8.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
             Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC5661] Shepler, S., Eisler, M., and D. Noveck, "Network File
             System (NFS) Version 4 Minor Version 1 Protocol",
             http://tools.ietf.org/html/rfc5661, January 2010.

   [RFC5663] Black, D., Glasgow, J., Fridella, S., "Parallel NFS (pNFS)
             Block/Volume Layout", http://tools.ietf.org/html/rfc5663,
             January 2010.

   [RFC5664] Halevy, B., Welch, B., Zelenka, J., "Object-Based Parallel
             NFS (pNFS) Operations", http://tools.ietf.org/html/rfc5664,
             January 2010

   [XDR]   Eisler, M., "XDR: External Data Representation Standard",
              STD 67, RFC 4506, May 2006.

Acknowledgments

Changes from draft-faibish-nfsv4-pnfs-access-permissions-check-03

   - First nfsv4 WG draft version.
   - Add ALL NO_ACCESS return type, so that there's a NO_ACCESS
     return type for every current return type.
   - Use NFS4ERR_ACCESS instead of NFS4ERR_PERM, and allow use of
     NFS4ERR_NXIO for an unreachable data server.
   - Simplify recommendation for initial access checks to only discuss
     GETDEVICELIST.
   - Add client guidance on riding through transient errors.
   - State that server does not need to indefinitely retain device
     inaccessibility information, and administrative intervention
     may be required to restore use of a previously inaccessible
     storage device.
   - Remove "MUST" requirement for layout return if retry via MDS
     fails. Add warning that if the layout isn't returned in advance
     of MDS I/O retry, the MDS may issue a callback to get it.
   - Specify allowed errors (in payload) via reference to errors
     allowed for READ and WRITE, plus allow NFS4ERR_NXIO.
   - Provide information about how to map device errors (especially
     from non-file layout types) to NFS errors.

Authors' Addresses

Sorin Faibish (editor)
EMC Corporation
228 South Street
Hopkinton, MA 01748
US

Phone: +1 (508) 249-5745
Email: sfaibish@emc.com

David L. Black
EMC Corporation
176 South Street
Hopkinton, MA 01748
US

Phone: +1 (508) 293-7953
Email: david.black@emc.com

Michael Eisler
NetApp
5765 Chase Point Circle
Colorado Springs, CO 80919
US

Phone: +1 (719) 599-9026
Email: mike@eisler.com

Jason Glasgow
Google
5 Cambridge Center, Floors 3-6
Cambridge, MA 02142
US

Phone: +1 (617) 575-1599
Email: jglasgow@google.com

              Network File System (NFS) Version 4 Protocol
                   draft-ietf-nfsv4-rfc3530bis-35.txt

   Abstract

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

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

   Requirements Language

      The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
      "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
      document are to be interpreted as described in RFC 2119 [RFC2119]
      except where "REQUIRED" and "RECOMMENDED" are used as qualifiers to
      distinguish classes of attributes as described in Section 1.3.3.2 and
      Section 5.

time. It is inappropriate to use Internet-Drafts as reference
material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 7, 2015.

Copyright Notice

Table of Contents

1. Introduction

1.1. NFS Version 4 Goals

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

   o Improved access and good performance on the Internet.

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

o Strong security with negotiation built into the protocol.

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

o Good cross-platform interoperability.

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

o Designed for protocol extensions.

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

This document, together with the companion XDR description document
[RFCNFSv4XDR], obsoletes [RFC3530] as the authoritative document
describing NFSv4. It does not introduce any over-the-wire protocol
changes, in the sense that previously valid requests remain valid.

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

[RFCNFSv4XDR], "Network File System (NFS) Version 4 External Data
Representation Standard (XDR) Description", contains the definitions
in XDR description language of the constructs used by the protocol.
Inside this document, several of the constructs are reproduced for
purposes of explanation. The reader is warned of the possibility of
errors in the reproduced constructs outside of [RFCNFSv4XDR]. For
any part of the document that is inconsistent with [RFCNFSv4XDR],
[RFCNFSv4XDR] is to be considered authoritative.

1.3. Overview of NFSv4 Features

To provide a reasonable context for the reader, the major features of
NFSv4 protocol will be reviewed in brief. This will be done to
provide an appropriate context for both the reader who is familiar
with the previous versions of the NFS protocol and the reader who is

new to the NFS protocols. For the reader new to the NFS protocols,
some fundamental knowledge is still expected. The reader should be
familiar with the XDR and RPC protocols as described in [RFC5531] and
[RFC4506]. A basic knowledge of file systems and distributed file
systems is expected as well.

1.3.1. RPC and Security

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

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

1.3.2. Procedure and Operation Structure

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

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

   The model used for COMPOUND is very simple. There is no logical OR
   or ANDing of operations. The operations combined within a COMPOUND
   request are evaluated in order by the server. Once an operation

   returns a failing result, the evaluation ends and the results of all
   evaluated operations are returned to the client.

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

1.3.3. Filesystem Model

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

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

1.3.3.1. Filehandle Types

   In previous versions of the NFS protocol, the filehandle provided by
   the server was guaranteed to be valid or persistent for the lifetime
   of the file system object to which it referred. For some server
   implementations, this persistence requirement has been difficult to
   meet. For the NFSv4 protocol, this requirement has been relaxed by
   introducing another type of filehandle, volatile. With persistent
   and volatile filehandle types, the server implementation can match
   the abilities of the file system at the server along with the
   operating environment. The client will have knowledge of the type of
   filehandle being provided by the server and can be prepared to deal
   with the semantics of each.

1.3.3.2. Attribute Types

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

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

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

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

1.3.3.3. Multi-server Namespace

A single-server namespace is the file system hierarchy that the server presents for remote access. It is a proper subset of all the file systems available locally. NFSv4 contains a number of features to allow implementation of namespaces that cross server boundaries and that allow and facilitate a non-disruptive transfer of support for individual file systems between servers. They are all based upon attributes that allow one file system to specify alternative or new locations for that file system. I.e., just as a client might traverse across local file systems on a single server, it can now traverse to a remote file system on a different server.

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

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

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

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

1.3.4. OPEN and CLOSE

   The NFSv4 protocol introduces OPEN and CLOSE operations. The OPEN
   operation provides a single point where file lookup, creation, and
   share semantics (see Section 9.9) can be combined. The CLOSE
   operation also provides for the release of state accumulated by OPEN.

1.3.5. File Locking

   With the NFSv4 protocol, the support for byte range file locking is
   part of the NFS protocol. The file locking support is structured so
   that an RPC callback mechanism is not required. This is a departure
   from the previous versions of the NFS file locking protocol, Network
   Lock Manager (NLM) [RFC1813]. The state associated with file locks
   is maintained at the server under a lease-based model. The server
   defines a single lease period for all state held by a NFS client. If
   the client does not renew its lease within the defined period, all
   state associated with the client's lease may be released by the
   server. The client may renew its lease with use of the RENEW
   operation or implicitly by use of other operations (primarily READ).

1.3.6. Client Caching and Delegation

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

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

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

The major addition to NFSv4 in the area of caching is the ability of
the server to delegate certain responsibilities to the client. When
the server grants a delegation for a file to a client, the client is
guaranteed certain semantics with respect to the sharing of that file
with other clients. At OPEN, the server may provide the client
either a read (OPEN_DELEGATE_READ) or a write (OPEN_DELEGATE_WRITE)
delegation for the file (see Section 10.4). If the client is granted
a OPEN_DELEGATE_READ delegation, it is assured that no other client
has the ability to write to the file for the duration of the
delegation. If the client is granted a OPEN_DELEGATE_WRITE
delegation, the client is assured that no other client has read or
write access to the file.

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

1.4. General Definitions

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

Anonymous Stateid: Special locking object defined in
   Section 9.1.4.3.

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

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

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

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

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

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

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

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

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

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

Lock-Owner: Each byte-range lock is associated with a specific lock-
   owner and an open-owner. The lock-owner consists of a Client ID
   and an opaque owner string. The client presents this to the
   server to establish the ownership of the byte-range lock as
   needed.

Open-Owner: Each open file is associated with a specific open-owner,
   which consists of a Client ID and an opaque owner string. The
   client presents this to the server to establish the ownership of
   the open as needed.

READ Bypass Stateid: Special locking object defined in
   Section 9.1.4.3.

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

Stable Storage: NFSv4 servers must be able to recover without data
   loss from multiple power failures (including cascading power
   failures, that is, several power failures in quick succession),
   operating system failures, and hardware failure of components

other than the storage medium itself (for example, disk, nonvolatile RAM).

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

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

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

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

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

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

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

1.5. Changes since RFC 3530

The main changes from RFC 3530 [RFC3530] are:

o The XDR definition has been moved to a companion document [RFCNFSv4XDR].

o The IETF intellectual property statements were updated to the latest version.

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

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

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

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

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

o The definition of stateid was strengthened.

1.6. Changes between RFC 3010 and RFC3530

The definition of the NFSv4 protocol in [RFC3530] replaced and
obsoleted the definition present in [RFC3010]. While portions of the
two documents remained the same, there were substantive changes in
others. The changes made between [RFC3010] and [RFC3530] reflect
implementation experience and further review of the protocol.

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

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

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

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

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

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

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

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

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

   o Verify error return possibilities for all operations.

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

2. Protocol Data Types

   The syntax and semantics to describe the data types of the NFS
   version 4 protocol are defined in the XDR [RFC4506] and RPC [RFC5531]
   documents. The next sections build upon the XDR data types to define
   types and structures specific to this protocol. As a reminder, the
   size constants and definitive definitions can be found in
   [RFCNFSv4XDR].

2.1. Basic Data Types

            These are the base NFSv4 data types.

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

```
|                   | Various defined file types.                      |
|  nfsstat4         | enum nfsstat4;                                    |
|                   | Return value for operations.                     |
|  nfs_lease4       | typedef uint32_t nfs_lease4;                     |
|                   | Duration of a lease in seconds.                  |
|  offset4          | typedef uint64_t offset4;                        |
|                   | Various offset designations (READ, WRITE, LOCK,  |
|                   | COMMIT).                                         |
|  qop4             | typedef uint32_t qop4;                           |
|                   | Quality of protection designation in SECINFO.    |
|  sec_oid4         | typedef opaque sec_oid4<>;                        |
|                   | Security Object Identifier. The sec_oid4 data    |
|                   | type is not really opaque. Instead it contains   |
|                   | an ASN.1 OBJECT IDENTIFIER as used by GSS-API    |
|                   | in the mech_type argument to                     |
|                   | GSS_Init_sec_context. See [RFC2743] for          |
|                   | details.                                         |
|  seqid4           | typedef uint32_t seqid4;                         |
|                   | Sequence identifier used for file locking.       |
|  utf8string       | typedef opaque utf8string<>;                     |
|                   | UTF-8 encoding for strings.                      |
|  utf8str_cis      | typedef utf8string utf8str_cis;                  |
|                   | Case insensitive UTF-8 string.                   |
|  utf8str_cs       | typedef utf8string utf8str_cs;                   |
|                   | Case sensitive UTF-8 string.                     |
|  utf8str_mixed    | typedef utf8string utf8str_mixed;                |
|                   | UTF-8 strings with a case sensitive prefix and   |
|                   | a case insensitive suffix.                       |
|  component4       | typedef utf8str_cs component4;                   |
|                   | Represents pathname components.                  |
|  linktext4        | typedef opaque linktext4<>;                      |
|                   | Symbolic link contents ("symbolic link" is       |
|                   | defined in an Open Group [openg_symlink]         |
|                   | standard).                                       |
|  ascii_REQUIRED4  | typedef utf8string ascii_REQUIRED4;              |
|                   | String is sent as ASCII and thus is              |
|                   | automatically UTF-8.                             |
|  pathname4        | typedef component4 pathname4<>;                  |
|                   | Represents path name for fs_locations.           |
|  nfs_lockid4      | typedef uint64_t nfs_lockid4;                     |
|  verifier4        | typedef opaque verifier4[NFS4_VERIFIER_SIZE];    |
|                   | Verifier used for various operations (COMMIT,    |
|                   | CREATE, OPEN, READDIR, WRITE)                     |
|                   | NFS4_VERIFIER_SIZE is defined as 8.              |
+-----------------+--------------------------------------------------+
```

                       End of Base Data Types

                                Table 1

2.2. Structured Data Types

2.2.1. nfstime4

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

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

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

2.2.2. time_how4

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

2.2.3. settime4

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

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

2.2.4. specdata4

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

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

2.2.5. fsid4

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

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

2.2.6. fs_location4

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

2.2.7. fs_locations4

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

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

2.2.8. fattr4

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

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

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

```
                        0           1
   +-----------+-----------+-----------+--
   |   count   | 31  ..  0 | 63  .. 32 |
   +-----------+-----------+-----------+--
```
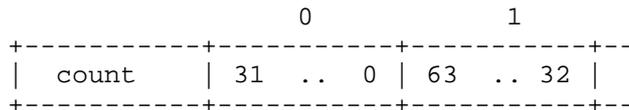
2.2.9. change_info4

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

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

2.2.10. clientaddr4

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

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

2.2.11. cb_client4

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

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

2.2.12. nfs_client_id4

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

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

2.2.13. open_owner4

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

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

2.2.14. lock_owner4

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

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

2.2.15. open_to_lock_owner4

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

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

2.2.16. stateid4

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

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

3. RPC and Security Flavor

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

3.1. Ports and Transports

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

Where an NFSv4 implementation supports operation over the IP network
protocol, the supported transport layer between NFS and IP MUST be an
IETF standardized transport protocol that is specified to avoid
network congestion; such transports include TCP and Stream Control
Transmission Protocol (SCTP). To enhance the possibilities for
interoperability, an NFSv4 implementation MUST support operation over
the TCP transport protocol.

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

As noted in Section 17, the authentication model for NFSv4 has moved
from machine-based to principal-based. However, this modification of
the authentication model does not imply a technical requirement to

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

3.1.1. Client Retransmission Behavior

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

o The transport connection has been broken

o The procedure being retried is the NULL procedure

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

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

3.2. Security Flavors

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

3.2.1. Security mechanisms for NFSv4

   RPCSEC_GSS, via GSS-API, supports multiple mechanisms that provide
   security services. For interoperability, NFSv4 clients and servers
   MUST support the Kerberos V5 security mechanism.

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

3.2.1.1. Kerberos V5 as a Security Triple

   The Kerberos V5 GSS-API mechanism as described in [RFC4121] MUST be
   implemented with the RPCSEC_GSS services as specified in Table 2.
   Both client and server MUST support each of the pseudo flavors.

                   Mapping pseudo flavor to service

   +--------+-------+----------------------+----------------------+
   | Number | Name | Mechanism's OID | RPCSEC_GSS service |
   +--------+-------+----------------------+----------------------+
390003	krb5	1.2.840.113554.1.2.2	rpc_gss_svc_none
390004	krb5i	1.2.840.113554.1.2.2	rpc_gss_svc_integrity
390005	krb5p	1.2.840.113554.1.2.2	rpc_gss_svc_privacy
   +--------+-------+----------------------+----------------------+

                               Table 2

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

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

3.2.1.1.1. Security Considerations for Cryptographic Algorithms in Kerberos V5

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

3.3. Security Negotiation

With the NFSv4 server potentially offering multiple security
mechanisms, the client needs a method to determine or negotiate which
mechanism is to be used for its communication with the server. The
NFS server can have multiple points within its file system name space
that are available for use by NFS clients. In turn the NFS server
can be configured such that each of these entry points can have
different or multiple security mechanisms in use.

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

3.3.1. SECINFO

   The SECINFO operation will allow the client to determine, on a per
   filehandle basis, what security triple (see [RFC2743]) is to be used
   for server access. In general, the client will not have to use the
   SECINFO operation except during initial communication with the server
   or when the client encounters a new security policy as the client
   navigates the name space. Either condition will force the client to
   negotiate a new security triple.

3.3.2. Security Error

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

3.3.3. Callback RPC Authentication

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

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

   AUTH_SYS has no notions of mutual authentication or a server
   principal, so the callback from the server simply uses the AUTH_SYS
   credential that the user used when he set up the delegation.

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

      unix.host@domain

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

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

    service@hostname

For NFS, the "service" element is

    nfs

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

    nfs/hostname

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

4. Filehandles

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

4.1. Obtaining the First Filehandle

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

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

MOUNT protocol is unnecessary for viable interaction between NFS
client and server.

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

4.1.1. Root Filehandle

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

4.1.2. Public Filehandle

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

4.2. Filehandle Types

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

The volatile filehandle type was introduced to address server
functionality or implementation issues which make correct
implementation of a persistent filehandle infeasible. Some server
environments do not provide a file system level invariant that can be
used to construct a persistent filehandle. The underlying server

file system may not provide the invariant or the server's file system
programming interfaces may not provide access to the needed
invariant. Volatile filehandles may ease the implementation of
server functionality such as hierarchical storage management or file
system reorganization or migration. However, the volatile filehandle
increases the implementation burden for the client.

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

4.2.1. General Properties of a Filehandle

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

As an example, in the case that two different path names when
traversed at the server terminate at the same file system object, the
server SHOULD return the same filehandle for each path. This can
occur if a hard link is used to create two file names which refer to
the same underlying file object and associated data. For example, if
paths /a/b/c and /a/d/c refer to the same file, the server SHOULD
return the same filehandle for both path names traversals.

4.2.2. Persistent Filehandle

A persistent filehandle is defined as having a fixed value for the
lifetime of the file system object to which it refers. Once the
server creates the filehandle for a file system object, the server
MUST accept the same filehandle for the object for the lifetime of
the object. If the server restarts or reboots the NFS server must

honor the same filehandle value as it did in the server's previous
instantiation. Similarly, if the file system is migrated, the new
NFS server must honor the same filehandle as the old NFS server.

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

4.2.3. Volatile Filehandle

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

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

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

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

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

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

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

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

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

4.2.4. One Method of Constructing a Volatile Filehandle

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

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

o slot is an index in the server volatile filehandle table

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

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

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

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

4.3. Client Recovery from Filehandle Expiration

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

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

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

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

      RENAME A B
      LOOKUP B
      GETFH

   Note that the COMPOUND procedure does not provide atomicity. This
   example only reduces the overhead of recovering from an expired
   filehandle.

5. Attributes

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

server supports and construct requests with only those supported
attributes (or a subset thereof).

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

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

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

Named attributes are intended for data needed by applications rather
than by an NFS client implementation. NFS implementers are strongly
encouraged to define their new attributes as RECOMMENDED attributes
by bringing them to the IETF Standards Track process.

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

Note that the hidden directory returned by OPENATTR is a convenience
for protocol processing. The client should not make any assumptions

about the server's implementation of named attributes and whether or
not the underlying file system at the server has a named attribute
directory. Therefore, operations such as SETATTR and GETATTR on the
named attribute directory are undefined.

5.1. REQUIRED Attributes

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

5.2. RECOMMENDED Attributes

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

5.3. Named Attributes

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

READDIR may be used to get a list of such named attributes, and
LOOKUP and OPEN may select a particular attribute. Creation of a new
named attribute may be the result of an OPEN specifying file
creation.

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

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

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

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

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

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

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

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

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

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

5.4. Classification of Attributes

Each of attributes accessed using SETATTR and GETATTR (i.e., REQUIRED
an RECOMMENDED attributes) can be classified in one of three
categories:

1. per server attributes for which the value of the attribute will
    be the same for all file objects that share the same server.

2. per file system attributes for which the value of the attribute
    will be the same for some or all file objects that share the same
    server and fsid attribute (Section 5.8.1.9). See below for
    details regarding when such sharing is in effect.

3. per file system object attributes

The handling of per file system attributes depends on the particular
attribute and the setting of the homogeneous (Section 5.8.2.12)
attribute. The following rules apply:

1. The values of the attribute supported_attrs, fsid, homogeneous,
    link_support, and symlink_support are always common to all object
    within the given file system.

2. For other attributes, different values may be returned for
    different file system objects if the attribute homogeneous is
    supported within the file system in question and has the value
    false.

The classification of attributes is as follows. Note that the
attributes time_access_set and time_modify_set are not listed in this
section because they are write-only attributes corresponding to
time_access and time_modify, and are used in a special instance of
SETATTR.

o The per-server attribute is:

            lease_time

    o The per-file system attributes are:

            supported_attrs, fh_expire_type, link_support, symlink_support,
            unique_handles, aclsupport, cansettime, case_insensitive,
            case_preserving, chown_restricted, files_avail, files_free,
            files_total, fs_locations, homogeneous, maxfilesize, maxname,
            maxread, maxwrite, no_trunc, space_avail, space_free,
            space_total, time_delta,

    o The per-file system object attributes are:

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

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

5.5. Set-Only and Get-Only Attributes

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

5.6. REQUIRED Attributes - List and Definition References

    The list of REQUIRED attributes appears in Table 3. The meaning of
    the columns of the table are:

    o Name: The name of attribute

    o Id: The number assigned to the attribute. In the event of
       conflicts between the assigned number and [RFCNFSv4XDR], the
       latter is authoritative, but in such an event, it should be
       resolved with Errata to this document and/or [RFCNFSv4XDR]. See
       [IESG_ERRATA] for the Errata process.

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

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

may set, GETATTR may not retrieve). R W means read/write (GETATTR
may retrieve, SETATTR may set).

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

REQUIRED attributes

Name	Id	Data Type	Acc	Defined in:
supported_attrs	0	bitmap4	R	Section 5.8.1.1
type	1	nfs_ftype4	R	Section 5.8.1.2
fh_expire_type	2	uint32_t	R	Section 5.8.1.3
change	3	changeid4	R	Section 5.8.1.4
size	4	uint64_t	R W	Section 5.8.1.5
link_support	5	bool	R	Section 5.8.1.6
symlink_support	6	bool	R	Section 5.8.1.7
named_attr	7	bool	R	Section 5.8.1.8
fsid	8	fsid4	R	Section 5.8.1.9
unique_handles	9	bool	R	Section 5.8.1.10
lease_time	10	nfs_lease4	R	Section 5.8.1.11
rdattr_error	11	nfsstat4	R	Section 5.8.1.12
filehandle	19	nfs_fh4	R	Section 5.8.1.13

Table 3

5.7. RECOMMENDED Attributes - List and Definition References

The RECOMMENDED attributes are defined in Table 4. The meanings of
the column headers are the same as Table 3; see Section 5.6 for the
meanings.

RECOMMENDED attributes

Name	Id	Data Type	Acc	Defined in:
acl	12	nfsace4<>	R W	Section 6.2.1
aclsupport	13	uint32_t	R	Section 6.2.1.2
archive	14	bool	R W	Section 5.8.2.1
cansettime	15	bool	R	Section 5.8.2.2
case_insensitive	16	bool	R	Section 5.8.2.3
case_preserving	17	bool	R	Section 5.8.2.4
chown_restricted	18	bool	R	Section 5.8.2.5
fileid	20	uint64_t	R	Section 5.8.2.6
files_avail	21	uint64_t	R	Section 5.8.2.7
files_free	22	uint64_t	R	Section 5.8.2.8
files_total	23	uint64_t	R	Section 5.8.2.9
fs_locations	24	fs_locations4	R	Section 5.8.2.10
hidden	25	bool	R W	Section 5.8.2.11
homogeneous	26	bool	R	Section 5.8.2.12
maxfilesize	27	uint64_t	R	Section 5.8.2.13
maxlink	28	uint32_t	R	Section 5.8.2.14
maxname	29	uint32_t	R	Section 5.8.2.15
maxread	30	uint64_t	R	Section 5.8.2.16
maxwrite	31	uint64_t	R	Section 5.8.2.17
mimetype	32	ascii_	R W	Section 5.8.2.18
		REQUIRED4<>		
mode	33	mode4	R W	Section 6.2.2
mounted_on_fileid	55	uint64_t	R	Section 5.8.2.19
no_trunc	34	bool	R	Section 5.8.2.20
numlinks	35	uint32_t	R	Section 5.8.2.21
owner	36	utf8str_mixed	R W	Section 5.8.2.22
owner_group	37	utf8str_mixed	R W	Section 5.8.2.23
quota_avail_hard	38	uint64_t	R	Section 5.8.2.24
quota_avail_soft	39	uint64_t	R	Section 5.8.2.25
quota_used	40	uint64_t	R	Section 5.8.2.26
rawdev	41	specdata4	R	Section 5.8.2.27
space_avail	42	uint64_t	R	Section 5.8.2.28
space_free	43	uint64_t	R	Section 5.8.2.29
space_total	44	uint64_t	R	Section 5.8.2.30
space_used	45	uint64_t	R	Section 5.8.2.31
system	46	bool	R W	Section 5.8.2.32
time_access	47	nfstime4	R	Section 5.8.2.33
time_access_set	48	settime4	W	Section 5.8.2.34
time_backup	49	nfstime4	R W	Section 5.8.2.35
time_create	50	nfstime4	R W	Section 5.8.2.36
time_delta	51	nfstime4	R	Section 5.8.2.37
time_metadata	52	nfstime4	R	Section 5.8.2.38
time_modify	53	nfstime4	R	Section 5.8.2.39
time_modify_set	54	settime4	W	Section 5.8.2.40

                              Table 4

5.8. Attribute Definitions

5.8.1. Definitions of REQUIRED Attributes

5.8.1.1. Attribute 0: supported_attrs

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

5.8.1.2. Attribute 1: type

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

   o NF4REG designates a regular file.

   o NF4DIR designates a directory.

   o NF4BLK designates a block device special file.

   o NF4CHR designates a character device special file.

   o NF4LNK designates a symbolic link.

   o NF4SOCK designates a named socket special file.

   o NF4FIFO designates a fifo special file.

   o NF4ATTRDIR designates a named attribute directory.

   o NF4NAMEDATTR designates a named attribute.

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

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

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

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

   o The phrase "is a symbolic link" means that the object's type
      attribute is NF4LNK.

5.8.1.3. Attribute 2: fh_expire_type

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

5.8.1.4. Attribute 3: change

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

5.8.1.5. Attribute 4: size

   The size of the object in bytes.

5.8.1.6. Attribute 5: link_support

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

5.8.1.7. Attribute 6: symlink_support

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

5.8.1.8. Attribute 7: named_attr

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

5.8.1.9. Attribute 8: fsid

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

5.8.1.10. Attribute 9: unique_handles

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

5.8.1.11. Attribute 10: lease_time

   Duration of the lease at server in seconds.

5.8.1.12. Attribute 11: rdattr_error

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

5.8.1.13. Attribute 19: filehandle

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

5.8.2. Definitions of Uncategorized RECOMMENDED Attributes

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

5.8.2.1. Attribute 14: archive

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

5.8.2.2. Attribute 15: cansettime

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

5.8.2.3. Attribute 16: case_insensitive

   TRUE, if file name comparisons on this file system are case
   insensitive. This refers only to comparisons, and not to the case in
   which file names are stored.

5.8.2.4. Attribute 17: case_preserving

   TRUE, if file name case on this file system is preserved. This
   refers only to how file names are stored, and not to how they are
   compared. File names stored in mixed case might be compared using
   either case-insensitive or case-sensitive comparisons.

5.8.2.5. Attribute 18: chown_restricted

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

5.8.2.6. Attribute 20: fileid

   A number uniquely identifying the file within the file system.

5.8.2.7. Attribute 21: files_avail

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

5.8.2.8. Attribute 22: files_free

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

5.8.2.9. Attribute 23: files_total

   Total file slots on the file system containing this object.

5.8.2.10. Attribute 24: fs_locations

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

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

5.8.2.11. Attribute 25: hidden

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

5.8.2.12. Attribute 26: homogeneous

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

5.8.2.13. Attribute 27: maxfilesize

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

5.8.2.14. Attribute 28: maxlink

   Maximum number of hard links for this object.

5.8.2.15. Attribute 29: maxname

   Maximum file name size supported for this object.

5.8.2.16. Attribute 30: maxread

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

5.8.2.17. Attribute 31: maxwrite

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

5.8.2.18. Attribute 32: mimetype

   MIME media type/subtype of this object.

5.8.2.19. Attribute 55: mounted_on_fileid

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

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

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

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

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

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

5.8.2.20. Attribute 34: no_trunc

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

5.8.2.21. Attribute 35: numlinks

Number of hard links to this object.

5.8.2.22. Attribute 36: owner

The string name of the owner of this object.

5.8.2.23. Attribute 37: owner_group

The string name of the group ownership of this object.

5.8.2.24. Attribute 38: quota_avail_hard

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

5.8.2.25. Attribute 39: quota_avail_soft

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

5.8.2.26. Attribute 40: quota_used

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

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

5.8.2.27. Attribute 41: rawdev

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

5.8.2.28. Attribute 42: space_avail

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

5.8.2.29. Attribute 43: space_free

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

5.8.2.30. Attribute 44: space_total

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

5.8.2.31. Attribute 45: space_used

   Number of file system bytes allocated to this object.

5.8.2.32. Attribute 46: system

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

5.8.2.33. Attribute 47: time_access

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

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

5.8.2.34. Attribute 48: time_access_set

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

5.8.2.35. Attribute 49: time_backup

   The time of last backup of the object.

5.8.2.36. Attribute 50: time_create

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

5.8.2.37. Attribute 51: time_delta

   Smallest useful server time granularity.

5.8.2.38. Attribute 52: time_metadata

   The time of last metadata modification of the object.

5.8.2.39. Attribute 53: time_modify

   The time of last modification to the object.

5.8.2.40. Attribute 54: time_modify_set

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

5.9. Interpreting owner and owner_group

   The RECOMMENDED attributes "owner" and "owner_group" (and also users
   and groups used as values of the "who" field within nfs4ace
   structures used in the acl attribute) are represented in the form of
   UTF-8 strings. This format avoids use of a representation that is
   tied to a particular underlying implementation at the client or
   server. Note that section 6.1 of [RFC2624] provides additional
   rationale. It is expected that the client and server will have their
   own local representation of owners and groups that is used for local
   storage or presentation to the application via API's that expect such
   a representation. Therefore, the protocol requires that when these
   attributes are transferred between the client and server, the local
   representation is translated to a string of the form
   "identifier@dns_domain". This allows clients and servers that do not
   use the same local representation to effectively interoperate since
   they both use a common syntax that can be interpreted by both.

   Similarly, security principals may be represented in different ways
   by different security mechanisms. Servers normally translate these
   representations into a common format, generally that used by local
   storage, to serve as a means of identifying the users corresponding

to these security principals. When these local identifiers are
translated to the form of the owner attribute, associated with files
created by such principals, they identify, in a common format, the
users associated with each corresponding set of security principals.

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

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

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

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

default, the current DNS domain name of the server should be the
value used for the "dns_domain".

As mentioned above, it is desirable that a server when accepting a
string of the form "user@domain" or "group@domain" in an attribute,
return this same string when that corresponding attribute is fetched.
Internationalization issues make this impossible under certain
circumstances and the client needs to take note of these. See
Section 12 for a detailed discussion of these issues.

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

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

A server SHOULD reject such a numeric value if the security mechanism
is using Kerberos. I.e., in such a scenario, the client will already
need to form "user@domain" strings. For any other security
mechanism, the server SHOULD accept such numeric values. As an
implementation note, the server could make such an acceptance be
configurable. If the server does not support numeric values or if it
is configured off, then it MUST return an NFS4ERR_BADOWNER error. If
the security mechanism is using Kerberos and the client attempts to
use the special form, then the server SHOULD return an
NFS4ERR_BADOWNER error when there is a valid translation for the user
or owner designated in this way. In that case, the client must use
the appropriate user@domain string and not the special form for
compatibility.

The client MUST always accept numeric values if the security
mechanism is not RPCSEC_GSS. A client can determine if a server
supports numeric identifiers by first attempting to provide a numeric
identifier. If this attempt rejected with an NFS4ERR_BADOWNER error,

then the client should only use named identifiers of the form
"user@dns_domain".

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

5.10. Character Case Attributes

With respect to the case_insensitive and case_preserving attributes,
case insensitive comparisons of Unicode characters SHOULD use Unicode
Default Case Folding as defined in Chapter 3 of the Unicode Standard
[UNICODE], and MAY override that behavior for specific selected
characters with the case folding defined in the SpecialCasing.txt
[SPECIALCASING] file in section 3.13 of the Unicode Standard.

The SpecialCasing.txt file replaces the Default Case Folding with
locale and context-dependent case folding for specific situations.
An example of locale and context-dependent case folding is that LATIN
CAPITAL LETTER I ("I", U+0049) is default case folded to LATIN SMALL
LETTER I ("i", U+0069); however, several languages (e.g. Turkish)
treat an "I" character with a dot as a different letter than an "I"
character without a dot, therefore in such languages, unless an I is
before a dot_above, the "I" (U+0049) character should be case folded
to a different character, LATIN SMALL LETTER DOTLESS I (U+0131).

The [UNICODE] and [SPECIALCASING] references in this RFC are for
version 6.3.0 of the Unicode standard, as that was the latest version
of Unicode when this RFC was published. Implementations SHOULD
always use the latest version of Unicode (http://www.unicode.org/
versions/latest/).

[RFC Editor: please check that 6.3.0 is the latest version before
publication of this document as an RFC.]

6. Access Control Attributes

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

6.1. Goals

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

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

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

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

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

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

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

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

6.2. File Attributes Discussion

Support for each of the ACL attributes is RECOMMENDED and not
required, since file systems accessed using NFSV4 might not support
ACL's.

6.2.1. Attribute 12: acl

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

The NFS ACE structure is defined as follows:

```
typedef uint32_t        acetype4;

typedef uint32_t        aceflag4;

typedef uint32_t        acemask4;

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

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

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

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

permissions. (An ACL that explicitly denies permission to read
attributes should still result in a denial.)

The situation is complicated by the fact that a server may have
multiple modules that enforce ACLs. For example, the enforcement for
NFSv4.0 access may be different from, but not weaker than, the
enforcement for local access, and both may be different from the
enforcement for access through other protocols such as Server Message
Block (SMB) [MS-SMB]. So it may be useful for a server to accept an
ACL even if not all of its modules are able to support it.

The guiding principle with regard to NFSv4 access is that the server
must not accept ACLs that give an appearance of more restricted
access to a file than what is actually enforced.

6.2.1.1. ACE Type

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

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

All four bit types are permitted in the acl attribute.

Value	Abbreviation	Description
ACE4_ACCESS_ALLOWED_ACE_TYPE	ALLOW	Explicitly grants the access defined in acemask4 to the file or directory.
ACE4_ACCESS_DENIED_ACE_TYPE	DENY	Explicitly denies the access defined in acemask4 to the file or directory.
ACE4_SYSTEM_AUDIT_ACE_TYPE	AUDIT	LOG (in a system dependent way) any access attempt to a file or directory which uses any of the access methods specified in acemask4.
ACE4_SYSTEM_ALARM_ACE_TYPE	ALARM	Generate a system ALARM (system dependent) when any access attempt is made to a file or directory for the access methods specified in acemask4.

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

6.2.1.2. Attribute 13: aclsupport

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

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

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

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

6.2.1.3. ACE Access Mask

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

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

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

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

6.2.1.3.1. Discussion of Mask Attributes

ACE4_READ_DATA

   Operation(s) affected:

      READ

      OPEN

   Discussion:

Permission to read the data of the file.

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

ACE4_LIST_DIRECTORY

Operation(s) affected:

READDIR

Discussion:

Permission to list the contents of a directory.

ACE4_WRITE_DATA

Operation(s) affected:

WRITE

OPEN

SETATTR of size

Discussion:

Permission to modify a file's data.

ACE4_ADD_FILE

Operation(s) affected:

CREATE

LINK

OPEN

RENAME

Discussion:

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

used to create a regular file. LINK and RENAME are always
affected.

ACE4_APPEND_DATA

Operation(s) affected:

WRITE

OPEN

SETATTR of size

Discussion:

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

ACE4_ADD_SUBDIRECTORY

Operation(s) affected:

CREATE

RENAME

Discussion:

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

ACE4_READ_NAMED_ATTRS

Operation(s) affected:

OPENATTR

Discussion:

Permission to read the named attributes of a file or to lookup
the named attributes directory. OPENATTR is affected when it
is not used to create a named attribute directory. This is

when 1.) createdir is TRUE, but a named attribute directory
already exists, or 2.) createdir is FALSE.

ACE4_WRITE_NAMED_ATTRS

   Operation(s) affected:

      OPENATTR

   Discussion:

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

ACE4_EXECUTE

   Operation(s) affected:

      READ

   Discussion:

      Permission to execute a file.

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

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

      nfsuser:ACE4_EXECUTE:ALLOW

A subsequent GETATTR of ACL for that file SHOULD return:

nfsuser:ACE4_EXECUTE:ALLOW

Rather than:

nfsuser:ACE4_EXECUTE/ACE4_READ_DATA:ALLOW

ACE4_EXECUTE

   Operation(s) affected:

      LOOKUP

      OPEN

      REMOVE

      RENAME

      LINK

      CREATE

   Discussion:

      Permission to traverse/search a directory.

ACE4_DELETE_CHILD

   Operation(s) affected:

      REMOVE

      RENAME

   Discussion:

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

ACE4_READ_ATTRIBUTES

   Operation(s) affected:

      GETATTR of file system object attributes

VERIFY

NVERIFY

READDIR

Discussion:

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

ACE4_WRITE_ATTRIBUTES

Operation(s) affected:

SETATTR of time_access_set, time_backup,

time_create, time_modify_set, mimetype, hidden, system

Discussion:

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

ACE4_DELETE

Operation(s) affected:

REMOVE

Discussion:

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

ACE4_READ_ACL

Operation(s) affected:

GETATTR of acl

NVERIFY

VERIFY

Discussion:

Permission to read the ACL.

ACE4_WRITE_ACL

Operation(s) affected:

SETATTR of acl and mode

Discussion:

Permission to write the acl and mode attributes.

ACE4_WRITE_OWNER

Operation(s) affected:

SETATTR of owner and owner_group

Discussion:

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

ACE4_SYNCHRONIZE

Operation(s) affected:

NONE

Discussion:

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

Typically, the ACE4_SYNCHRONIZE permission is only meaningful
on local file systems, i.e., file systems not accessed via
NFSv4.0. The reason that the permission bit exists is that
some operating environments, such as Windows, use
ACE4_SYNCHRONIZE.

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

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

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

6.2.1.3.2. ACE4_DELETE vs. ACE4_DELETE_CHILD

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

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

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

If the ACLs in question neither explicitly ALLOW nor DENY either of
the above, and if MODE4_SVTX is not set on the parent, then the
server SHOULD allow the removal if and only if ACE4_ADD_FILE is

permitted. In the case where MODE4_SVTX is set, the server may also
require the remover to own either the parent or the target, or may
require the target to be writable.

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

6.2.1.4. ACE flag

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

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

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

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

6.2.1.4.1. Discussion of Flag Bits

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

ACE4_DIRECTORY_INHERIT_ACE
   Can be placed on a directory and indicates that this ACE should be
   added to each new directory created.
   If this flag is set in an ACE in an ACL attribute to be set on a
   non-directory file system object, the operation attempting to set
   the ACL SHOULD fail with NFS4ERR_ATTRNOTSUPP.

ACE4_INHERIT_ONLY_ACE

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

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

ACE4_SUCCESSFUL_ACCESS_ACE_FLAG

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

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

ACE4_IDENTIFIER_GROUP
    Indicates that the "who" refers to a GROUP as defined under UNIX
    or a GROUP ACCOUNT as defined under Windows. Clients and servers
    MUST ignore the ACE4_IDENTIFIER_GROUP flag on ACEs with a who
    value equal to one of the special identifiers outlined in
    Section 6.2.1.5.

6.2.1.5. ACE Who

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

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

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

                                Table 5

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

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

6.2.1.5.1. Discussion of EVERYONE@

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

6.2.2. Attribute 33: mode

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

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

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

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

6.3. Common Methods

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

6.3.1. Interpreting an ACL

6.3.1.1. Server Considerations

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

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

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

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

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

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

6.3.1.2. Client Considerations

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

   Clients must be aware of situations in which an object's ACL will
   define a certain access even though the server will not have adequate
   information to enforce it. For example, the server has no way of
   determining whether a particular OPEN reflects a user's open for read
   access, or is done as part of executing the file in question. In
   such situations, the client needs to do its part in the enforcement
   of access as defined by the ACL. To do this, the client will send
   the appropriate ACCESS operation (or use a cached previous
   determination) prior to servicing the request of the user or
   application in order to determine whether the user or application
   should be granted the access requested. For examples in which the
   ACL may define accesses that the server does not enforce see
   Section 6.3.1.1.

6.3.2. Computing a Mode Attribute from an ACL

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

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

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

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

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

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

6.3.2.1. Discussion

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

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

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

6.4. Requirements

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

Many requirements refer to Section 6.3.2, but note that the methods
have behaviors specified with "SHOULD". This is intentional, to

avoid invalidating existing implementations that compute the mode
according to the withdrawn POSIX ACL draft ([P1003.1e]), rather than
by actual permissions on owner, group, and other.

6.4.1. Setting the mode and/or ACL Attributes

6.4.1.1. Setting mode and not ACL

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

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

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

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

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

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

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

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

Also note that the requirement may be met by discarding the acl in
favor of an ACL that represents the mode and only the mode. This is
permitted, but it is preferable for a server to preserve as much of
the ACL as possible without violating the above requirements.

Discarding the ACL makes it effectively impossible for a file created
with a mode attribute to inherit an ACL (see Section 6.4.3).

6.4.1.2. Setting ACL and not mode

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

6.4.1.3. Setting both ACL and mode

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

6.4.2. Retrieving the mode and/or ACL Attributes

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

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

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

6.4.3. Creating New Objects

If a server supports any ACL attributes, it may use the ACL
attributes on the parent directory to compute an initial ACL
attribute for a newly created object. This will be referred to as

the inherited ACL within this section. The act of adding one or more
ACEs to the inherited ACL that are based upon ACEs in the parent
directory's ACL will be referred to as inheriting an ACE within this
section.

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

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

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

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

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

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

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

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

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

    In other cases, inheritance SHOULD take place, and no
    modifications to the ACL will happen. The mode attribute, if
    supported, MUST be as computed in Section 6.3.2, with the

MODE4_SUID, MODE4_SGID and MODE4_SVTX bits clear. If no
inheritable ACEs exist on the parent directory, the rules for
creating acl attributes are implementation defined.

6.4.3.1. The Inherited ACL

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

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

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

7. NFS Server Name Space

7.1. Server Exports

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

7.2. Browsing Exports

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

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

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

7.3. Server Pseudo Filesystem

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

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

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

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

7.4. Multiple Roots

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

7.5. Filehandle Volatility

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

7.6. Exported Root

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

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

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

7.7. Mount Point Crossing

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

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

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

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

It is the server's responsibility to present the pseudo file system that is complete to the client. If the client sends a lookup request for the path "/a/b/c/d", the server's response is the filehandle of the file system "/a/b/c/d". In previous versions of the NFS protocol, the server would respond with the filehandle of directory " /a/b/c/d" within the file system "/a/b".

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

7.8. Security Policy and Name Space Presentation

Because NFSv4 clients possess the ability to change the security mechanisms used, after determining what is allowed, by using SECINFO the server SHOULD NOT present a different view of the namespace based on the security mechanism being used by a client. Instead, it should present a consistent view and return NFS4ERR_WRONGSEC if an attempt is made to access data with an inappropriate security mechanism.

If security considerations make it necessary to hide the existence of a particular file system, as opposed to all of the data within it, the server can apply the security policy of a shared resource in the server's namespace to components of the resource's ancestors. For example:

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

The /a/b/MySecretProject directory is a real file system and is the shared resource. Suppose the security policy for /a/b/ MySecretProject is Kerberos with integrity and it is desired to limit knowledge of the existence of this file system. In this case, the server should apply the same security policy to /a/b. This allows for knowledge of the existence of a file system to be secured when desirable.

For the case of the use of multiple, disjoint security mechanisms in the server's resources, applying that sort of policy would result in the higher-level file system not being accessible using any security flavor. Therefore, that sort of configuration is not compatible with

hiding the existence (as opposed to the contents) from clients using
multiple disjoint sets of security flavors.

In other circumstances, a desirable policy is for the security of a
particular object in the server's namespace to include the union of
all security mechanisms of all direct descendants. A common and
convenient practice, unless strong security requirements dictate
otherwise, is to make the entire pseudo file system accessible by all
of the valid security mechanisms.

Where there is concern about the security of data on the network,
clients should use strong security mechanisms to access the pseudo
file system in order to prevent man-in-the-middle attacks.

8. Multi-Server Namespace

NFSv4 supports attributes that allow a namespace to extend beyond the
boundaries of a single server. It is RECOMMENDED that clients and
servers support construction of such multi-server namespaces. Use of
such multi-server namespaces is optional, however, and for many
purposes, single-server namespaces are perfectly acceptable. Use of
multi-server namespaces can provide many advantages, however, by
separating a file system's logical position in a namespace from the
(possibly changing) logistical and administrative considerations that
result in particular file systems being located on particular
servers.

8.1. Location Attributes

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

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

8.2. File System Presence or Absence

A given location in an NFSv4 namespace (typically but not necessarily
a multi-server namespace) can have a number of file system instance
locations associated with it via the fs_locations attribute. There
may also be an actual current file system at that location,
accessible via normal namespace operations (e.g., LOOKUP). In this

case, the file system is said to be "present" at that position in the
namespace, and clients will typically use it, reserving use of
additional locations specified via the location-related attributes to
situations in which the principal location is no longer available.

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

While the error name suggests that we have a case of a file system
that once was present, and has only become absent later, this is only
one possibility. A position in the namespace may be permanently
absent with the set of file system(s) designated by the location
attributes being the only realization. The name NFS4ERR_MOVED
reflects an earlier, more limited conception of its function, but
this error will be returned whenever the referenced file system is
absent, whether it has moved or simply never existed.

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

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

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

8.3. Getting Attributes for an Absent File System

When a file system is absent, most attributes are not available, but
it is necessary to allow the client access to the small set of
attributes that are available, and most particularly that which gives

information about the correct current locations for this file system, fs_locations.

8.3.1. GETATTR Within an Absent File System

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

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

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

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

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

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

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

8.3.2. READDIR and Absent File Systems

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

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

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

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

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

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

8.4. Uses of Location Information

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

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

fact present, and the replicas are instead different paths to the
same data.

When a file system is present and subsequently becomes absent,
clients can be given the opportunity to have continued access to
their data, at an alternative location. Transfer of the file system
contents to the new location is referred to as "migration". See
Section 8.4.2 for details.

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

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

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

8.4.1. File System Replication

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

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

   Multiple server addresses, whether they are derived from a single
   entry with a DNS name representing a set of IP addresses or from

multiple entries each with its own server address, may correspond to
the same actual server.

8.4.2. File System Migration

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

Such migration can be helpful in providing load balancing or general
resource reallocation. The protocol does not specify how the file
system will be moved between servers. It is anticipated that a
number of different server-to-server transfer mechanisms might be
used with the choice left to the server implementer. The NFSv4
protocol specifies the method used to communicate the migration event
between client and server.

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

8.4.3. Referrals

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

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

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

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

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

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

8.5. Location Entries and Server Identity

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

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

If a single location entry designates multiple server IP addresses,
the client should choose a single one to use. When two server
addresses are designated by a single location entry and they
correspond to different servers, this normally indicates some sort of
misconfiguration, and so the client should avoid using such location
entries when alternatives are available. When they are not, clients
should pick one of IP addresses and use it, without using others that
are not directed to the same server.

8.6. Additional Client-Side Considerations

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

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

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

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

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

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

8.7. Effecting File System Referrals

   Referrals are effected when an absent file system is encountered, and
   one or more alternative locations are made available by the
   fs_locations attribute. The client will typically get an
   NFS4ERR_MOVED error, fetch the appropriate location information, and
   proceed to access the file system on a different server, even though
   it retains its logical position within the original namespace.
   Referrals differ from migration events in that they happen only when
   the client has not previously referenced the file system in question
   (so there is nothing to transition). Referrals can only come into
   effect when an absent file system is encountered at its root.

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

8.7.1. Referral Example (LOOKUP)

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

   o PUTROOTFH

   o LOOKUP "this"

   o LOOKUP "is"

   o LOOKUP "the"

   o LOOKUP "path"

   o GETFH

   o GETATTR(fsid,fileid,size,time_modify)

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

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

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

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

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

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

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

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

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

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

    OP01: PUTROOTFH --> NFS_OK

    - Current fh is root of pseudo-fs.

    OP02: GETATTR(fsid) --> NFS_OK

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

    OP03: LOOKUP "this" --> NFS_OK

    OP04: GETATTR(fsid) --> NFS_OK

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

    OP05: GETFH --> NFS_OK

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

    OP06: LOOKUP "is" --> NFS_OK

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

    OP07: GETATTR(fsid) --> NFS_OK

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

    OP08: GETFH --> NFS_OK

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

    OP09: LOOKUP "the" --> NFS_OK

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

    OP10: GETATTR(fsid) --> NFS_OK

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

    OP11: GETFH --> NFS_OK

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

OP12: LOOKUP "path" --> NFS_OK

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

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

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

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

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

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

OP14: GETFH --> NFS4ERR_MOVED

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

Given the above, the client knows where the root of the absent file
system is (/this/is/the/path) by noting where the change of fsid
occurred (between "the" and "path"). The fs_locations attribute also
gives the client the actual location of the absent file system, so

that the referral can proceed. The server gives the client the bare
minimum of information about the absent file system so that there
will be very little scope for problems of conflict between
information sent by the referring server and information of the file
system's home. No filehandles and very few attributes are present on
the referring server, and the client can treat those it receives as
transient information with the function of enabling the referral.

8.7.2. Referral Example (READDIR)

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

   Suppose such a directory is read as follows:

   o PUTROOTFH

   o LOOKUP "this"

   o LOOKUP "is"

   o LOOKUP "the"

   o READDIR (fsid, size, time_modify, mounted_on_fileid)

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

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

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

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

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

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

So now suppose that we re-send with rdattr_error:

o PUTROOTFH

o LOOKUP "this"

o LOOKUP "is"

o LOOKUP "the"

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

The results will be:

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

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

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

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

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

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

o PUTROOTFH

o LOOKUP "this"

o LOOKUP "is"

o LOOKUP "the"

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

The results would be:

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

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

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

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

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

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

   o rdattr_error (value: NFS_OK)

   o fs_locations

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

   o fsid (value: unique value within referring server)

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

8.8. The Attribute fs_locations

   The fs_locations attribute is defined by both fs_location4
   (Section 2.2.6) and fs_locations4 (Section 2.2.7). It is used to
   represent the location of a file system by providing a server name
   and the path to the root of the file system within that server's
   namespace. When a set of servers have corresponding file systems at
   the same path within their namespaces, an array of server names may
   be provided. An entry in the server array is a UTF-8 string and
   represents one of a traditional DNS host name, IPv4 address, IPv6
   address, or a zero-length string. A zero-length string SHOULD be
   used to indicate the current address being used for the RPC call. It
   is not a requirement that all servers that share the same rootpath be
   listed in one fs_location4 instance. The array of server names is
   provided for convenience. Servers that share the same rootpath may

also be listed in separate fs_location4 entries in the fs_locations
attribute.

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

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

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

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

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

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

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

9. File Locking and Share Reservations

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

o clear division between client and server

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

o simple and robust recovery mechanisms

In this model, the server owns the state information. The client
requests changes in locks and the server responds with the changes
made. Non-client-initiated changes in locking state are infrequent.
The client receives prompt notification of such changes and can
adjust its view of the locking state to reflect the server's changes.

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

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

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

9.1. Opens and Byte-Range Locks

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

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

9.1.1. Client ID

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

CLAIM_DELEGATE_PREV claim type, all delegation state associated with same client with the same identity. For discussion of delegation state recovery, see Section 10.2.1.

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

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

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

Client identification is encapsulated in the following structure:

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

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

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

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

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

o The string should be selected so the subsequent incarnations (e.g., reboots) of the same client cause the client to present the same string. The implementer is cautioned against an approach

that requires the string to be recorded in a local file because
this precludes the use of the implementation in an environment
where there is no local disk and all file access is from an NFSv4
server.

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

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

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

o The server's network address.

o The client's network address.

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

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

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

   * A MAC address (for privacy reasons, it is best to perform some
      one way function on the MAC address).

   * The timestamp of when the NFSv4 software was first installed on
      the client (though this is subject to the previously mentioned

caution about using information that is stored in a file, because the file might only be accessible over NFSv4).

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

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

Note that SETCLIENTID (Section 15.35) and SETCLIENTID_CONFIRM (Section 15.36) have a secondary purpose of establishing the information the server needs to make callbacks to the client for the purpose of supporting delegations. It is permitted to change this information via SETCLIENTID and SETCLIENTID_CONFIRM within the same incarnation of the client without removing the client's leased state.

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

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

See the detailed descriptions of SETCLIENTID (Section 15.35.4) and SETCLIENTID_CONFIRM (Section 15.36.4) for a complete specification of the operations.

9.1.2. Server Release of Client ID

If the server determines that the client holds no associated state for its client ID, the server may choose to release the client ID. The server may make this choice for an inactive client so that

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

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

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

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

9.1.3. Use of Seqids

In several contexts, 32-bit sequence values, called "seqids" are used
as part of managing locking state. Such values are used:

o To provide an ordering of locking-related operations associated
   with a particular lock-owner or open-owner. See Section 9.1.7 for
   a detailed explanation.

o To define an ordered set of instances of a set of locks sharing a
   particular set of ownership characteristics. See Section 9.1.4.2
   for a detailed explanation.

Successive seqid values for the same object are normally arrived at
by incrementing the current value by one. This pattern continues
until the seqid is incremented past NFS4_UINT32_MAX, in which case
one (rather than zero) is to be the next seqid value.

When two seqid values are to be compared to determine which of the two is later, the possibility of wraparound needs to be considered. In many cases, the values are such that simple numeric comparisons can be used. For example, if the seqid values to be compared are both less than one million, the higher value can be considered the later. On the other hand, if one of the values is at or near NFS_UINT32_MAX and the other is less than one million, then implementations can reasonably decide that the lower value has had one more wraparound and is thus, while numerically lower, actually later.

Implementations can compare seqids in the presence of potential wraparound by adopting the reasonable assumption that the chain of increments from one to the other is shorter than 2**31. So, if the difference between the two seqids is less than 2**31, then the lower seqid is to be treated as earlier. If, however, the difference between the two seqids is greater than or equal to 2**31, then it can be assumed that the lower seqid has encountered one more wraparound and can be treated as later.

9.1.4. Stateid Definition

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

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

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

9.1.4.1. Stateid Types

   With the exception of special stateids (see Section 9.1.4.3), each
   stateid represents locking objects of one of a set of types defined
   by the NFSv4 protocol. Note that in all these cases, where we speak
   of a guarantee, it is understood there are situations such as a
   client restart, or lock revocation, that allow the guarantee to be
   voided.

   o Stateids may represent opens of files.

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

   o Stateids may represent sets of byte-range locks.

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

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

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

9.1.4.2. Stateid Structure

   Stateids are divided into two fields, a 96-bit "other" field
   identifying the specific set of locks and a 32-bit "seqid" sequence
   value. Except in the case of special stateids (see Section 9.1.4.3),
   a particular value of the "other" field denotes a set of locks of the
   same type (for example, byte-range locks, opens, or delegations), for
   a specific file or directory, and sharing the same ownership
   characteristics. The seqid designates a specific instance of such a
   set of locks, and is incremented to indicate changes in such a set of
   locks, either by the addition or deletion of locks from the set, a
   change in the byte-range they apply to, or an upgrade or downgrade in
   the type of one or more locks.

   When such a set of locks is first created, the server returns a
   stateid with seqid value of one. On subsequent operations that

modify the set of locks, the server is required to advance the
"seqid" field by one whenever it returns a stateid for the same
state-owner/file/type combination and the operation is one that might
make some change in the set of locks actually designated. In this
case, the server will return a stateid with an "other" field the same
as previously used for that state-owner/file/type combination, with
an incremented "seqid" field.

Seqids will be compared, by both the client and the server. The
client uses such comparisons to determine the order of operations
while the server uses them to determine whether the
NFS4ERR_OLD_STATEID error is to be returned. In all cases, the
possibility of seqid wraparound needs to be taken into account, as
discussed in Section 9.1.3

9.1.4.3. Special Stateids

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

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

Anonymous Stateid: When "other" and "seqid" are both zero, the
   stateid is treated as a special anonymous stateid, which can be
   used in READ, WRITE, and SETATTR requests to indicate the absence
   of any open state associated with the request. When an anonymous
   stateid value is used, and an existing open denies the form of
   access requested, then access will be denied to the request.

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

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

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

9.1.4.4. Stateid Lifetime and Validation

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

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

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

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

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

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

   And then store in each table entry,

   o The client ID with which the stateid is associated.

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

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

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

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

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

   With this information, an incoming stateid can be validated and the
   appropriate error returned when necessary. Special and non-special
   stateids are handled separately. (See Section 9.1.4.3 for a
   discussion of special stateids.)

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

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

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

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

   When a stateid is being tested, and the "other" field is neither all
   zeros or all ones, the following procedure could be used to validate
   an incoming stateid and return an appropriate error, when necessary,
   assuming that the "other" field would be divided into a table index
   and an entry generation. Note that the terms "earlier" and "later"
   used in connection with seqid comparison are to be understood as
   explained in Section 9.1.3.

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

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

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

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

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

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

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

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

9.1.4.5. Stateid Use for I/O Operations

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

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

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

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

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

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

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

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

9.1.4.6. Stateid Use for SETATTR Operations

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

9.1.5. lock-owner

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

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

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

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

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

9.1.6. Use of the Stateid and Locking

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

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

   Share reservations are established by OPEN operations and by their
   nature are mandatory in that when the OPEN denies READ or WRITE
   operations, that denial results in such operations being rejected
   with error NFS4ERR_LOCKED. Byte-range locks may be implemented by
   the server as either mandatory or advisory, or the choice of
   mandatory or advisory behavior may be determined by the server on the
   basis of the file being accessed (for example, some UNIX-based
   servers support a "mandatory lock bit" on the mode attribute such
   that if set, byte-range locks are required on the file before I/O is
   possible). When byte-range locks are advisory, they only prevent the
   granting of conflicting lock requests and have no effect on READs or
   WRITEs. Mandatory byte-range locks, however, prevent conflicting I/O
   operations. When they are attempted, they are rejected with
   NFS4ERR_LOCKED. When the client gets NFS4ERR_LOCKED on a file it
   knows it has the proper share reservation for, it will need to issue
   a LOCK request on the region of the file that includes the region the
   I/O was to be performed on, with an appropriate locktype (i.e.,
   READ*_LT for a READ operation, WRITE*_LT for a WRITE operation).

   With NFSv3, there was no notion of a stateid so there was no way to
   tell if the application process of the client sending the READ or
   WRITE operation had also acquired the appropriate byte-range lock on

the file. Thus there was no way to implement mandatory locking.
With the stateid construct, this barrier has been removed.

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

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

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

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

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

A READ bypass stateid MAY allow READ operations to bypass locking
checks at the server. However, WRITE operations with a READ bypass
stateid MUST NOT bypass locking checks and are treated exactly the
same as if an anonymous stateid were used.

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

9.1.7. Sequencing of Lock Requests

Locking is different than most NFS operations as it requires "at-
most-one" semantics that are not provided by ONC RPC. ONC RPC over a
reliable transport is not sufficient because a sequence of locking
requests may span multiple TCP connections. In the face of
retransmission or reordering, lock or unlock requests must have a
well defined and consistent behavior. To accomplish this, each lock
request contains a sequence number that is a consecutively increasing
integer. Different state-owners have different sequences. The
server maintains the last sequence number (L) received and the
response that was returned. The server SHOULD assign a seqid value
of one for the first request issued for any given state-owner.
Subsequent values are arrived at by incrementing the seqid value,
subject to wraparound as described in Section 9.1.3.

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

When a request is received, its sequence number (r) is compared to
that of the last one received (L). Only if it has the correct next
sequence, normally L + 1, is the request processed beyond the point
of seqid checking. Given a properly-functioning client, the response
to (r) must have been received before the last request (L) was sent.
If a duplicate of last request (r == L) is received, the stored
response is returned. If the sequence value received is any other
value, it is rejected with the return of error NFS4ERR_BAD_SEQID.
Sequence history is reinitialized whenever the SETCLIENTID/
SETCLIENTID_CONFIRM sequence changes the client verifier.

It is critical the server maintain the last response sent to the
client to provide a more reliable cache of duplicate non-idempotent
requests than that of the traditional cache described in [Chet]. The
traditional duplicate request cache uses a least recently used

algorithm for removing unneeded requests. However, the last lock
request and response on a given state-owner must be cached as long as
the lock state exists on the server.

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

9.1.8. Recovery from Replayed Requests

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

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

9.1.9. Interactions of multiple sequence values

Some Operations may have multiple sources of data for request
sequence checking and retransmission determination. Some Operations
have multiple sequence values associated with multiple types of
state-owners. In addition, such Operations may also have a stateid
with its own seqid value, that will be checked for validity.

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

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

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

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

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

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

9.1.10. Releasing state-owner State

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

   If the server were able to be sure that a given state-owner would
   never again be used by a client, such an issue could not arise. Even
   when the state-owner state is released and the client subsequently
   uses that state-owner, retransmitted requests will be detected as
   invalid and the request not executed, although the client may have a
   recovery path that is more complicated than simply getting the
   original response back transparently.

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

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

9.1.11. Use of Open Confirmation

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

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

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

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

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

Delegations being sent with an open requiring confirmation are
troublesome because recovering from non-confirmation adds undue
complexity to the protocol while requiring confirmation on reclaim-
type opens poses difficulties in that the inability to resolve the
status of the reclaim until lease expiration may make it difficult to

have timely determination of the set of locks being reclaimed (since
the grace period may expire).

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

9.2. Lock Ranges

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

The client is discouraged from combining multiple independent locking
ranges that happen to be adjacent into a single request since the
server may not support sub-range requests and for reasons related to
the recovery of file locking state in the event of server failure.
As discussed in the Section 9.6.2 below, the server may employ
certain optimizations during recovery that work effectively only when
the client's behavior during lock recovery is similar to the client's
locking behavior prior to server failure.

9.3. Upgrading and Downgrading Locks

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

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

9.4. Blocking Locks

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

Servers may also note the lock types and delay returning denial of
the request to allow extra time for a conflicting lock to be
released, allowing a successful return. In this way, clients can
avoid the burden of needlessly frequent polling for blocking locks.
The server should take care in the length of delay in the event the
client retransmits the request.

If a server receives a blocking lock request, denies it, and then
later receives a nonblocking request for the same lock, which is also
denied, then it should remove the lock in question from its list of
pending blocking locks. Clients should use such a nonblocking
request to indicate to the server that this is the last time they
intend to poll for the lock, as may happen when the process
requesting the lock is interrupted. This is a courtesy to the
server, to prevent it from unnecessarily waiting a lease period

before granting other lock requests. However, clients are not
required to perform this courtesy, and servers must not depend on
them doing so. Also, clients must be prepared for the possibility
that this final locking request will be accepted.

9.5. Lease Renewal

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

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

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

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

This approach allows for low overhead lease renewal which scales
well. In the typical case no extra RPC calls are required for lease
renewal and in the worst case one RPC is required every lease period
(i.e., a RENEW operation). The number of locks held by the client is
not a factor since all state for the client is involved with the
lease renewal action.

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

9.6. Crash Recovery

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

9.6.1. Client Failure and Recovery

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

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

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

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

9.6.2. Server Failure and Recovery

   If the server loses locking state (usually as a result of a restart
   or reboot), it must allow clients time to discover this fact and re-
   establish the lost locking state. The client must be able to re-
   establish the locking state without having the server deny valid
   requests because the server has granted conflicting access to another
   client. Likewise, if there is the possibility that clients have not

yet re-established their locking state for a file, the server must
disallow READ and WRITE operations for that file. The duration of
this recovery period is equal to the duration of the lease period.

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

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

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

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

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

from stable storage or another similar mechanism, the processing of I
/O requests could proceed normally for the file.

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

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

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

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

9.6.3. Network Partitions and Recovery

If the duration of a network partition is greater than the lease
period provided by the server, the server will have not received a
lease renewal from the client. If this occurs, the server may cancel
the lease and free all locks held for the client. As a result, all
stateids held by the client will become invalid or stale. Once the
client is able to reach the server after such a network partition,
all I/O submitted by the client with the now invalid stateids will
fail with the server returning the error NFS4ERR_EXPIRED. Once this
error is received, the client will suitably notify the application
that held the lock.

9.6.3.1. Courtesy Locks

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

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

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

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

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

9.6.3.2. Lease Cancellation

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

   When a lease is canceled, all locking state associated with it is
   freed and use of any the associated stateids will result in
   NFS4ERR_EXPIRED being returned. Similarly, use of the associated
   clientid will result in NFS4ERR_EXPIRED being returned.

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

9.6.3.3. Client's Reaction to a Freed Lock

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

NFS4ERR_EXPIRED:

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

NFS4ERR_ADMIN_REVOKED:

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

NFS4ERR_BAD_STATEID:

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

NFS4ERR_RECLAIM_BAD:

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

NFS4ERR_NO_GRACE:

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

NFS4ERR_OLD_STATEID:

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

9.6.3.4. Edge Conditions

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

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

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

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

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

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

9.6.3.4.1. First Server Edge Condition

   The first edge condition has the following scenario:

   1. Client A acquires a lock.

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

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

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

   5. Client B releases the lock

6. Server reboots

7. Network partition between client A and server heals.

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

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

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

9.6.3.4.2. Second Server Edge Condition

The second known edge condition follows:

1.   Client A acquires a lock.

2.   Server reboots.

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

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

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

6.   Client B releases the lock.

7.   Server reboots a second time.

8.   Network partition between client A and server heals.

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

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

As with the first edge condition, the final step of the scenario of
the second edge condition has the server erroneously granting client
A's lock reclaim.

9.6.3.4.3. Handling Server Edge Conditions

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

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

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

   o the client's id string

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

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

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

   Assuming the above record keeping, for the first edge condition,
   after the server reboots, the record that client A's lease expired
   means that another client could have acquired a conflicting record
   lock, share reservation, or delegation. Hence the server must reject
   a reclaim from client A with the error NFS4ERR_NO_GRACE or
   NFS4ERR_RECLAIM_BAD.

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

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

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

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

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

9.6.3.4.4. Client Edge Condition

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

The third known edge condition follows:

1. Client A acquires a lock 1.

2. Client A acquires a lock 2.

3. Server reboots.

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

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

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

7.   Server's reclaim grace period ends.

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

9.   Client B releases the lock.

10. Server reboots a second time.

11. Network partition between client A and server heals.

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

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

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

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

A server could avoid this situation by rejecting the reclaim of lock 2. However, to do so accurately it would have to ensure that additional information about individual locks held survives reboot. Server implementations are not required to do that, so the client must not assume that the server will.

Instead, a client MUST reclaim only those locks which it successfully acquired from the previous server instance, omitting any that it failed to reclaim before a new reboot. Thus, in the last step above, client A should reclaim only lock 1.

9.6.3.4.5. Client's Handling of Reclaim Errors

A mandate for the client's handling of the NFS4ERR_NO_GRACE and NFS4ERR_RECLAIM_BAD errors is outside the scope of this specification, since the strategies for such handling are very dependent on the client's operating environment. However, one potential approach is described below.

When the client's reclaim fails, it could examine the change attribute of the objects the client is trying to reclaim state for,

and use that to determine whether to re-establish the state via
normal OPEN or LOCK requests. This is acceptable provided the
client's operating environment allows it. In other words, the client
implementer is advised to document for his users the behavior. The
client could also inform the application that its byte-range lock or
share reservations (whether they were delegated or not) have been
lost, such as via a UNIX signal, a GUI pop-up window, etc. See
Section 10.5, for a discussion of what the client should do for
dealing with unreclaimed delegations on client state.

For further discussion of revocation of locks see Section 9.8.

9.7. Recovery from a Lock Request Timeout or Abort

In the event a lock request times out, a client may decide to not
retry the request. The client may also abort the request when the
process for which it was issued is terminated (e.g., in UNIX due to a
signal). It is possible though that the server received the request
and acted upon it. This would change the state on the server without
the client being aware of the change. It is paramount that the
client re-synchronize state with server before it attempts any other
operation that takes a seqid and/or a stateid with the same state-
owner. This is straightforward to do without a special re-
synchronize operation.

Since the server maintains the last lock request and response
received on the state-owner, for each state-owner, the client should
cache the last lock request it sent such that the lock request did
not receive a response. From this, the next time the client does a
lock operation for the state-owner, it can send the cached request,
if there is one, and if the request was one that established state
(e.g., a LOCK or OPEN operation), the server will return the cached
result or if never saw the request, perform it. The client can
follow up with a request to remove the state (e.g., a LOCKU or CLOSE
operation). With this approach, the sequencing and stateid
information on the client and server for the given state-owner will
re-synchronize and in turn the lock state will re-synchronize.

9.8. Server Revocation of Locks

At any point, the server can revoke locks held by a client and the
client must be prepared for this event. When the client detects that
its locks have been or may have been revoked, the client is
responsible for validating the state information between itself and
the server. Validating locking state for the client means that it
must verify or reclaim state for each lock currently held.

The first instance of lock revocation is upon server reboot or re-
initialization. In this instance the client will receive an error
(NFS4ERR_STALE_STATEID or NFS4ERR_STALE_CLIENTID) and the client will
proceed with normal crash recovery as described in the previous
section.

The second lock revocation event is the inability to renew the lease
before expiration. While this is considered a rare or unusual event,
the client must be prepared to recover. Both the server and client
will be able to detect the failure to renew the lease and are capable
of recovering without data corruption. For the server, it tracks the
last renewal event serviced for the client and knows when the lease
will expire. Similarly, the client must track operations which will
renew the lease period. Using the time that each such request was
sent and the time that the corresponding reply was received, the
client should bound the time that the corresponding renewal could
have occurred on the server and thus determine if it is possible that
a lease period expiration could have occurred.

The third lock revocation event can occur as a result of
administrative intervention within the lease period. While this is
considered a rare event, it is possible that the server's
administrator has decided to release or revoke a particular lock held
by the client. As a result of revocation, the client will receive an
error of NFS4ERR_ADMIN_REVOKED. In this instance the client may
assume that only the state-owner's locks have been lost. The client
notifies the lock holder appropriately. The client cannot assume the
lease period has been renewed as a result of a failed operation.

When the client determines the lease period may have expired, the
client must mark all locks held for the associated lease as
"unvalidated". This means the client has been unable to re-establish
or confirm the appropriate lock state with the server. As described
in Section 9.6, there are scenarios in which the server may grant
conflicting locks after the lease period has expired for a client.
When it is possible that the lease period has expired, the client
must validate each lock currently held to ensure that a conflicting
lock has not been granted. The client may accomplish this task by
issuing an I/O request; if there is no relevant I/O pending, a zero-
length read specifying the stateid associated with the lock in
question can be synthesised to trigger the renewal. If the response
to the request is success, the client has validated all of the locks
governed by that stateid and re-established the appropriate state
between itself and the server.

If the I/O request is not successful, then one or more of the locks
associated with the stateid was revoked by the server and the client
must notify the owner.

9.9. Share Reservations

   A share reservation is a mechanism to control access to a file. It
   is a separate and independent mechanism from byte-range locking.
   When a client opens a file, it issues an OPEN operation to the server
   specifying the type of access required (READ, WRITE, or BOTH) and the
   type of access to deny others (OPEN4_SHARE_DENY_NONE,
   OPEN4_SHARE_DENY_READ, OPEN4_SHARE_DENY_WRITE, or
   OPEN4_SHARE_DENY_BOTH). If the OPEN fails the client will fail the
   application's open request.

   Pseudo-code definition of the semantics:

      if (request.access == 0)
            return (NFS4ERR_INVAL)
      else if ((request.access & file_state.deny)) ||
         (request.deny & file_state.access))
            return (NFS4ERR_DENIED)

   This checking of share reservations on OPEN is done with no exception
   for an existing OPEN for the same open-owner.

   The constants used for the OPEN and OPEN_DOWNGRADE operations for the
   access and deny fields are as follows:

   const OPEN4_SHARE_ACCESS_READ   = 0x00000001;
   const OPEN4_SHARE_ACCESS_WRITE   = 0x00000002;
   const OPEN4_SHARE_ACCESS_BOTH   = 0x00000003;

   const OPEN4_SHARE_DENY_NONE   = 0x00000000;
   const OPEN4_SHARE_DENY_READ   = 0x00000001;
   const OPEN4_SHARE_DENY_WRITE   = 0x00000002;
   const OPEN4_SHARE_DENY_BOTH   = 0x00000003;

9.10. OPEN/CLOSE Operations

   To provide correct share semantics, a client MUST use the OPEN
   operation to obtain the initial filehandle and indicate the desired
   access and what access, if any, to deny. Even if the client intends
   to use one of the special stateids (anonymous stateid or READ bypass
   stateid), it must still obtain the filehandle for the regular file
   with the OPEN operation so the appropriate share semantics can be
   applied. Clients that do not have a deny mode built into their
   programming interfaces for opening a file should request a deny mode
   of OPEN4_SHARE_DENY_NONE.

The OPEN operation with the CREATE flag, also subsumes the CREATE operation for regular files as used in previous versions of the NFS protocol. This allows a create with a share to be done atomically.

The CLOSE operation removes all share reservations held by the open-owner on that file. If byte-range locks are held, the client SHOULD release all locks before issuing a CLOSE. The server MAY free all outstanding locks on CLOSE but some servers may not support the CLOSE of a file that still has byte-range locks held. The server MUST return failure, NFS4ERR_LOCKS_HELD, if any locks would exist after the CLOSE.

The LOOKUP operation will return a filehandle without establishing any lock state on the server. Without a valid stateid, the server will assume the client has the least access. For example, if one client opened a file with OPEN4_SHARE_DENY_BOTH and another client accesses the file via a filehandle obtained through LOOKUP, the second client could only read the file using the special READ bypass stateid. The second client could not WRITE the file at all because it would not have a valid stateid from OPEN and the special anonymous stateid would not be allowed access.

9.10.1. Close and Retention of State Information

Since a CLOSE operation requests deallocation of a stateid, dealing with retransmission of the CLOSE, may pose special difficulties, since the state information, which normally would be used to determine the state of the open file being designated, might be deallocated, resulting in an NFS4ERR_BAD_STATEID error.

Servers may deal with this problem in a number of ways. To provide the greatest degree assurance that the protocol is being used properly, a server should, rather than deallocate the stateid, mark it as close-pending, and retain the stateid with this status, until later deallocation. In this way, a retransmitted CLOSE can be recognized since the stateid points to state information with this distinctive status, so that it can be handled without error.

When adopting this strategy, a server should retain the state information until the earliest of:

o Another validly sequenced request for the same open-owner, that is not a retransmission.

o The time that an open-owner is freed by the server due to period with no activity.

o All locks for the client are freed as a result of a SETCLIENTID.

Servers may avoid this complexity, at the cost of less complete
protocol error checking, by simply responding NFS4_OK in the event of
a CLOSE for a deallocated stateid, on the assumption that this case
must be caused by a retransmitted close. When adopting this
approach, it is desirable to at least log an error when returning a
no-error indication in this situation. If the server maintains a
reply-cache mechanism, it can verify the CLOSE is indeed a
retransmission and avoid error logging in most cases.

9.11. Open Upgrade and Downgrade

When an OPEN is done for a file and the open-owner for which the open
is being done already has the file open, the result is to upgrade the
open file status maintained on the server to include the access and
deny bits specified by the new OPEN as well as those for the existing
OPEN. The result is that there is one open file, as far as the
protocol is concerned, and it includes the union of the access and
deny bits for all of the OPEN requests completed. Only a single
CLOSE will be done to reset the effects of both OPENs. Note that the
client, when issuing the OPEN, may not know that the same file is in
fact being opened. The above only applies if both OPENs result in
the OPENed object being designated by the same filehandle.

When the server chooses to export multiple filehandles corresponding
to the same file object and returns different filehandles on two
different OPENs of the same file object, the server MUST NOT "OR"
together the access and deny bits and coalesce the two open files.
Instead the server must maintain separate OPENs with separate
stateids and will require separate CLOSEs to free them.

When multiple open files on the client are merged into a single open
file object on the server, the close of one of the open files (on the
client) may necessitate change of the access and deny status of the
open file on the server. This is because the union of the access and
deny bits for the remaining opens may be smaller (i.e., a proper
subset) than previously. The OPEN_DOWNGRADE operation is used to
make the necessary change and the client should use it to update the
server so that share reservation requests by other clients are
handled properly. The stateid returned has the same "other" field as
that passed to the server. The "seqid" value in the returned stateid
MUST be incremented (Section 9.1.4), even in situations in which
there has been no change to the access and deny bits for the file.

9.12. Short and Long Leases

When determining the time period for the server lease, the usual
lease tradeoffs apply. Short leases are good for fast server
recovery at a cost of increased RENEW or READ (with zero length)

requests. Longer leases are certainly kinder and gentler to servers
trying to handle very large numbers of clients. The number of RENEW
requests drop in proportion to the lease time. The disadvantages of
long leases are slower recovery after server failure (the server must
wait for the leases to expire and the grace period to elapse before
granting new lock requests) and increased file contention (if client
fails to transmit an unlock request then server must wait for lease
expiration before granting new locks).

Long leases are usable if the server is able to store lease state in
non-volatile memory. Upon recovery, the server can reconstruct the
lease state from its non-volatile memory and continue operation with
its clients and therefore long leases would not be an issue.

9.13. Clocks, Propagation Delay, and Calculating Lease Expiration

To avoid the need for synchronized clocks, lease times are granted by
the server as a time delta. However, there is a requirement that the
client and server clocks do not drift excessively over the duration
of the lock. There is also the issue of propagation delay across the
network which could easily be several hundred milliseconds as well as
the possibility that requests will be lost and need to be
retransmitted.

To take propagation delay into account, the client should subtract it
from lease times (e.g., if the client estimates the one-way
propagation delay as 200 msec, then it can assume that the lease is
already 200 msec old when it gets it). In addition, it will take
another 200 msec to get a response back to the server. So the client
must send a lock renewal or write data back to the server 400 msec
before the lease would expire.

The server's lease period configuration should take into account the
network distance of the clients that will be accessing the server's
resources. It is expected that the lease period will take into
account the network propagation delays and other network delay
factors for the client population. Since the protocol does not allow
for an automatic method to determine an appropriate lease period, the
server's administrator may have to tune the lease period.

9.14. Migration, Replication and State

When responsibility for handling a given file system is transferred
to a new server (migration) or the client chooses to use an
alternative server (e.g., in response to server unresponsiveness) in
the context of file system replication, the appropriate handling of
state shared between the client and server (i.e., locks, leases,
stateids, and client IDs) is as described below. The handling

differs between migration and replication. For related discussion of
file server state and recover of such see the sections under
Section 9.6.

If a server replica or a server immigrating a file system agrees to,
or is expected to, accept opaque values from the client that
originated from another server, then servers SHOULD encode the
"opaque" values in network byte order. This way, servers acting as
replicas or immigrating file systems will be able to parse values
like stateids, directory cookies, filehandles, etc. even if their
native byte order is different from other servers cooperating in the
replication and migration of the file system.

9.14.1. Migration and State

In the case of migration, the servers involved in the migration of a
file system SHOULD transfer all server state from the original to the
new server. This must be done in a way that is transparent to the
client. This state transfer will ease the client's transition when a
file system migration occurs. If the servers are successful in
transferring all state, the client will continue to use stateids
assigned by the original server. Therefore the new server must
recognize these stateids as valid. This holds true for the client ID
as well. Since responsibility for an entire file system is
transferred with a migration event, there is no possibility that
conflicts will arise on the new server as a result of the transfer of
locks.

As part of the transfer of information between servers, leases would
be transferred as well. The leases being transferred to the new
server will typically have a different expiration time from those for
the same client, previously on the old server. To maintain the
property that all leases on a given server for a given client expire
at the same time, the server should advance the expiration time to
the later of the leases being transferred or the leases already
present. This allows the client to maintain lease renewal of both
classes without special effort.

The servers may choose not to transfer the state information upon
migration. However, this choice is discouraged. In this case, when
the client presents state information from the original server (e.g.,
in a RENEW op or a READ op of zero length), the client must be
prepared to receive either NFS4ERR_STALE_CLIENTID or
NFS4ERR_STALE_STATEID from the new server. The client should then
recover its state information as it normally would in response to a
server failure. The new server must take care to allow for the
recovery of state information as it would in the event of server
restart.

A client SHOULD re-establish new callback information with the new
server as soon as possible, according to sequences described in
Section 15.35 and Section 15.36. This ensures that server operations
are not blocked by the inability to recall delegations.

9.14.2. Replication and State

Since client switch-over in the case of replication is not under
server control, the handling of state is different. In this case,
leases, stateids and client IDs do not have validity across a
transition from one server to another. The client must re-establish
its locks on the new server. This can be compared to the re-
establishment of locks by means of reclaim-type requests after a
server reboot. The difference is that the server has no provision to
distinguish requests reclaiming locks from those obtaining new locks
or to defer the latter. Thus, a client re-establishing a lock on the
new server (by means of a LOCK or OPEN request), may have the
requests denied due to a conflicting lock. Since replication is
intended for read-only use of file systems, such denial of locks
should not pose large difficulties in practice. When an attempt to
re-establish a lock on a new server is denied, the client should
treat the situation as if his original lock had been revoked.

9.14.3. Notification of Migrated Lease

In the case of lease renewal, the client may not be submitting
requests for a file system that has been migrated to another server.
This can occur because of the implicit lease renewal mechanism. The
client renews leases for all file systems when submitting a request
to any one file system at the server.

In order for the client to schedule renewal of leases that may have
been relocated to the new server, the client must find out about
lease relocation before those leases expire. To accomplish this, all
operations which implicitly renew leases for a client (such as OPEN,
CLOSE, READ, WRITE, RENEW, LOCK, and others), will return the error
NFS4ERR_LEASE_MOVED if responsibility for any of the leases to be
renewed has been transferred to a new server. This condition will
continue until the client receives an NFS4ERR_MOVED error and the
server receives the subsequent GETATTR(fs_locations) for an access to
each file system for which a lease has been moved to a new server.
By convention, the compound including the GETATTR(fs_locations)
SHOULD append a RENEW operation to permit the server to identify the
client doing the access.

Upon receiving the NFS4ERR_LEASE_MOVED error, a client that supports
file system migration MUST probe all file systems from that server on
which it holds open state. Once the client has successfully probed

all those file systems which are migrated, the server MUST resume
normal handling of stateful requests from that client.

In order to support legacy clients that do not handle the
NFS4ERR_LEASE_MOVED error correctly, the server SHOULD time out after
a wait of at least two lease periods, at which time it will resume
normal handling of stateful requests from all clients. If a client
attempts to access the migrated files, the server MUST reply
NFS4ERR_MOVED.

When the client receives an NFS4ERR_MOVED error, the client can
follow the normal process to obtain the new server information
(through the fs_locations attribute) and perform renewal of those
leases on the new server. If the server has not had state
transferred to it transparently, the client will receive either
NFS4ERR_STALE_CLIENTID or NFS4ERR_STALE_STATEID from the new server,
as described above. The client can then recover state information as
it does in the event of server failure.

9.14.4. Migration and the lease_time Attribute

In order that the client may appropriately manage its leases in the
case of migration, the destination server must establish proper
values for the lease_time attribute.

When state is transferred transparently, that state should include
the correct value of the lease_time attribute. The lease_time
attribute on the destination server must never be less than that on
the source since this would result in premature expiration of leases
granted by the source server. Upon migration in which state is
transferred transparently, the client is under no obligation to re-
fetch the lease_time attribute and may continue to use the value
previously fetched (on the source server).

If state has not been transferred transparently (i.e., the client
sees a real or simulated server reboot), the client should fetch the
value of lease_time on the new (i.e., destination) server, and use it
for subsequent locking requests. However the server must respect a
grace period at least as long as the lease_time on the source server,
in order to ensure that clients have ample time to reclaim their
locks before potentially conflicting non-reclaimed locks are granted.
The means by which the new server obtains the value of lease_time on
the old server is left to the server implementations. It is not
specified by the NFS version 4 protocol.

10. Client-Side Caching

   Client-side caching of data, of file attributes, and of file names is
   essential to providing good performance with the NFS protocol.
   Providing distributed cache coherence is a difficult problem and
   previous versions of the NFS protocol have not attempted it.
   Instead, several NFS client implementation techniques have been used
   to reduce the problems that a lack of coherence poses for users.
   These techniques have not been clearly defined by earlier protocol
   specifications and it is often unclear what is valid or invalid
   client behavior.

   The NFSv4 protocol uses many techniques similar to those that have
   been used in previous protocol versions. The NFSv4 protocol does not
   provide distributed cache coherence. However, it defines a more
   limited set of caching guarantees to allow locks and share
   reservations to be used without destructive interference from client
   side caching.

   In addition, the NFSv4 protocol introduces a delegation mechanism
   which allows many decisions normally made by the server to be made
   locally by clients. This mechanism provides efficient support of the
   common cases where sharing is infrequent or where sharing is read-
   only.

10.1. Performance Challenges for Client-Side Caching

   Caching techniques used in previous versions of the NFS protocol have
   been successful in providing good performance. However, several
   scalability challenges can arise when those techniques are used with
   very large numbers of clients. This is particularly true when
   clients are geographically distributed which classically increases
   the latency for cache re-validation requests.

   The previous versions of the NFS protocol repeat their file data
   cache validation requests at the time the file is opened. This
   behavior can have serious performance drawbacks. A common case is
   one in which a file is only accessed by a single client. Therefore,
   sharing is infrequent.

   In this case, repeated reference to the server to find that no
   conflicts exist is expensive. A better option with regards to
   performance is to allow a client that repeatedly opens a file to do
   so without reference to the server. This is done until potentially
   conflicting operations from another client actually occur.

   A similar situation arises in connection with file locking. Sending
   file lock and unlock requests to the server as well as the read and

write requests necessary to make data caching consistent with the
locking semantics (see Section 10.3.2) can severely limit
performance. When locking is used to provide protection against
infrequent conflicts, a large penalty is incurred. This penalty may
discourage the use of file locking by applications.

The NFSv4 protocol provides more aggressive caching strategies with
the following design goals:

o Compatibility with a large range of server semantics.

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

o Requirements for aggressive caching are organized so that a large
   portion of the benefit can be obtained even when not all of the
   requirements can be met.

The appropriate requirements for the server are discussed in later
sections in which specific forms of caching are covered (see
Section 10.4).

10.2. Delegation and Callbacks

Recallable delegation of server responsibilities for a file to a
client improves performance by avoiding repeated requests to the
server in the absence of inter-client conflict. With the use of a
"callback" RPC from server to client, a server recalls delegated
responsibilities when another client engages in sharing of a
delegated file.

A delegation is passed from the server to the client, specifying the
object of the delegation and the type of delegation. There are
different types of delegations but each type contains a stateid to be
used to represent the delegation when performing operations that
depend on the delegation. This stateid is similar to those
associated with locks and share reservations but differs in that the
stateid for a delegation is associated with a client ID and may be
used on behalf of all the open-owners for the given client. A
delegation is made to the client as a whole and not to any specific
process or thread of control within it.

Because callback RPCs may not work in all environments (due to
firewalls, for example), correct protocol operation does not depend
on them. Preliminary testing of callback functionality by means of a
CB_NULL procedure determines whether callbacks can be supported. The
CB_NULL procedure checks the continuity of the callback path. A
server makes a preliminary assessment of callback availability to a

given client and avoids delegating responsibilities until it has
determined that callbacks are supported. Because the granting of a
delegation is always conditional upon the absence of conflicting
access, clients must not assume that a delegation will be granted and
they must always be prepared for OPENs to be processed without any
delegations being granted.

Once granted, a delegation behaves in most ways like a lock. There
is an associated lease that is subject to renewal together with all
of the other leases held by that client.

Unlike locks, an operation by a second client to a delegated file
will cause the server to recall a delegation through a callback.

On recall, the client holding the delegation must flush modified
state (such as modified data) to the server and return the
delegation. The conflicting request will not be acted on until the
recall is complete. The recall is considered complete when the
client returns the delegation or the server times out its wait for
the delegation to be returned and revokes the delegation as a result
of the timeout. In the interim, the server will either delay
responding to conflicting requests or respond to them with
NFS4ERR_DELAY. Following the resolution of the recall, the server
has the information necessary to grant or deny the second client's
request.

At the time the client receives a delegation recall, it may have
substantial state that needs to be flushed to the server. Therefore,
the server should allow sufficient time for the delegation to be
returned since it may involve numerous RPCs to the server. If the
server is able to determine that the client is diligently flushing
state to the server as a result of the recall, the server MAY extend
the usual time allowed for a recall. However, the time allowed for
recall completion should not be unbounded.

An example of this is when responsibility to mediate opens on a given
file is delegated to a client (see Section 10.4). The server will
not know what opens are in effect on the client. Without this
knowledge the server will be unable to determine if the access and
deny state for the file allows any particular open until the
delegation for the file has been returned.

A client failure or a network partition can result in failure to
respond to a recall callback. In this case, the server will revoke
the delegation which in turn will render useless any modified state
still on the client.

Clients need to be aware that server implementers may enforce
practical limitations on the number of delegations issued. Further,
as there is no way to determine which delegations to revoke, the
server is allowed to revoke any. If the server is implemented to
revoke another delegation held by that client, then the client may be
able to determine that a limit has been reached because each new
delegation request results in a revoke. The client could then
determine which delegations it may not need and preemptively release
them.

10.2.1. Delegation Recovery

There are three situations that delegation recovery must deal with:

o Client reboot or restart

o Server reboot or restart (see Section 9.6.3.1)

o Network partition (full or callback-only)

In the event the client reboots or restarts, the confirmation of a
SETCLIENTID done with an nfs_client_id4 with a new verifier4 value
will result in the release of byte-range locks and share
reservations. Delegations, however, may be treated a bit
differently.

There will be situations in which delegations will need to be
reestablished after a client reboots or restarts. The reason for
this is the client may have file data stored locally and this data
was associated with the previously held delegations. The client will
need to reestablish the appropriate file state on the server.

To allow for this type of client recovery, the server MAY allow
delegations to be retained after other sort of locks are released.
This implies that requests from other clients that conflict with
these delegations will need to wait. Because the normal recall
process may require significant time for the client to flush changed
state to the server, other clients need to be prepared for delays
that occur because of a conflicting delegation. In order to give
clients a chance to get through the reboot process during which
leases will not be renewed, the server MAY extend the period for
delegation recovery beyond the typical lease expiration period. For
open delegations, such delegations that are not released are
reclaimed using OPEN with a claim type of CLAIM_DELEGATE_PREV. (See
Section 10.5 and Section 15.18 for discussion of open delegation and
the details of OPEN respectively).

A server MAY support a claim type of CLAIM_DELEGATE_PREV, but if it does, it MUST NOT remove delegations upon SETCLIENTID_CONFIRM and instead MUST make them available for client reclaim using CLAIM_DELEGATE_PREV. The server MUST NOT remove the delegations until either the client does a DELEGPURGE, or one lease period has elapsed from the time the later of the SETCLIENTID_CONFIRM or the last successful CLAIM_DELEGATE_PREV reclaim.

Note that the requirement stated above is not meant to imply that when the server is no longer obliged, as required above, to retain delegation information, that it should necessarily dispose of it. Some specific cases are:

o When the period is terminated by the occurrence of DELEGPURGE, deletion of unreclaimed delegations is appropriate and desirable.

o When the period is terminated by a lease period elapsing without a successful CLAIM_DELEGATE_PREV reclaim, and that situation appears to be the result of a network partition (i.e., lease expiration has occurred), a server's lease expiration approach, possibly including the use of courtesy locks would normally provide for the retention of unreclaimed delegations. Even in the event that lease cancellation occurs, such delegation should be reclaimed using CLAIM_DELEGATE_PREV as part of network partition recovery.

o When the period of non-communicating is followed by a client reboot, unreclaimed delegations, should also be reclaimable by use of CLAIM_DELEGATE_PREV as part of client reboot recovery.

o When the period is terminated by a lease period elapsing without a successful CLAIM_DELEGATE_PREV reclaim, and lease renewal is occurring, the server may well conclude that unreclaimed delegations have been abandoned, and consider the situation as one in which an implied DELEGPURGE should be assumed.

A server that supports a claim type of CLAIM_DELEGATE_PREV MUST support the DELEGPURGE operation, and similarly a server that supports DELEGPURGE MUST support CLAIM_DELEGATE_PREV. A server which does not support CLAIM_DELEGATE_PREV MUST return NFS4ERR_NOTSUPP if the client attempts to use that feature or performs a DELEGPURGE operation.

Support for a claim type of CLAIM_DELEGATE_PREV, is often referred to as providing for "client-persistent delegations" in that they allow use of client persistent storage on the client to store data written by the client, even across a client restart. It should be noted that, with the optional exception noted below, this feature requires

persistent storage to be used on the client and does not add to
persistent storage requirements on the server.

One good way to think about client-persistent delegations is that for
the most part, they function like "courtesy locks", with special
semantic adjustments to allow them to be retained across a client
restart, which cause all other sorts of locks to be freed. Such
locks are generally not retained across a server restart. The one
exception is the case of simultaneous failure of the client and
server and is discussed below.

When the server indicates support of CLAIM_DELEGATE_PREV (implicitly)
by returning NFS_OK to DELEGPURGE, a client with a write delegation,
can use write-back caching for data to be written to the server,
deferring the write-back, until such time as the delegation is
recalled, possibly after intervening client restarts. Similarly,
when the server indicates support of CLAIM_DELEGATE_PREV, a client
with a read delegation and an open-for-write subordinate to that
delegation, may be sure of the integrity of its persistently cached
copy of the file after a client restart without specific verification
of the change attribute.

When the server reboots or restarts, delegations are reclaimed (using
the OPEN operation with CLAIM_PREVIOUS) in a similar fashion to byte-
range locks and share reservations. However, there is a slight
semantic difference. In the normal case, if the server decides that
a delegation should not be granted, it performs the requested action
(e.g., OPEN) without granting any delegation. For reclaim, the
server grants the delegation but a special designation is applied so
that the client treats the delegation as having been granted but
recalled by the server. Because of this, the client has the duty to
write all modified state to the server and then return the
delegation. This process of handling delegation reclaim reconciles
three principles of the NFSv4 protocol:

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

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

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

When a client has more than a single open associated with a
delegation, state for those additional opens can be established using
OPEN operations of type CLAIM_DELEGATE_CUR. When these are used to

establish opens associated with reclaimed delegations, the server
MUST allow them when made within the grace period.

Situations in which there is a series of client and server restarts
where there is no restart of both at the same time, are dealt with
via a combination of CLAIM_DELEGATE_PREV and CLAIM_PREVIOUS reclaim
cycles. Persistent storage is needed only on the client. For each
server failure, a CLAIM_PREVIOUS reclaim cycle is done, while for
each client restart, a CLAIM_DELEGATE_PREV reclaim cycle is done.

To deal with the possibility of simultaneous failure of client and
server (e.g., a data center power outage), the server MAY
persistently store delegation information so that it can respond to a
CLAIM_DELEGATE_PREV reclaim request which it receives from a
restarting client. This is the one case in which persistent
delegation state can be retained across a server restart. A server
is not required to store this information, but if it does do so, it
should do so for write delegations and for read delegations, during
the pendency of which (across multiple client and/or server
instances), some open-for-write was done as part of delegation. When
the space to persistently record such information is limited, the
server should recall delegations in this class in preference to
keeping them active without persistent storage recording.

When a network partition occurs, delegations are subject to freeing
by the server when the lease renewal period expires. This is similar
to the behavior for locks and share reservations, and, as for locks
and share reservations it may be modified by support for "courtesy
locks" in which locks are not freed in the absence of a conflicting
lock request. Whereas, for locks and share reservations, freeing of
locks will occur immediately upon the appearance of a conflicting
request, for delegations, the server MAY institute period during
which conflicting requests are held off. Eventually the occurrence
of a conflicting request from another client will cause revocation of
the delegation.

A loss of the callback path (e.g., by later network configuration
change) will have a similar effect in that it can also result in
revocation of a delegation A recall request will fail and revocation
of the delegation will result.

A client normally finds out about revocation of a delegation when it
uses a stateid associated with a delegation and receives one of the
errors NFS4ERR_EXPIRED, NFS4ERR_BAD_STATEID, or NFS4ERR_ADMIN_REVOKED
(NFS4ERR_EXPIRED indicates that all lock state associated with the
client has been lost). It also may find out about delegation
revocation after a client reboot when it attempts to reclaim a
delegation and receives NFS4ERR_EXPIRED. Note that in the case of a

revoked OPEN_DELEGATE_WRITE delegation, there are issues because data
may have been modified by the client whose delegation is revoked and
separately by other clients. See Section 10.5.1 for a discussion of
such issues. Note also that when delegations are revoked,
information about the revoked delegation will be written by the
server to stable storage (as described in Section 9.6). This is done
to deal with the case in which a server reboots after revoking a
delegation but before the client holding the revoked delegation is
notified about the revocation.

Note that when there is a loss of a delegation, due to a network
partition in which all locks associated with the lease are lost, the
client will also receive the error NFS4ERR_EXPIRED. This case can be
distinguished from other situations in which delegations are revoked
by seeing that the associated clientid becomes invalid so that
NFS4ERR_STALE_CLIENTID is returned when it is used.

When NFS4ERR_EXPIRED is returned, the server MAY retain information
about the delegations held by the client, deleting those that are
invalidated by a conflicting request. Retaining such information
will allow the client to recover all non-invalidated delegations
using the claim type CLAIM_DELEGATE_PREV, once the
SETCLIENTID_CONFIRM is done to recover. Attempted recovery of a
delegation that the client has no record of, typically because they
were invalidated by conflicting requests, will get the error
NFS4ERR_BAD_RECLAIM. Once a reclaim is attempted for all delegations
that the client held, it SHOULD do a DELEGPURGE to allow any
remaining server delegation information to be freed.

10.3. Data Caching

When applications share access to a set of files, they need to be
implemented so as to take account of the possibility of conflicting
access by another application. This is true whether the applications
in question execute on different clients or reside on the same
client.

Share reservations and byte-range locks are the facilities the NFS
version 4 protocol provides to allow applications to coordinate
access by providing mutual exclusion facilities. The NFSv4
protocol's data caching must be implemented such that it does not
invalidate the assumptions that those using these facilities depend
upon.

10.3.1. Data Caching and OPENs

   In order to avoid invalidating the sharing assumptions that
   applications rely on, NFSv4 clients should not provide cached data to
   applications or modify it on behalf of an application when it would
   not be valid to obtain or modify that same data via a READ or WRITE
   operation.

   Furthermore, in the absence of open delegation (see Section 10.4) two
   additional rules apply. Note that these rules are obeyed in practice
   by many NFSv2 and NFSv3 clients.

   o First, cached data present on a client must be revalidated after
      doing an OPEN. Revalidating means that the client fetches the
      change attribute from the server, compares it with the cached
      change attribute, and if different, declares the cached data (as
      well as the cached attributes) as invalid. This is to ensure that
      the data for the OPENed file is still correctly reflected in the
      client's cache. This validation must be done at least when the
      client's OPEN operation includes DENY=WRITE or BOTH thus
      terminating a period in which other clients may have had the
      opportunity to open the file with WRITE access. Clients may
      choose to do the revalidation more often (such as at OPENs
      specifying DENY=NONE) to parallel the NFSv3 protocol's practice
      for the benefit of users assuming this degree of cache
      revalidation.

      Since the change attribute is updated for data and metadata
      modifications, some client implementers may be tempted to use the
      time_modify attribute and not the change attribute to validate
      cached data, so that metadata changes do not spuriously invalidate
      clean data. The implementer is cautioned against this approach.
      The change attribute is guaranteed to change for each update to
      the file, whereas time_modify is guaranteed to change only at the
      granularity of the time_delta attribute. Use by the client's data
      cache validation logic of time_modify and not the change attribute
      runs the risk of the client incorrectly marking stale data as
      valid.

   o Second, modified data must be flushed to the server before closing
      a file OPENed for write. This is complementary to the first rule.
      If the data is not flushed at CLOSE, the revalidation done after
      the client OPENs a file is unable to achieve its purpose. The
      other aspect to flushing the data before close is that the data
      must be committed to stable storage, at the server, before the
      CLOSE operation is requested by the client. In the case of a
      server reboot or restart and a CLOSEd file, it may not be possible

      to retransmit the data to be written to the file. Hence, this
      requirement.

10.3.2. Data Caching and File Locking

   For those applications that choose to use file locking instead of
   share reservations to exclude inconsistent file access, there is an
   analogous set of constraints that apply to client side data caching.
   These rules are effective only if the file locking is used in a way
   that matches in an equivalent way the actual READ and WRITE
   operations executed. This is as opposed to file locking that is
   based on pure convention. For example, it is possible to manipulate
   a two-megabyte file by dividing the file into two one-megabyte
   regions and protecting access to the two regions by file locks on
   bytes zero and one. A lock for write on byte zero of the file would
   represent the right to do READ and WRITE operations on the first
   region. A lock for write on byte one of the file would represent the
   right to do READ and WRITE operations on the second region. As long
   as all applications manipulating the file obey this convention, they
   will work on a local file system. However, they may not work with
   the NFSv4 protocol unless clients refrain from data caching.

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

   o First, when a client obtains a file lock for a particular region,
      the data cache corresponding to that region (if any cached data
      exists) must be revalidated. If the change attribute indicates
      that the file may have been updated since the cached data was
      obtained, the client must flush or invalidate the cached data for
      the newly locked region. A client might choose to invalidate all
      of non-modified cached data that it has for the file but the only
      requirement for correct operation is to invalidate all of the data
      in the newly locked region.

   o Second, before releasing a write lock for a region, all modified
      data for that region must be flushed to the server. The modified
      data must also be written to stable storage.

   Note that flushing data to the server and the invalidation of cached
   data must reflect the actual byte ranges locked or unlocked.
   Rounding these up or down to reflect client cache block boundaries
   will cause problems if not carefully done. For example, writing a
   modified block when only half of that block is within an area being
   unlocked may cause invalid modification to the region outside the
   unlocked area. This, in turn, may be part of a region locked by
   another client. Clients can avoid this situation by synchronously
   performing portions of write operations that overlap that portion
   (initial or final) that is not a full block. Similarly, invalidating

a locked area which is not an integral number of full buffer blocks
would require the client to read one or two partial blocks from the
server if the revalidation procedure shows that the data which the
client possesses may not be valid.

The data that is written to the server as a prerequisite to the
unlocking of a region must be written, at the server, to stable
storage. The client may accomplish this either with synchronous
writes or by following asynchronous writes with a COMMIT operation.
This is required because retransmission of the modified data after a
server reboot might conflict with a lock held by another client.

A client implementation may choose to accommodate applications which
use byte-range locking in non-standard ways (e.g., using a byte-range
lock as a global semaphore) by flushing to the server more data upon
a LOCKU than is covered by the locked range. This may include
modified data within files other than the one for which the unlocks
are being done. In such cases, the client must not interfere with
applications whose READs and WRITEs are being done only within the
bounds of record locks which the application holds. For example, an
application locks a single byte of a file and proceeds to write that
single byte. A client that chose to handle a LOCKU by flushing all
modified data to the server could validly write that single byte in
response to an unrelated unlock. However, it would not be valid to
write the entire block in which that single written byte was located
since it includes an area that is not locked and might be locked by
another client. Client implementations can avoid this problem by
dividing files with modified data into those for which all
modifications are done to areas covered by an appropriate byte-range
lock and those for which there are modifications not covered by a
byte-range lock. Any writes done for the former class of files must
not include areas not locked and thus not modified on the client.

10.3.3. Data Caching and Mandatory File Locking

Client side data caching needs to respect mandatory file locking when
it is in effect. The presence of mandatory file locking for a given
file is indicated when the client gets back NFS4ERR_LOCKED from a
READ or WRITE on a file it has an appropriate share reservation for.
When mandatory locking is in effect for a file, the client must check
for an appropriate file lock for data being read or written. If a
lock exists for the range being read or written, the client may
satisfy the request using the client's validated cache. If an
appropriate file lock is not held for the range of the read or write,
the read or write request must not be satisfied by the client's cache
and the request must be sent to the server for processing. When a
read or write request partially overlaps a locked region, the request

should be subdivided into multiple pieces with each region (locked or
not) treated appropriately.

10.3.4. Data Caching and File Identity

When clients cache data, the file data needs to be organized
according to the file system object to which the data belongs. For
NFSv3 clients, the typical practice has been to assume for the
purpose of caching that distinct filehandles represent distinct file
system objects. The client then has the choice to organize and
maintain the data cache on this basis.

In the NFSv4 protocol, there is now the possibility to have
significant deviations from a "one filehandle per object" model
because a filehandle may be constructed on the basis of the object's
pathname. Therefore, clients need a reliable method to determine if
two filehandles designate the same file system object. If clients
were simply to assume that all distinct filehandles denote distinct
objects and proceed to do data caching on this basis, caching
inconsistencies would arise between the distinct client side objects
which mapped to the same server side object.

By providing a method to differentiate filehandles, the NFSv4
protocol alleviates a potential functional regression in comparison
with the NFSv3 protocol. Without this method, caching
inconsistencies within the same client could occur and this has not
been present in previous versions of the NFS protocol. Note that it
is possible to have such inconsistencies with applications executing
on multiple clients but that is not the issue being addressed here.

For the purposes of data caching, the following steps allow an NFSv4
client to determine whether two distinct filehandles denote the same
server side object:

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

o If GETATTR for any file with an fsid that matches the fsid of the
   two filehandles in question returns a unique_handles attribute
   with a value of TRUE, then the two objects are distinct.

o If GETATTR directed to the two filehandles does not return the
   fileid attribute for both of the handles, then it cannot be
   determined whether the two objects are the same. Therefore,
   operations which depend on that knowledge (e.g., client side data
   caching) cannot be done reliably. Note that if GETATTR does not
   return the fileid attribute for both filehandles, it will return

   it for neither of the filehandles, since the fsid for both
   filehandles is the same.

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

   o Otherwise they are the same object.

10.4. Open Delegation

   When a file is being OPENed, the server may delegate further handling
   of opens and closes for that file to the opening client. Any such
   delegation is recallable, since the circumstances that allowed for
   the delegation are subject to change. In particular, the server may
   receive a conflicting OPEN from another client, the server must
   recall the delegation before deciding whether the OPEN from the other
   client may be granted. Making a delegation is up to the server and
   clients should not assume that any particular OPEN either will or
   will not result in an open delegation. The following is a typical
   set of conditions that servers might use in deciding whether OPEN
   should be delegated:

   o The client must be able to respond to the server's callback
      requests. The server will use the CB_NULL procedure for a test of
      callback ability.

   o The client must have responded properly to previous recalls.

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

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

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

   o The existence of any server-specific semantics of OPEN/CLOSE that
      would make the required handling incompatible with the prescribed
      handling that the delegated client would apply (see below).

   There are two types of open delegations, OPEN_DELEGATE_READ and
   OPEN_DELEGATE_WRITE. A OPEN_DELEGATE_READ delegation allows a client
   to handle, on its own, requests to open a file for reading that do
   not deny read access to others. It MUST, however, continue to send
   all requests to open a file for writing to the server. Multiple
   OPEN_DELEGATE_READ delegations may be outstanding simultaneously and
   do not conflict. A OPEN_DELEGATE_WRITE delegation allows the client

to handle, on its own, all opens. Only one OPEN_DELEGATE_WRITE
delegation may exist for a given file at a given time and it is
inconsistent with any OPEN_DELEGATE_READ delegations.

When a single client holds a OPEN_DELEGATE_READ delegation, it is
assured that no other client may modify the contents or attributes of
the file. If more than one client holds an OPEN_DELEGATE_READ
delegation, then the contents and attributes of that file are not
allowed to change. When a client has an OPEN_DELEGATE_WRITE
delegation, it may modify the file data since no other client will be
accessing the file's data. The client holding a OPEN_DELEGATE_WRITE
delegation may only affect file attributes which are intimately
connected with the file data: size, time_modify, change.

When a client has an open delegation, it does not send OPENs or
CLOSEs to the server but updates the appropriate status internally.
For a OPEN_DELEGATE_READ delegation, opens that cannot be handled
locally (opens for write or that deny read access) must be sent to
the server.

When an open delegation is made, the response to the OPEN contains an
open delegation structure which specifies the following:

o the type of delegation (read or write)

o space limitation information to control flushing of data on close
   (OPEN_DELEGATE_WRITE delegation only, see Section 10.4.1)

o an nfsace4 specifying read and write permissions

o a stateid to represent the delegation for READ and WRITE

The delegation stateid is separate and distinct from the stateid for
the OPEN proper. The standard stateid, unlike the delegation
stateid, is associated with a particular open-owner and will continue
to be valid after the delegation is recalled and the file remains
open.

When a request internal to the client is made to open a file and open
delegation is in effect, it will be accepted or rejected solely on
the basis of the following conditions. Any requirement for other
checks to be made by the delegate should result in open delegation
being denied so that the checks can be made by the server itself.

o The access and deny bits for the request and the file as described
   in Section 9.9.

o The read and write permissions as determined below.

The nfsace4 passed with delegation can be used to avoid frequent
ACCESS calls. The permission check should be as follows:

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

o If the nfsace4 indicates that the open may not be done, then an
   ACCESS request must be sent to the server to obtain the definitive
   answer.

The server may return an nfsace4 that is more restrictive than the
actual ACL of the file. This includes an nfsace4 that specifies
denial of all access. Note that some common practices such as
mapping the traditional user "root" to the user "nobody" may make it
incorrect to return the actual ACL of the file in the delegation
response.

The use of delegation together with various other forms of caching
creates the possibility that no server authentication will ever be
performed for a given user since all of the user's requests might be
satisfied locally. Where the client is depending on the server for
authentication, the client should be sure authentication occurs for
each user by use of the ACCESS operation. This should be the case
even if an ACCESS operation would not be required otherwise. As
mentioned before, the server may enforce frequent authentication by
returning an nfsace4 denying all access with every open delegation.

10.4.1. Open Delegation and Data Caching

OPEN delegation allows much of the message overhead associated with
the opening and closing files to be eliminated. An open when an open
delegation is in effect does not require that a validation message be
sent to the server unless there exists a potential for conflict with
the requested share mode. The continued endurance of the
"OPEN_DELEGATE_READ delegation" provides a guarantee that no OPEN for
write and thus no write has occurred that did not originate from this
client. Similarly, when closing a file opened for write and if
OPEN_DELEGATE_WRITE delegation is in effect, the data written does
not have to be flushed to the server until the open delegation is
recalled. The continued endurance of the open delegation provides a
guarantee that no open and thus no read or write has been done by
another client.

For the purposes of open delegation, READs and WRITEs done without an
OPEN (anonymous and READ bypass stateids) are treated as the
functional equivalents of a corresponding type of OPEN. READs and
WRITEs done with an anonymous stateid done by another client will
force the server to recall a OPEN_DELEGATE_WRITE delegation. A WRITE

with an anonymous stateid done by another client will force a recall
of OPEN_DELEGATE_READ delegations. The handling of a READ bypass
stateid is identical except that a READ done with a READ bypass
stateid will not force a recall of an OPEN_DELEGATE_READ delegation.

With delegations, a client is able to avoid writing data to the
server when the CLOSE of a file is serviced. The file close system
call is the usual point at which the client is notified of a lack of
stable storage for the modified file data generated by the
application. At the close, file data is written to the server and
through normal accounting the server is able to determine if the
available file system space for the data has been exceeded (i.e.,
server returns NFS4ERR_NOSPC or NFS4ERR_DQUOT). This accounting
includes quotas. The introduction of delegations requires that a
alternative method be in place for the same type of communication to
occur between client and server.

In the delegation response, the server provides either the limit of
the size of the file or the number of modified blocks and associated
block size. The server must ensure that the client will be able to
flush data to the server of a size equal to that provided in the
original delegation. The server must make this assurance for all
outstanding delegations. Therefore, the server must be careful in
its management of available space for new or modified data taking
into account available file system space and any applicable quotas.
The server can recall delegations as a result of managing the
available file system space. The client should abide by the server's
state space limits for delegations. If the client exceeds the stated
limits for the delegation, the server's behavior is undefined.

Based on server conditions, quotas or available file system space,
the server may grant OPEN_DELEGATE_WRITE delegations with very
restrictive space limitations. The limitations may be defined in a
way that will always force modified data to be flushed to the server
on close.

With respect to authentication, flushing modified data to the server
after a CLOSE has occurred may be problematic. For example, the user
of the application may have logged off the client and unexpired
authentication credentials may not be present. In this case, the
client may need to take special care to ensure that local unexpired
credentials will in fact be available. One way that this may be
accomplished by tracking the expiration time of credentials and
flushing data well in advance of their expiration.

10.4.2. Open Delegation and File Locks

   When a client holds a OPEN_DELEGATE_WRITE delegation, lock operations
   may be performed locally. This includes those required for mandatory
   file locking. This can be done since the delegation implies that
   there can be no conflicting locks. Similarly, all of the
   revalidations that would normally be associated with obtaining locks
   and the flushing of data associated with the releasing of locks need
   not be done.

   When a client holds a OPEN_DELEGATE_READ delegation, lock operations
   are not performed locally. All lock operations, including those
   requesting non-exclusive locks, are sent to the server for
   resolution.

10.4.3. Handling of CB_GETATTR

   The server needs to employ special handling for a GETATTR where the
   target is a file that has a OPEN_DELEGATE_WRITE delegation in effect.
   The reason for this is that the client holding the
   OPEN_DELEGATE_WRITE delegation may have modified the data and the
   server needs to reflect this change to the second client that
   submitted the GETATTR. Therefore, the client holding the
   OPEN_DELEGATE_WRITE delegation needs to be interrogated. The server
   will use the CB_GETATTR operation. The only attributes that the
   server can reliably query via CB_GETATTR are size and change.

   Since CB_GETATTR is being used to satisfy another client's GETATTR
   request, the server only needs to know if the client holding the
   delegation has a modified version of the file. If the client's copy
   of the delegated file is not modified (data or size), the server can
   satisfy the second client's GETATTR request from the attributes
   stored locally at the server. If the file is modified, the server
   only needs to know about this modified state. If the server
   determines that the file is currently modified, it will respond to
   the second client's GETATTR as if the file had been modified locally
   at the server.

   Since the form of the change attribute is determined by the server
   and is opaque to the client, the client and server need to agree on a
   method of communicating the modified state of the file. For the size
   attribute, the client will report its current view of the file size.
   For the change attribute, the handling is more involved.

   For the client, the following steps will be taken when receiving a
   OPEN_DELEGATE_WRITE delegation:

o The value of the change attribute will be obtained from the server
   and cached. Let this value be represented by c.

o The client will create a value greater than c that will be used
   for communicating modified data is held at the client. Let this
   value be represented by d.

o When the client is queried via CB_GETATTR for the change
   attribute, it checks to see if it holds modified data. If the
   file is modified, the value d is returned for the change attribute
   value. If this file is not currently modified, the client returns
   the value c for the change attribute.

For simplicity of implementation, the client MAY for each CB_GETATTR
return the same value d. This is true even if, between successive
CB_GETATTR operations, the client again modifies in the file's data
or metadata in its cache. The client can return the same value
because the only requirement is that the client be able to indicate
to the server that the client holds modified data. Therefore, the
value of d may always be c + 1.

While the change attribute is opaque to the client in the sense that
it has no idea what units of time, if any, the server is counting
change with, it is not opaque in that the client has to treat it as
an unsigned integer, and the server has to be able to see the results
of the client's changes to that integer. Therefore, the server MUST
encode the change attribute in network order when sending it to the
client. The client MUST decode it from network order to its native
order when receiving it and the client MUST encode it network order
when sending it to the server. For this reason, the change attribute
is defined as an unsigned integer rather than an opaque array of
bytes.

For the server, the following steps will be taken when providing a
OPEN_DELEGATE_WRITE delegation:

o Upon providing a OPEN_DELEGATE_WRITE delegation, the server will
   cache a copy of the change attribute in the data structure it uses
   to record the delegation. Let this value be represented by sc.

o When a second client sends a GETATTR operation on the same file to
   the server, the server obtains the change attribute from the first
   client. Let this value be cc.

o If the value cc is equal to sc, the file is not modified and the
   server returns the current values for change, time_metadata, and
   time_modify (for example) to the second client.

o If the value cc is NOT equal to sc, the file is currently modified
   at the first client and most likely will be modified at the server
   at a future time. The server then uses its current time to
   construct attribute values for time_metadata and time_modify. A
   new value of sc, which we will call nsc, is computed by the
   server, such that nsc >= sc + 1. The server then returns the
   constructed time_metadata, time_modify, and nsc values to the
   requester. The server replaces sc in the delegation record with
   nsc. To prevent the possibility of time_modify, time_metadata,
   and change from appearing to go backward (which would happen if
   the client holding the delegation fails to write its modified data
   to the server before the delegation is revoked or returned), the
   server SHOULD update the file's metadata record with the
   constructed attribute values. For reasons of reasonable
   performance, committing the constructed attribute values to stable
   storage is OPTIONAL.

As discussed earlier in this section, the client MAY return the same
cc value on subsequent CB_GETATTR calls, even if the file was
modified in the client's cache yet again between successive
CB_GETATTR calls. Therefore, the server must assume that the file
has been modified yet again, and MUST take care to ensure that the
new nsc it constructs and returns is greater than the previous nsc it
returned. An example implementation's delegation record would
satisfy this mandate by including a boolean field (let us call it
"modified") that is set to FALSE when the delegation is granted, and
an sc value set at the time of grant to the change attribute value.
The modified field would be set to TRUE the first time cc != sc, and
would stay TRUE until the delegation is returned or revoked. The
processing for constructing nsc, time_modify, and time_metadata would
use this pseudo code:

```
if (!modified) {
    do CB_GETATTR for change and size;

    if (cc != sc)
        modified = TRUE;
} else {
    do CB_GETATTR for size;
}

if (modified) {
    sc = sc + 1;
    time_modify = time_metadata = current_time;
    update sc, time_modify, time_metadata into file's metadata;
}
```

This would return to the client (that sent GETATTR) the attributes it requested, but make sure size comes from what CB_GETATTR returned. The server would not update the file's metadata with the client's modified size.

In the case that the file attribute size is different than the server's current value, the server treats this as a modification regardless of the value of the change attribute retrieved via CB_GETATTR and responds to the second client as in the last step.

This methodology resolves issues of clock differences between client and server and other scenarios where the use of CB_GETATTR breaks down.

It should be noted that the server is under no obligation to use CB_GETATTR and therefore the server MAY simply recall the delegation to avoid its use.

10.4.4. Recall of Open Delegation

The following events necessitate recall of an open delegation:

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

o SETATTR issued by another client

o REMOVE request for the file

o RENAME request for the file as either source or target of the RENAME

Whether a RENAME of a directory in the path leading to the file results in recall of an open delegation depends on the semantics of the server file system. If that file system denies such RENAMEs when a file is open, the recall must be performed to determine whether the file in question is, in fact, open.

In addition to the situations above, the server may choose to recall open delegations at any time if resource constraints make it advisable to do so. Clients should always be prepared for the possibility of recall.

When a client receives a recall for an open delegation, it needs to update state on the server before returning the delegation. These same updates must be done whenever a client chooses to return a delegation voluntarily. The following items of state need to be dealt with:

o If the file associated with the delegation is no longer open and
   no previous CLOSE operation has been sent to the server, a CLOSE
   operation must be sent to the server.

o If a file has other open references at the client, then OPEN
   operations must be sent to the server. The appropriate stateids
   will be provided by the server for subsequent use by the client
   since the delegation stateid will not longer be valid. These OPEN
   requests are done with the claim type of CLAIM_DELEGATE_CUR. This
   will allow the presentation of the delegation stateid so that the
   client can establish the appropriate rights to perform the OPEN.
   (see Section 15.18 for details.)

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

o For a OPEN_DELEGATE_WRITE delegation, if at the time of recall the
   file is not open for write, all modified data for the file must be
   flushed to the server. If the delegation had not existed, the
   client would have done this data flush before the CLOSE operation.

o For a OPEN_DELEGATE_WRITE delegation when a file is still open at
   the time of recall, any modified data for the file needs to be
   flushed to the server.

o With the OPEN_DELEGATE_WRITE delegation in place, it is possible
   that the file was truncated during the duration of the delegation.
   For example, the truncation could have occurred as a result of an
   OPEN UNCHECKED4 with a size attribute value of zero. Therefore,
   if a truncation of the file has occurred and this operation has
   not been propagated to the server, the truncation must occur
   before any modified data is written to the server.

In the case of OPEN_DELEGATE_WRITE delegation, file locking imposes
some additional requirements. To precisely maintain the associated
invariant, it is required to flush any modified data in any region
for which a write lock was released while the OPEN_DELEGATE_WRITE
delegation was in effect. However, because the OPEN_DELEGATE_WRITE
delegation implies no other locking by other clients, a simpler
implementation is to flush all modified data for the file (as
described just above) if any write lock has been released while the
OPEN_DELEGATE_WRITE delegation was in effect.

An implementation need not wait until delegation recall (or deciding
to voluntarily return a delegation) to perform any of the above
actions, if implementation considerations (e.g., resource
availability constraints) make that desirable. Generally, however,

the fact that the actual open state of the file may continue to
change makes it not worthwhile to send information about opens and
closes to the server, except as part of delegation return. Only in
the case of closing the open that resulted in obtaining the
delegation would clients be likely to do this early, since, in that
case, the close once done will not be undone. Regardless of the
client's choices on scheduling these actions, all must be performed
before the delegation is returned, including (when applicable) the
close that corresponds to the open that resulted in the delegation.
These actions can be performed either in previous requests or in
previous operations in the same COMPOUND request.

10.4.5. OPEN Delegation Race with CB_RECALL

   The server informs the client of recall via a CB_RECALL. A race case
   which may develop is when the delegation is immediately recalled
   before the COMPOUND which established the delegation is returned to
   the client. As the CB_RECALL provides both a stateid and a
   filehandle for which the client has no mapping, it cannot honor the
   recall attempt. At this point, the client has two choices, either do
   not respond or respond with NFS4ERR_BADHANDLE. If it does not
   respond, then it runs the risk of the server deciding to not grant it
   further delegations.

   If instead it does reply with NFS4ERR_BADHANDLE, then both the client
   and the server might be able to detect that a race condition is
   occurring. The client can keep a list of pending delegations. When
   it receives a CB_RECALL for an unknown delegation, it can cache the
   stateid and filehandle on a list of pending recalls. When it is
   provided with a delegation, it would only use it if it was not on the
   pending recall list. Upon the next CB_RECALL, it could immediately
   return the delegation.

   In turn, the server can keep track of when it issues a delegation and
   assume that if a client responds to the CB_RECALL with a
   NFS4ERR_BADHANDLE, then the client has yet to receive the delegation.
   The server SHOULD give the client a reasonable time both to get this
   delegation and to return it before revoking the delegation. Unlike a
   failed callback path, the server should periodically probe the client
   with CB_RECALL to see if it has received the delegation and is ready
   to return it.

   When the server finally determines that enough time has lapsed, it
   SHOULD revoke the delegation and it SHOULD NOT revoke the lease.
   During this extended recall process, the server SHOULD be renewing
   the client lease. The intent here is that the client not pay too
   onerous a burden for a condition caused by the server.

10.4.6. Clients that Fail to Honor Delegation Recalls

   A client may fail to respond to a recall for various reasons, such as
   a failure of the callback path from server to the client. The client
   may be unaware of a failure in the callback path. This lack of
   awareness could result in the client finding out long after the
   failure that its delegation has been revoked, and another client has
   modified the data for which the client had a delegation. This is
   especially a problem for the client that held a OPEN_DELEGATE_WRITE
   delegation.

   The server also has a dilemma in that the client that fails to
   respond to the recall might also be sending other NFS requests,
   including those that renew the lease before the lease expires.
   Without returning an error for those lease renewing operations, the
   server leads the client to believe that the delegation it has is in
   force.

   This difficulty is solved by the following rules:

   o When the callback path is down, the server MUST NOT revoke the
      delegation if one of the following occurs:

      * The client has issued a RENEW operation and the server has
         returned an NFS4ERR_CB_PATH_DOWN error. The server MUST renew
         the lease for any byte-range locks and share reservations the
         client has that the server has known about (as opposed to those
         locks and share reservations the client has established but not
         yet sent to the server, due to the delegation). The server
         SHOULD give the client a reasonable time to return its
         delegations to the server before revoking the client's
         delegations.

      * The client has not issued a RENEW operation for some period of
         time after the server attempted to recall the delegation. This
         period of time MUST NOT be less than the value of the
         lease_time attribute.

   o When the client holds a delegation, it cannot rely on operations,
      except for RENEW, that take a stateid, to renew delegation leases
      across callback path failures. The client that wants to keep
      delegations in force across callback path failures must use RENEW
      to do so.

10.4.7. Delegation Revocation

   At the point a delegation is revoked, if there are associated opens
   on the client, the applications holding these opens need to be
   notified. This notification usually occurs by returning errors for
   READ/WRITE operations or when a close is attempted for the open file.

   If no opens exist for the file at the point the delegation is
   revoked, then notification of the revocation is unnecessary.
   However, if there is modified data present at the client for the
   file, the user of the application should be notified. Unfortunately,
   it may not be possible to notify the user since active applications
   may not be present at the client. See Section 10.5.1 for additional
   details.

10.5. Data Caching and Revocation

   When locks and delegations are revoked, the assumptions upon which
   successful caching depend are no longer guaranteed. For any locks or
   share reservations that have been revoked, the corresponding owner
   needs to be notified. This notification includes applications with a
   file open that has a corresponding delegation which has been revoked.
   Cached data associated with the revocation must be removed from the
   client. In the case of modified data existing in the client's cache,
   that data must be removed from the client without it being written to
   the server. As mentioned, the assumptions made by the client are no
   longer valid at the point when a lock or delegation has been revoked.
   For example, another client may have been granted a conflicting lock
   after the revocation of the lock at the first client. Therefore, the
   data within the lock range may have been modified by the other
   client. Obviously, the first client is unable to guarantee to the
   application what has occurred to the file in the case of revocation.

   Notification to a lock-owner will in many cases consist of simply
   returning an error on the next and all subsequent READs/WRITEs to the
   open file or on the close. Where the methods available to a client
   make such notification impossible because errors for certain
   operations may not be returned, more drastic action such as signals
   or process termination may be appropriate. The justification for
   this is that an invariant for which an application depends on may be
   violated. Depending on how errors are typically treated for the
   client operating environment, further levels of notification
   including logging, console messages, and GUI pop-ups may be
   appropriate.

10.5.1. Revocation Recovery for Write Open Delegation

   Revocation recovery for a OPEN_DELEGATE_WRITE delegation poses the
   special issue of modified data in the client cache while the file is
   not open. In this situation, any client which does not flush
   modified data to the server on each close must ensure that the user
   receives appropriate notification of the failure as a result of the
   revocation. Since such situations may require human action to
   correct problems, notification schemes in which the appropriate user
   or administrator is notified may be necessary. Logging and console
   messages are typical examples.

   If there is modified data on the client, it must not be flushed
   normally to the server. A client may attempt to provide a copy of
   the file data as modified during the delegation under a different
   name in the file system name space to ease recovery. Note that when
   the client can determine that the file has not been modified by any
   other client, or when the client has a complete cached copy of the
   file in question, such a saved copy of the client's view of the file
   may be of particular value for recovery. In other cases, recovery
   using a copy of the file based partially on the client's cached data
   and partially on the server copy as modified by other clients, will
   be anything but straightforward, so clients may avoid saving file
   contents in these situations or mark the results specially to warn
   users of possible problems.

   Saving of such modified data in delegation revocation situations may
   be limited to files of a certain size or might be used only when
   sufficient disk space is available within the target file system.
   Such saving may also be restricted to situations when the client has
   sufficient buffering resources to keep the cached copy available
   until it is properly stored to the target file system.

10.6. Attribute Caching

   The attributes discussed in this section do not include named
   attributes. Individual named attributes are analogous to files and
   caching of the data for these needs to be handled just as data
   caching is for regular files. Similarly, LOOKUP results from an
   OPENATTR directory are to be cached on the same basis as any other
   pathnames and similarly for directory contents.

   Clients may cache file attributes obtained from the server and use
   them to avoid subsequent GETATTR requests. Such caching is write
   through in that modification to file attributes is always done by
   means of requests to the server and should not be done locally and
   cached. The exception to this are modifications to attributes that
   are intimately connected with data caching. Therefore, extending a

file by writing data to the local data cache is reflected immediately
in the size as seen on the client without this change being
immediately reflected on the server. Normally such changes are not
propagated directly to the server but when the modified data is
flushed to the server, analogous attribute changes are made on the
server. When open delegation is in effect, the modified attributes
may be returned to the server in the response to a CB_GETATTR call.

The result of local caching of attributes is that the attribute
caches maintained on individual clients will not be coherent.
Changes made in one order on the server may be seen in a different
order on one client and in a third order on a different client.

The typical file system application programming interfaces do not
provide means to atomically modify or interrogate attributes for
multiple files at the same time. The following rules provide an
environment where the potential incoherency mentioned above can be
reasonably managed. These rules are derived from the practice of
previous NFS protocols.

o All attributes for a given file (per-fsid attributes excepted) are
   cached as a unit at the client so that no non-serializability can
   arise within the context of a single file.

o An upper time boundary is maintained on how long a client cache
   entry can be kept without being refreshed from the server.

o When operations are performed that modify attributes at the
   server, the updated attribute set is requested as part of the
   containing RPC. This includes directory operations that update
   attributes indirectly. This is accomplished by following the
   modifying operation with a GETATTR operation and then using the
   results of the GETATTR to update the client's cached attributes.

Note that if the full set of attributes to be cached is requested by
READDIR, the results can be cached by the client on the same basis as
attributes obtained via GETATTR.

A client may validate its cached version of attributes for a file by
fetching just both the change and time_access attributes and assuming
that if the change attribute has the same value as it did when the
attributes were cached, then no attributes other than time_access
have changed. The reason why time_access is also fetched is because
many servers operate in environments where the operation that updates
change does not update time_access. For example, POSIX file
semantics do not update access time when a file is modified by the
write system call. Therefore, the client that wants a current

time_access value should fetch it with change during the attribute
cache validation processing and update its cached time_access.

The client may maintain a cache of modified attributes for those
attributes intimately connected with data of modified regular files
(size, time_modify, and change). Other than those three attributes,
the client MUST NOT maintain a cache of modified attributes.
Instead, attribute changes are immediately sent to the server.

In some operating environments, the equivalent to time_access is
expected to be implicitly updated by each read of the content of the
file object. If an NFS client is caching the content of a file
object, whether it is a regular file, directory, or symbolic link,
the client SHOULD NOT update the time_access attribute (via SETATTR
or a small READ or READDIR request) on the server with each read that
is satisfied from cache. The reason is that this can defeat the
performance benefits of caching content, especially since an explicit
SETATTR of time_access may alter the change attribute on the server.
If the change attribute changes, clients that are caching the content
will think the content has changed, and will re-read unmodified data
from the server. Nor is the client encouraged to maintain a modified
version of time_access in its cache, since this would mean that the
client will either eventually have to write the access time to the
server with bad performance effects, or it would never update the
server's time_access, thereby resulting in a situation where an
application that caches access time between a close and open of the
same file observes the access time oscillating between the past and
present. The time_access attribute always means the time of last
access to a file by a read that was satisfied by the server. This
way clients will tend to see only time_access changes that go forward
in time.

10.7. Data and Metadata Caching and Memory Mapped Files

Some operating environments include the capability for an application
to map a file's content into the application's address space. Each
time the application accesses a memory location that corresponds to a
block that has not been loaded into the address space, a page fault
occurs and the file is read (or if the block does not exist in the
file, the block is allocated and then instantiated in the
application's address space).

As long as each memory mapped access to the file requires a page
fault, the relevant attributes of the file that are used to detect
access and modification (time_access, time_metadata, time_modify, and
change) will be updated. However, in many operating environments,
when page faults are not required these attributes will not be
updated on reads or updates to the file via memory access (regardless

of whether the file is a local file or is being accessed remotely).
A client or server MAY fail to update attributes of a file that is
being accessed via memory mapped I/O. This has several implications:

o If there is an application on the server that has memory mapped a
   file that a client is also accessing, the client may not be able
   to get a consistent value of the change attribute to determine
   whether its cache is stale or not. A server that knows that the
   file is memory mapped could always pessimistically return updated
   values for change so as to force the application to always get the
   most up to date data and metadata for the file. However, due to
   the negative performance implications of this, such behavior is
   OPTIONAL.

o If the memory mapped file is not being modified on the server, and
   instead is just being read by an application via the memory mapped
   interface, the client will not see an updated time_access
   attribute. However, in many operating environments, neither will
   any process running on the server. Thus NFS clients are at no
   disadvantage with respect to local processes.

o If there is another client that is memory mapping the file, and if
   that client is holding a OPEN_DELEGATE_WRITE delegation, the same
   set of issues as discussed in the previous two bullet items apply.
   So, when a server does a CB_GETATTR to a file that the client has
   modified in its cache, the response from CB_GETATTR will not
   necessarily be accurate. As discussed earlier, the client's
   obligation is to report that the file has been modified since the
   delegation was granted, not whether it has been modified again
   between successive CB_GETATTR calls, and the server MUST assume
   that any file the client has modified in cache has been modified
   again between successive CB_GETATTR calls. Depending on the
   nature of the client's memory management system, this weak
   obligation may not be possible. A client MAY return stale
   information in CB_GETATTR whenever the file is memory mapped.

o The mixture of memory mapping and file locking on the same file is
   problematic. Consider the following scenario, where the page size
   on each client is 8192 bytes.

   * Client A memory maps first page (8192 bytes) of file X

   * Client B memory maps first page (8192 bytes) of file X

   * Client A write locks first 4096 bytes

   * Client B write locks second 4096 bytes

      * Client A, via a STORE instruction modifies part of its locked
         region.

      * Simultaneous to client A, client B issues a STORE on part of
         its locked region.

   Here the challenge is for each client to resynchronize to get a
   correct view of the first page. In many operating environments, the
   virtual memory management systems on each client only know a page is
   modified, not that a subset of the page corresponding to the
   respective lock regions has been modified. So it is not possible for
   each client to do the right thing, which is to only write to the
   server that portion of the page that is locked. For example, if
   client A simply writes out the page, and then client B writes out the
   page, client A's data is lost.

   Moreover, if mandatory locking is enabled on the file, then we have a
   different problem. When clients A and B issue the STORE
   instructions, the resulting page faults require a byte-range lock on
   the entire page. Each client then tries to extend their locked range
   to the entire page, which results in a deadlock.

   Communicating the NFS4ERR_DEADLOCK error to a STORE instruction is
   difficult at best.

   If a client is locking the entire memory mapped file, there is no
   problem with advisory or mandatory byte-range locking, at least until
   the client unlocks a region in the middle of the file.

   Given the above issues the following are permitted:

   o Clients and servers MAY deny memory mapping a file they know there
      are byte-range locks for.

   o Clients and servers MAY deny a byte-range lock on a file they know
      is memory mapped.

   o A client MAY deny memory mapping a file that it knows requires
      mandatory locking for I/O. If mandatory locking is enabled after
      the file is opened and mapped, the client MAY deny the application
      further access to its mapped file.

10.8. Name Caching

   The results of LOOKUP and READDIR operations may be cached to avoid
   the cost of subsequent LOOKUP operations. Just as in the case of
   attribute caching, inconsistencies may arise among the various client
   caches. To mitigate the effects of these inconsistencies and given

the context of typical file system APIs, an upper time boundary is
maintained on how long a client name cache entry can be kept without
verifying that the entry has not been made invalid by a directory
change operation performed by another client.

When a client is not making changes to a directory for which there
exist name cache entries, the client needs to periodically fetch
attributes for that directory to ensure that it is not being
modified. After determining that no modification has occurred, the
expiration time for the associated name cache entries may be updated
to be the current time plus the name cache staleness bound.

When a client is making changes to a given directory, it needs to
determine whether there have been changes made to the directory by
other clients. It does this by using the change attribute as
reported before and after the directory operation in the associated
change_info4 value returned for the operation. The server is able to
communicate to the client whether the change_info4 data is provided
atomically with respect to the directory operation. If the change
values are provided atomically, the client is then able to compare
the pre-operation change value with the change value in the client's
name cache. If the comparison indicates that the directory was
updated by another client, the name cache associated with the
modified directory is purged from the client. If the comparison
indicates no modification, the name cache can be updated on the
client to reflect the directory operation and the associated timeout
extended. The post-operation change value needs to be saved as the
basis for future change_info4 comparisons.

As demonstrated by the scenario above, name caching requires that the
client revalidate name cache data by inspecting the change attribute
of a directory at the point when the name cache item was cached.
This requires that the server update the change attribute for
directories when the contents of the corresponding directory is
modified. For a client to use the change_info4 information
appropriately and correctly, the server must report the pre and post
operation change attribute values atomically. When the server is
unable to report the before and after values atomically with respect
to the directory operation, the server must indicate that fact in the
change_info4 return value. When the information is not atomically
reported, the client should not assume that other clients have not
changed the directory.

10.9. Directory Caching

The results of READDIR operations may be used to avoid subsequent
READDIR operations. Just as in the cases of attribute and name
caching, inconsistencies may arise among the various client caches.

To mitigate the effects of these inconsistencies, and given the
context of typical file system APIs, the following rules should be
followed:

o Cached READDIR information for a directory which is not obtained
   in a single READDIR operation must always be a consistent snapshot
   of directory contents. This is determined by using a GETATTR
   before the first READDIR and after the last of READDIR that
   contributes to the cache.

o An upper time boundary is maintained to indicate the length of
   time a directory cache entry is considered valid before the client
   must revalidate the cached information.

The revalidation technique parallels that discussed in the case of
name caching. When the client is not changing the directory in
question, checking the change attribute of the directory with GETATTR
is adequate. The lifetime of the cache entry can be extended at
these checkpoints. When a client is modifying the directory, the
client needs to use the change_info4 data to determine whether there
are other clients modifying the directory. If it is determined that
no other client modifications are occurring, the client may update
its directory cache to reflect its own changes.

As demonstrated previously, directory caching requires that the
client revalidate directory cache data by inspecting the change
attribute of a directory at the point when the directory was cached.
This requires that the server update the change attribute for
directories when the contents of the corresponding directory is
modified. For a client to use the change_info4 information
appropriately and correctly, the server must report the pre and post
operation change attribute values atomically. When the server is
unable to report the before and after values atomically with respect
to the directory operation, the server must indicate that fact in the
change_info4 return value. When the information is not atomically
reported, the client should not assume that other clients have not
changed the directory.

11. Minor Versioning

To address the requirement of an NFS protocol that can evolve as the
need arises, the NFSv4 protocol contains the rules and framework to
allow for future minor changes or versioning.

The base assumption with respect to minor versioning is that any
future accepted minor version must follow the IETF process and be
documented in a standards track RFC. Therefore, each minor version
number will correspond to an RFC. Minor version 0 of the NFS version

4 protocol is represented by this RFC. The COMPOUND and CB_COMPOUND
procedures support the encoding of the minor version being requested
by the client.

Future minor versions will extend, rather than replace the XDR for
the preceding minor version, as had been done in moving from NFSv2 to
NFSv3 and from NFSv3 to NFSv4.0.

Specification of detailed rules for the construction of minor
versions will be addressed in documents defining early minor versions
or, more desirably, in an RFC establishing a versioning framework for
NFSv4 as a whole.

12. Internationalization

12.1. Introduction

Internationalization is a complex topic with its own set of
terminology (see [RFC6365]). The topic is made more complex in
NFSv4.0 by the tangled history and state of NFS implementations.
This section describes what we might call "NFSv4.0
internationalization" (i.e., internationalization as implemented by
existing clients and servers) as the basis upon which NFSv4.0 clients
may implement internationalization support.

This section is based on the behavior of existing implementations.
Note that the behaviors described are each demonstrated by a
combination of an NFSv4 server implementation proper and a server-
side physical file system. It is common for servers and physical
file systems to be configurable as to the behavior shown. In the
discussion below, each configuration that shows different behavior is
considered separately.

Note that in this section, the keywords "MUST", "SHOULD", and "MAY",
retain their normal meanings. However, in deriving this
specification from implementation patterns, we document below how the
normative terms used derive from the behavior of existing
implementations, in those situations in which existing implementation
behavior patterns can be determined.

o Behavior implemented by all existing clients or servers is
   described using "MUST", since new implementations need to follow
   existing ones to be assured of interoperability. While it is
   possible that different behavior might be workable, we have found
   no case where this seems reasonable.

o Behavior implemented by no existing clients or servers is
   described using "MUST NOT", if such behavior poses
   interoperability problems.

o Behavior implemented by most existing clients or servers, where
   that behavior is more desirable than any alternative is described
   using "SHOULD", since new implementations need to follow that
   existing practice unless there are strong reasons to do otherwise.

   The converse holds for "SHOULD NOT".

o Behavior implemented by some, but not all existing clients or
   servers, is described using "MAY", indicating that new
   implementations have a choice as to whether they will behave in
   that way. Thus, new implementations will have the same
   flexibility that existing ones do.

o Behavior implemented by all existing clients or servers, so far as
   is known, but where there remains some uncertainty as to details
   is described using "should". Such cases primarily concern details
   of error returns. New implementations should follow existing
   practice even though such situations generally do not affect
   interoperability.

There are also cases in which certain server behaviors, while not
known to exist, cannot be reliably determined not to exist. In part,
this is a consequence of the long period of time that has elapsed
since the publication of [RFC3530], resulting in a situation in which
those involved in the implementation may no longer be involved in or
aware of working group activities.

In the case of possible server behavior that is neither known to
exist nor known not to exist, we use SHOULD NOT and MUST NOT as
follows, and similarly for "SHOULD" and "MUST".

o In some cases, the potential behavior is not known to exist but is
   of such a nature that, if it were in fact implemented,
   interoperability difficulties would be expected and reported,
   giving us cause to conclude that the potential behavior is not
   implemented. For such behavior, we use MUST NOT. Similarly we
   use "MUST" to apply to the contrary behavior.

o In other cases, potential behavior is not known to exist but the
   behavior, while undesirable, is not of such a nature that we are
   able to draw any conclusions about its potential existence. In
   such cases, we use SHOULD NOT. Similarly we use "SHOULD" to apply
   to the contrary behavior.

In the case of a MAY, SHOULD, or SHOULD NOT that applies to servers, clients need to be aware that there are servers which may or may not take the specified action, and they need to be prepared for either eventuality.

12.2. Limitations on internationalization-related processing in the
       NFSv4 context

There are a number of noteworthy circumstances that limit the degree to which internationalization-related processing can be made universal with regard to NFSv4 clients and servers:

o The NFSv4 client is part of an extensive set of client-side software components whose design and internal interfaces are not within the IETF's purview, limiting the degree to which a particular character encoding may be made standard.

o Server-side handling of file component names is typically implemented within a server-side physical file system, whose handling of character encoding and normalization is not specifiable by the IETF.

o Typical implementation patterns in Unix systems result in the NFSv4 client having no knowledge of the character encoding being used, which may even vary between processes on the same client system.

o Users may need access to files stored previously with non-UTF-8 encodings, or with UTF-8 encodings that do not match any particular normalization form.

12.3. Summary of Server Behavior Types

As mentioned in Section 12.6, servers MAY reject component name strings that are not valid UTF-8. This leads to a number of types of valid server behavior as outlined below. When these are combined with the valid normalization-related behaviors as described in Section 12.4, this leads to the combined behaviors outlined below.

o Servers which limit file component names to UTF-8 strings exist with normalization-related handling described in Section 12.4. These are best described as "UTF-8-only servers".

o Servers which do not limit file component names to UTF-8 strings are very common and are necessary to deal with clients/ applications not oriented to the use of UTF-8. Such servers ignore normalization-related issues and there is no way for them to implement either normalization or representation-independent

   lookups. These are best described as "UTF-8-unaware servers"
   since they treat file component names as uninterpreted strings of
   bytes and have no knowledge of the characters represented. See
   Section 12.7 for details.

   o It is possible for a server to allow component names which are not
      valid UTF-8, while still being aware of the structure of UTF-8
      strings. Such servers could implement either normalization or
      representation-independent lookups, but apply those techniques
      only to valid UTF-8 strings. Such servers are not common, but it
      is possible to configure at least one known server to have this
      behavior. This behavior SHOULD NOT be used due to the possibility
      that a filename using one character set may, by coincidence, have
      the appearance of a UTF-8 filename; the results of UTF-8
      normalization or representation-independent lookups are unlikely
      to be correct in all cases with respect to the other character
      set.

12.4. String Encoding

   Strings that potentially contain characters outside the ASCII range
   [RFC20] are generally represented in NFSv4 using the UTF-8 encoding
   [RFC3629] of Unicode [UNICODE]. See [RFC3629] for precise encoding
   and decoding rules.

   Some details of the protocol treatment depend on the type of string:

   o For strings which are component names, the preferred encoding for
      any non-ASCII characters is the UTF-8 representation of Unicode.

      In many cases, clients have no knowledge of the encoding being
      used, with the encoding done at user-level under control of a per-
      process locale specification. As a result, it may be impossible
      for the NFSv4 client to enforce use of UTF-8. Use of non-UTF-8
      encodings can be problematic since it may interfere with access to
      files stored using other forms of name encoding. Also,
      normalization-related processing (see Section 12.5) of a string
      not encoded in UTF-8 could result in inappropriate name
      modification or aliasing. In cases in which one has a non-UTF8
      encoded name that accidentally conforms to UTF-8 rules,
      substitution of canonically equivalent strings can change the non-
      UTF-8 encoded name drastically.

      The kinds of modification and aliasing mentioned here can lead to
      both false negatives and false positives depending on the strings
      in question, which can result in security issues such as elevation
      of privilege and denial of service (see [RFC6943] for further
      discussion).

o For strings based on domain names, non-ASCII characters MUST be
   represented using the UTF-8 encoding of Unicode, and additional
   string format restrictions apply. See Section 12.6 for details.

o The contents of symbolic links (of type linktext4 in the XDR) MUST
   be treated as opaque data by NFSv4 servers. Although UTF-8
   encoding is often used, it need not be. In this respect, the
   contents of symbolic links are like the contents of regular files
   in that their encoding is not within the scope of this
   specification.

o For other sorts of strings, any non-ASCII characters SHOULD be
   represented using the UTF-8 encoding of Unicode.

12.5. Normalization

   The client and server operating environments may differ in their
   policies and operational methods with respect to character
   normalization (See [UNICODE] for a discussion of normalization
   forms). This difference may also exist between applications on the
   same client. This adds to the difficulty of providing a single
   normalization policy for the protocol that allows for maximal
   interoperability. This issue is similar to the character case issues
   where the server may or may not support case insensitive file name
   matching and may or may not preserve the character case when storing
   file names. The protocol does not mandate a particular behavior but
   allows for a range of useful behaviors.

   The NFS version 4 protocol does not mandate the use of a particular
   normalization form at this time. A subsequent minor version of the
   NFSv4 protocol might specify a particular normalization form.
   Therefore, the server and client can expect that they may receive
   unnormalized characters within protocol requests and responses. If
   the operating environment requires normalization, then the
   implementation will need to normalize the various UTF-8 encoded
   strings within the protocol before presenting the information to an
   application (at the client) or local file system (at the server).

   Server implementations MAY normalize file names to conform to a
   particular normalization form before using the resulting string when
   looking up or creating a file. Servers MAY also perform
   normalization-insensitive string comparisons without modifying the
   names to match a particular normalization form. Except in cases in
   which component names are excluded from normalization-related
   handling because they are not valid UTF-8 strings, a server MUST make
   the same choice (as to whether to normalize or not, the target form
   of normalization and whether to do normalization-insensitive string
   comparisons) in the same way for all accesses to a particular file

system. Servers SHOULD NOT reject a file name because it does not
conform to a particular normalization form as this may deny access to
clients that use a different normalization form.

12.6. Types with Processing Defined by Other Internet Areas

There are two types of strings that NFSv4 deals with that are based
on domain names. Processing of such strings is defined by other
Internet standards, and hence the processing behavior for such
strings should be consistent across all server operating systems and
server file systems.

These are as follows:

o Server names as they appear in the fs_locations attribute. Note
   that for most purposes, such server names will only be sent by the
   server to the client. The exception is use of the fs_locations
   attribute in a VERIFY or NVERIFY operation.

o Principal suffixes which are used to denote sets of users and
   groups, and are in the form of domain names.

The general rules for handling all of these domain-related strings
are similar and independent of the role of the sender or receiver as
client or server although the consequences of failure to obey these
rules may be different for client or server. The server can report
errors when it is sent invalid strings, whereas the client will
simply ignore invalid string or use a default value in their place.

The string sent SHOULD be in the form of one or more U-labels as
defined by [RFC5890]. If that is impractical, it can instead be in
the form of one or more LDH labels [RFC5890] or a UTF-8 domain name
that contains labels that are not properly formatted U-labels. The
receiver needs to be able to accept domain and server names in any of
the formats allowed. The server MUST reject, using the error
NFS4ERR_INVAL, a string that is not valid UTF-8, or that contains an
ASCII label that is not a valid LDH label, or that contains an XN-
label (begins with "xn--") for which the characters after "xn--" are
not valid output of the Punycode algorithm [RFC3492].

When a domain string is part of id@domain or group@domain, there are
two possible approaches:

1. The server treats the domain string as a series of U-labels. In
    cases where the domain string is a series of A-labels or NR-LDH
    labels, it converts them to U-labels using the Punycode algorithm
    [RFC3492]. In cases where the domain string is series of other
    sorts of LDH labels, the server can use the ToUnicode function

defined in [RFC3490] to convert the string to a series of labels
that generally conform to the U-label syntax. In cases where the
domain string is a UTF-8 string that contains non-U-labels, the
server can attempt to use to ToASCII function defined in
[RFC3490] and then the ToUnicode function on the string to
convert it to a series of labels that generally conform to the
U-label syntax. As a result, the domain string returned within a
userid on a GETATTR may not match that sent when the userid is
set using SETATTR, although when this happens, the domain will be
in the form that generally conform to the U-label syntax.

2. The server does not attempt to treat the domain string as a
    series of U-labels; specifically, it does not map a domain string
    which is not a U-label into a U-label using the methods described
    above. As a result, the domain string returned on a GETATTR of
    the userid MUST be the same as that used when setting the userid
    by the SETATTR.

A server SHOULD use the first method.

For VERIFY and NVERIFY, additional string processing requirements
apply to verification of the owner and owner_group attributes, see
Section 5.9.

12.7. UTF-8 Related Errors

Where the client sends an invalid UTF-8 string, the server MAY return
an NFS4ERR_INVAL error. This includes cases in which inappropriate
prefixes are detected and where the count includes trailing bytes
that do not constitute a full UCS character.

Requirements for server handling of component names which are not
valid UTF-8, when a server does not return NFS4ERR_INVAL in response
to receiving them, are described in Section 12.8.

Where the client supplied string is not rejected with NFS4ERR_INVAL
but contains characters that are not supported by the server as a
value for that string (e.g., names containing slashes, or characters
that do not fit into 16 bits when converted from UTF-8 to a Unicode
codepoint), the server should return an NFS4ERR_BADCHAR error.

Where a UTF-8 string is used as a file name, and the file system,
while supporting all of the characters within the name, does not
allow that particular name to be used, the error should return the
error NFS4ERR_BADNAME. This includes such situations as file system
prohibitions of "." and ".." as file names for certain operations,
and similar constraints

12.8. Servers that accept file component names that are not valid UTF-8
       strings

   As stated previously, servers MAY accept, on all or on some subset of
   the physical file systems exported, component names that are not
   valid UTF-8 strings. A typical pattern is for a server to use
   UTF-8-unaware physical file systems that treat component names as
   uninterpreted strings of bytes, rather than having any awareness of
   the character set being used.

   Such servers SHOULD NOT change the stored representation of component
   names from those received on the wire, and SHOULD use an octet-by-
   octet comparison of component name strings to determine equivalence
   (as opposed to any broader notion of string comparison). This is
   because the server has no knowledge of the character encoding being
   used.

   Nonetheless, when such a server uses a broader notion of string
   equivalence than recommended in the preceding paragraph the following
   considerations apply:

   o Outside of 7-bit ASCII, string processing that changes string
      contents is usually specific to a character set and hence is
      generally unsafe when the character set is unknown. This
      processing could change the filename in an unexpected fashion,
      rendering the file inaccessible to the application or client that
      created or renamed the file and to others expecting the original
      filename. Hence, such processing should not be performed because
      doing so is likely to result in incorrect string modification or
      aliasing.

   o Unicode normalization is particularly dangerous, as such
      processing assumes that the string is UTF-8. When that assumption
      is false because a different character set was used to create the
      filename, normalization may corrupt the filename with respect to
      that character set, rendering the file inaccessible to the
      application that created it and others expecting the original
      filename. Hence, Unicode normalization SHOULD NOT be performed,
      because it may cause incorrect string modification or aliasing.

   When the above recommendations are not followed, the resulting string
   modification and aliasing can lead to both false negatives and false
   positives depending on the strings in question, which can result in
   security issues such as elevation of privilege and denial of service
   (see [RFC6943] for further discussion).

13. Error Values

   NFS error numbers are assigned to failed operations within a Compound
   (COMPOUND or CB_COMPOUND) request. A Compound request contains a
   number of NFS operations that have their results encoded in sequence
   in a Compound reply. The results of successful operations will
   consist of an NFS4_OK status followed by the encoded results of the
   operation. If an NFS operation fails, an error status will be
   entered in the reply and the Compound request will be terminated.

13.1. Error Definitions

                     Protocol Error Definitions

Error	Number	Description
NFS4_OK	0	Section 13.1.3.1
NFS4ERR_ACCESS	13	Section 13.1.6.1
NFS4ERR_ADMIN_REVOKED	10047	Section 13.1.5.1
NFS4ERR_ATTRNOTSUPP	10032	Section 13.1.11.1
NFS4ERR_BADCHAR	10040	Section 13.1.7.1
NFS4ERR_BADHANDLE	10001	Section 13.1.2.1
NFS4ERR_BADNAME	10041	Section 13.1.7.2
NFS4ERR_BADOWNER	10039	Section 13.1.11.2
NFS4ERR_BADTYPE	10007	Section 13.1.4.1
NFS4ERR_BADXDR	10036	Section 13.1.1.1
NFS4ERR_BAD_COOKIE	10003	Section 13.1.1.2
NFS4ERR_BAD_RANGE	10042	Section 13.1.8.1
NFS4ERR_BAD_SEQID	10026	Section 13.1.8.2
NFS4ERR_BAD_STATEID	10025	Section 13.1.5.2
NFS4ERR_CB_PATH_DOWN	10048	Section 13.1.12.1
NFS4ERR_CLID_INUSE	10017	Section 13.1.10.1
NFS4ERR_DEADLOCK	10045	Section 13.1.8.3
NFS4ERR_DELAY	10008	Section 13.1.1.3
NFS4ERR_DENIED	10010	Section 13.1.8.4
NFS4ERR_DQUOT	69	Section 13.1.4.2
NFS4ERR_EXIST	17	Section 13.1.4.3
NFS4ERR_EXPIRED	10011	Section 13.1.5.3
NFS4ERR_FBIG	27	Section 13.1.4.4
NFS4ERR_FHEXPIRED	10014	Section 13.1.2.2
NFS4ERR_FILE_OPEN	10046	Section 13.1.4.5
NFS4ERR_GRACE	10013	Section 13.1.9.1
NFS4ERR_INVAL	22	Section 13.1.1.4
NFS4ERR_IO	5	Section 13.1.4.6
NFS4ERR_ISDIR	21	Section 13.1.2.3
NFS4ERR_LEASE_MOVED	10031	Section 13.1.5.4
NFS4ERR_LOCKED	10012	Section 13.1.8.5

NFS4ERR_LOCKS_HELD	10037	Section 13.1.8.6
NFS4ERR_LOCK_NOTSUPP	10043	Section 13.1.8.7
NFS4ERR_LOCK_RANGE	10028	Section 13.1.8.8
NFS4ERR_MINOR_VERS_MISMATCH	10021	Section 13.1.3.2
NFS4ERR_MLINK	31	Section 13.1.4.7
NFS4ERR_MOVED	10019	Section 13.1.2.4
NFS4ERR_NAMETOOLONG	63	Section 13.1.7.3
NFS4ERR_NOENT	2	Section 13.1.4.8
NFS4ERR_NOFILEHANDLE	10020	Section 13.1.2.5
NFS4ERR_NOSPC	28	Section 13.1.4.9
NFS4ERR_NOTDIR	20	Section 13.1.2.6
NFS4ERR_NOTEMPTY	66	Section 13.1.4.10
NFS4ERR_NOTSUPP	10004	Section 13.1.1.5
NFS4ERR_NOT_SAME	10027	Section 13.1.11.3
NFS4ERR_NO_GRACE	10033	Section 13.1.9.2
NFS4ERR_NXIO	6	Section 13.1.4.11
NFS4ERR_OLD_STATEID	10024	Section 13.1.5.5
NFS4ERR_OPENMODE	10038	Section 13.1.8.9
NFS4ERR_OP_ILLEGAL	10044	Section 13.1.3.3
NFS4ERR_PERM	1	Section 13.1.6.2
NFS4ERR_RECLAIM_BAD	10034	Section 13.1.9.3
NFS4ERR_RECLAIM_CONFLICT	10035	Section 13.1.9.4
NFS4ERR_RESOURCE	10018	Section 13.1.3.4
NFS4ERR_RESTOREFH	10030	Section 13.1.4.12
NFS4ERR_ROFS	30	Section 13.1.4.13
NFS4ERR_SAME	10009	Section 13.1.11.4
NFS4ERR_SERVERFAULT	10006	Section 13.1.1.6
NFS4ERR_SHARE_DENIED	10015	Section 13.1.8.10
NFS4ERR_STALE	70	Section 13.1.2.7
NFS4ERR_STALE_CLIENTID	10022	Section 13.1.10.2
NFS4ERR_STALE_STATEID	10023	Section 13.1.5.6
NFS4ERR_SYMLINK	10029	Section 13.1.2.8
NFS4ERR_TOOSMALL	10005	Section 13.1.1.7
NFS4ERR_WRONGSEC	10016	Section 13.1.6.3
NFS4ERR_XDEV	18	Section 13.1.4.14

                              Table 6

13.1.1. General Errors

   This section deals with errors that are applicable to a broad set of
   different purposes.

13.1.1.1. NFS4ERR_BADXDR (Error Code 10036)

   The arguments for this operation do not match those specified in the
   XDR definition. This includes situations in which the request ends
   before all the arguments have been seen. Note that this error
   applies when fixed enumerations (these include booleans) have a value
   within the input stream which is not valid for the enum. A replier
   may pre-parse all operations for a Compound procedure before doing
   any operation execution and return RPC-level XDR errors in that case.

13.1.1.2. NFS4ERR_BAD_COOKIE (Error Code 10003)

   Used for operations that provide a set of information indexed by some
   quantity provided by the client or cookie sent by the server for an
   earlier invocation. Where the value cannot be used for its intended
   purpose, this error results.

13.1.1.3. NFS4ERR_DELAY (Error Code 10008)

   For any of a number of reasons, the replier could not process this
   operation in what was deemed a reasonable time. The client should
   wait and then try the request with a new RPC transaction ID.

   Some example of situations that might lead to this situation:

   o A server that supports hierarchical storage receives a request to
      process a file that had been migrated.

   o An operation requires a delegation recall to proceed and waiting
      for this delegation recall makes processing this request in a
      timely fashion impossible.

13.1.1.4. NFS4ERR_INVAL (Error Code 22)

   The arguments for this operation are not valid for some reason, even
   though they do match those specified in the XDR definition for the
   request.

13.1.1.5. NFS4ERR_NOTSUPP (Error Code 10004)

   Operation not supported, either because the operation is an OPTIONAL
   one and is not supported by this server or because the operation MUST
   NOT be implemented in the current minor version.

13.1.1.6. NFS4ERR_SERVERFAULT (Error Code 10006)

   An error occurred on the server which does not map to any of the
   specific legal NFSv4 protocol error values. The client should
   translate this into an appropriate error. UNIX clients may choose to
   translate this to EIO.

13.1.1.7. NFS4ERR_TOOSMALL (Error Code 10005)

   Used where an operation returns a variable amount of data, with a
   limit specified by the client. Where the data returned cannot be
   fitted within the limit specified by the client, this error results.

13.1.2. Filehandle Errors

   These errors deal with the situation in which the current or saved
   filehandle, or the filehandle passed to PUTFH intended to become the
   current filehandle, is invalid in some way. This includes situations
   in which the filehandle is a valid filehandle in general but is not
   of the appropriate object type for the current operation.

   Where the error description indicates a problem with the current or
   saved filehandle, it is to be understood that filehandles are only
   checked for the condition if they are implicit arguments of the
   operation in question.

13.1.2.1. NFS4ERR_BADHANDLE (Error Code 10001)

   Illegal NFS filehandle for the current server. The current
   filehandle failed internal consistency checks. Once accepted as
   valid (by PUTFH), no subsequent status change can cause the
   filehandle to generate this error.

13.1.2.2. NFS4ERR_FHEXPIRED (Error Code 10014)

   A current or saved filehandle which is an argument to the current
   operation is volatile and has expired at the server.

13.1.2.3. NFS4ERR_ISDIR (Error Code 21)

   The current or saved filehandle designates a directory when the
   current operation does not allow a directory to be accepted as the
   target of this operation.

13.1.2.4. NFS4ERR_MOVED (Error Code 10019)

   The file system which contains the current filehandle object is not
   present at the server. It may have been relocated, migrated to
   another server or may have never been present. The client may obtain
   the new file system location by obtaining the "fs_locations" or
   attribute for the current filehandle. For further discussion, refer
   to Section 8.

13.1.2.5. NFS4ERR_NOFILEHANDLE (Error Code 10020)

   The logical current or saved filehandle value is required by the
   current operation and is not set. This may be a result of a
   malformed COMPOUND operation (i.e., no PUTFH or PUTROOTFH before an
   operation that requires the current filehandle be set).

13.1.2.6. NFS4ERR_NOTDIR (Error Code 20)

   The current (or saved) filehandle designates an object which is not a
   directory for an operation in which a directory is required.

13.1.2.7. NFS4ERR_STALE (Error Code 70)

   The current or saved filehandle value designating an argument to the
   current operation is invalid. The file system object referred to by
   that filehandle no longer exists or access to it has been revoked.

13.1.2.8. NFS4ERR_SYMLINK (Error Code 10029)

   The current filehandle designates a symbolic link when the current
   operation does not allow a symbolic link as the target.

13.1.3. Compound Structure Errors

   This section deals with errors that relate to overall structure of a
   Compound request (by which we mean to include both COMPOUND and
   CB_COMPOUND), rather than to particular operations.

   There are a number of basic constraints on the operations that may
   appear in a Compound request.

13.1.3.1. NFS_OK (Error code 0)

   Indicates the operation completed successfully, in that all of the
   constituent operations completed without error.

13.1.3.2. NFS4ERR_MINOR_VERS_MISMATCH (Error code 10021)

   The minor version specified is not one that the current listener
   supports. This value is returned in the overall status for the
   Compound but is not associated with a specific operation since the
   results must specify a result count of zero.

13.1.3.3. NFS4ERR_OP_ILLEGAL (Error Code 10044)

   The operation code is not a valid one for the current Compound
   procedure. The opcode in the result stream matched with this error
   is the ILLEGAL value, although the value that appears in the request
   stream may be different. Where an illegal value appears and the
   replier pre-parses all operations for a Compound procedure before
   doing any operation execution, an RPC-level XDR error may be returned
   in this case.

13.1.3.4. NFS4ERR_RESOURCE (Error Code 10018)

   For the processing of the Compound procedure, the server may exhaust
   available resources and cannot continue processing operations within
   the Compound procedure. This error will be returned from the server
   in those instances of resource exhaustion related to the processing
   of the Compound procedure.

13.1.4. File System Errors

   These errors describe situations which occurred in the underlying
   file system implementation rather than in the protocol or any NFSv4.x
   feature.

13.1.4.1. NFS4ERR_BADTYPE (Error Code 10007)

   An attempt was made to create an object with an inappropriate type
   specified to CREATE. This may be because the type is undefined,
   because it is a type not supported by the server, or because it is a
   type for which create is not intended such as a regular file or named
   attribute, for which OPEN is used to do the file creation.

13.1.4.2. NFS4ERR_DQUOT (Error Code 69)

   Resource (quota) hard limit exceeded. The user's resource limit on
   the server has been exceeded.

13.1.4.3. NFS4ERR_EXIST (Error Code 17)

   A file system object of the specified target name (when creating,
   renaming or linking) already exists.

13.1.4.4. NFS4ERR_FBIG (Error Code 27)

   Filesystem object too large. The operation would have caused a file
   system object to grow beyond the server's limit.

13.1.4.5. NFS4ERR_FILE_OPEN (Error Code 10046)

   The operation is not allowed because a file system object involved in
   the operation is currently open. Servers may, but are not required
   to disallow linking-to, removing, or renaming open file system
   objects.

13.1.4.6. NFS4ERR_IO (Error Code 5)

   Indicates that an I/O error occurred for which the file system was
   unable to provide recovery.

13.1.4.7. NFS4ERR_MLINK (Error Code 31)

   The request would have caused the server's limit for the number of
   hard links a file system object may have to be exceeded.

13.1.4.8. NFS4ERR_NOENT (Error Code 2)

   Indicates no such file or directory. The file system object
   referenced by the name specified does not exist.

13.1.4.9. NFS4ERR_NOSPC (Error Code 28)

   Indicates no space left on device. The operation would have caused
   the server's file system to exceed its limit.

13.1.4.10. NFS4ERR_NOTEMPTY (Error Code 66)

   An attempt was made to remove a directory that was not empty.

13.1.4.11. NFS4ERR_NXIO (Error Code 6)

   I/O error. No such device or address.

13.1.4.12. NFS4ERR_RESTOREFH (Error Code 10030)

   The RESTOREFH operation does not have a saved filehandle (identified
   by SAVEFH) to operate upon.

13.1.4.13. NFS4ERR_ROFS (Error Code 30)

   Indicates a read-only file system. A modifying operation was
   attempted on a read-only file system.

13.1.4.14. NFS4ERR_XDEV (Error Code 18)

   Indicates an attempt to do an operation, such as linking, that
   inappropriately crosses a boundary. This may be due to such
   boundaries as:

   o That between file systems (where the fsids are different).

   o That between different named attribute directories or between a
      named attribute directory and an ordinary directory.

   o That between regions of a file system that the file system
      implementation treats as separate (for example for space
      accounting purposes), and where cross-connection between the
      regions are not allowed.

13.1.5. State Management Errors

   These errors indicate problems with the stateid (or one of the
   stateids) passed to a given operation. This includes situations in
   which the stateid is invalid as well as situations in which the
   stateid is valid but designates revoked locking state. Depending on
   the operation, the stateid when valid may designate opens, byte-range
   locks, or file delegations.

13.1.5.1. NFS4ERR_ADMIN_REVOKED (Error Code 10047)

   A stateid designates locking state of any type that has been revoked
   due to administrative interaction, possibly while the lease is valid,
   or because a delegation was revoked because of failure to return it,
   while the lease was valid.

13.1.5.2. NFS4ERR_BAD_STATEID (Error Code 10025)

   A stateid generated by the current server instance was used which
   either:

o Does not designate any locking state (either current or
   superseded) for a current (state-owner, file) pair.

o Designates locking state that was freed after lease expiration but
   without any lease cancellation, as may happen in the handling of
   "courtesy locks".

13.1.5.3. NFS4ERR_EXPIRED (Error Code 10011)

   A stateid or clientid designates locking state of any type that has
   been revoked or released due to cancellation of the client's lease,
   either immediately upon lease expiration, or following a later
   request for a conflicting lock.

13.1.5.4. NFS4ERR_LEASE_MOVED (Error Code 10031)

   A lease being renewed is associated with a file system that has been
   migrated to a new server.

13.1.5.5. NFS4ERR_OLD_STATEID (Error Code 10024)

   A stateid is provided with a seqid value that is not the most
   current.

13.1.5.6. NFS4ERR_STALE_STATEID (Error Code 10023)

   A stateid generated by an earlier server instance was used.

13.1.6. Security Errors

   These are the various permission-related errors in NFSv4.

13.1.6.1. NFS4ERR_ACCESS (Error Code 13)

   Indicates permission denied. The caller does not have the correct
   permission to perform the requested operation. Contrast this with
   NFS4ERR_PERM (Section 13.1.6.2), which restricts itself to owner or
   privileged user permission failures.

13.1.6.2. NFS4ERR_PERM (Error Code 1)

   Indicates requester is not the owner. The operation was not allowed
   because the caller is neither a privileged user (root) nor the owner
   of the target of the operation.

13.1.6.3. NFS4ERR_WRONGSEC (Error Code 10016)

   Indicates that the security mechanism being used by the client for
   the operation does not match the server's security policy. The
   client should change the security mechanism being used and re-send
   the operation. SECINFO can be used to determine the appropriate
   mechanism.

13.1.7. Name Errors

   Names in NFSv4 are UTF-8 strings. When the strings are not of length
   zero, the error NFS4ERR_INVAL results. When they are not valid UTF-8
   the error NFS4ERR_INVAL also results, but servers may accommodate
   file systems with different character formats and not return this
   error. Besides this, there are a number of other errors to indicate
   specific problems with names.

13.1.7.1. NFS4ERR_BADCHAR (Error Code 10040)

   A UTF-8 string contains a character which is not supported by the
   server in the context in which it is being used.

13.1.7.2. NFS4ERR_BADNAME (Error Code 10041)

   A name string in a request consisted of valid UTF-8 characters
   supported by the server but the name is not supported by the server
   as a valid name for current operation. An example might be creating
   a file or directory named ".." on a server whose file system uses
   that name for links to parent directories.

   This error should not be returned due a normalization issue in a
   string. When a file system keeps names in a particular normalization
   form, it is the server's responsibility to do the appropriate
   normalization, rather than rejecting the name.

13.1.7.3. NFS4ERR_NAMETOOLONG (Error Code 63)

   Returned when the filename in an operation exceeds the server's
   implementation limit.

13.1.8. Locking Errors

   This section deal with errors related to locking, both as to share
   reservations and byte-range locking. It does not deal with errors
   specific to the process of reclaiming locks. Those are dealt with in
   the next section.

13.1.8.1. NFS4ERR_BAD_RANGE (Error Code 10042)

   The range for a LOCK, LOCKT, or LOCKU operation is not appropriate to
   the allowable range of offsets for the server. E.g., this error
   results when a server which only supports 32-bit ranges receives a
   range that cannot be handled by that server. (See Section 15.12.4).

13.1.8.2. NFS4ERR_BAD_SEQID (Error Code 10026)

   The sequence number (seqid) in a locking request is neither the next
   expected number or the last number processed.

13.1.8.3. NFS4ERR_DEADLOCK (Error Code 10045)

   The server has been able to determine a file locking deadlock
   condition for a blocking lock request.

13.1.8.4. NFS4ERR_DENIED (Error Code 10010)

   An attempt to lock a file is denied. Since this may be a temporary
   condition, the client is encouraged to re-send the lock request until
   the lock is accepted. See Section 9.4 for a discussion of the re-
   send.

13.1.8.5. NFS4ERR_LOCKED (Error Code 10012)

   A read or write operation was attempted on a file where there was a
   conflict between the I/O and an existing lock:

   o There is a share reservation inconsistent with the I/O being done.

   o The range to be read or written intersects an existing mandatory
      byte range lock.

13.1.8.6. NFS4ERR_LOCKS_HELD (Error Code 10037)

   An operation was prevented by the unexpected presence of locks.

13.1.8.7. NFS4ERR_LOCK_NOTSUPP (Error Code 10043)

   A locking request was attempted which would require the upgrade or
   downgrade of a lock range already held by the owner when the server
   does not support atomic upgrade or downgrade of locks.

13.1.8.8. NFS4ERR_LOCK_RANGE (Error Code 10028)

   A lock request is operating on a range that overlaps in part a
   currently held lock for the current lock-owner and does not precisely
   match a single such lock where the server does not support this type
   of request, and thus does not implement POSIX locking semantics
   [fcntl]. See Section 15.12.5, Section 15.13.5, and Section 15.14.5
   for a discussion of how this applies to LOCK, LOCKT, and LOCKU
   respectively.

13.1.8.9. NFS4ERR_OPENMODE (Error Code 10038)

   The client attempted a READ, WRITE, LOCK or other operation not
   sanctioned by the stateid passed (e.g., writing to a file opened only
   for read).

13.1.8.10. NFS4ERR_SHARE_DENIED (Error Code 10015)

   An attempt to OPEN a file with a share reservation has failed because
   of a share conflict.

13.1.9. Reclaim Errors

   These errors relate to the process of reclaiming locks after a server
   restart.

13.1.9.1. NFS4ERR_GRACE (Error Code 10013)

   The server is in its recovery or grace period which should at least
   match the lease period of the server. A locking request other than a
   reclaim could not be granted during that period.

13.1.9.2. NFS4ERR_NO_GRACE (Error Code 10033)

   The server cannot guarantee that it has not granted state to another
   client which may conflict with this client's state. No further
   reclaims from this client will succeed.

13.1.9.3. NFS4ERR_RECLAIM_BAD (Error Code 10034)

   The server cannot guarantee that it has not granted state to another
   client which may conflict with the requested state. However, this
   applies only to the state requested in this call; further reclaims
   may succeed.

   Unlike NFS4ERR_RECLAIM_CONFLICT, this can occur between correctly
   functioning clients and servers: the "edge condition" scenarios
   described in Section 9.6.3.1 leave only the server knowing whether

the client's locks are still valid, and NFS4ERR_RECLAIM_BAD is the
server's way of informing the client that they are not.

13.1.9.4. NFS4ERR_RECLAIM_CONFLICT (Error Code 10035)

The reclaim attempted by the client conflicts with a lock already
held by another client. Unlike NFS4ERR_RECLAIM_BAD, this can only
occur if one of the clients misbehaved.

13.1.10. Client Management Errors

This sections deals with errors associated with requests used to
create and manage client IDs.

13.1.10.1. NFS4ERR_CLID_INUSE (Error Code 10017)

The SETCLIENTID operation has found that a clientid is already in use
by another client.

13.1.10.2. NFS4ERR_STALE_CLIENTID (Error Code 10022)

A client ID not recognized by the server was used in a locking or
SETCLIENTID_CONFIRM request.

13.1.11. Attribute Handling Errors

This section deals with errors specific to attribute handling within
NFSv4.

13.1.11.1. NFS4ERR_ATTRNOTSUPP (Error Code 10032)

An attribute specified is not supported by the server. This error
MUST NOT be returned by the GETATTR operation.

13.1.11.2. NFS4ERR_BADOWNER (Error Code 10039)

Returned when an owner or owner_group attribute value or the who
field of an ace within an ACL attribute value cannot be translated to
a local representation.

13.1.11.3. NFS4ERR_NOT_SAME (Error Code 10027)

This error is returned by the VERIFY operation to signify that the
attributes compared were not the same as those provided in the
client's request.

13.1.11.4. NFS4ERR_SAME (Error Code 10009)

   This error is returned by the NVERIFY operation to signify that the
   attributes compared were the same as those provided in the client's
   request.

13.1.12. Miscellaneous Errors

13.1.12.1. NFS4ERR_CB_PATH_DOWN (Error Code 10048)

   There is a problem contacting the client via the callback path.

13.2. Operations and their valid errors

   This section contains a table which gives the valid error returns for
   each protocol operation. The error code NFS4_OK (indicating no
   error) is not listed but should be understood to be returnable by all
   operations except ILLEGAL.

                Valid error returns for each protocol operation

   +-------------------+---+
   | Operation | Errors |
   +-------------------+---+
ACCESS	NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,
	NFS4ERR_BADXDR, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,
	NFS4ERR_IO, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CLOSE	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADHANDLE,
	NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,
	NFS4ERR_BADXDR, NFS4ERR_DELAY,
	NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,
	NFS4ERR_INVAL, NFS4ERR_ISDIR,
	NFS4ERR_LEASE_MOVED, NFS4ERR_LOCKS_HELD,
	NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OLD_STATEID, NFS4ERR_RESOURCE,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_STALE_STATEID
COMMIT	NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,
	NFS4ERR_BADXDR, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,
	NFS4ERR_IO, NFS4ERR_ISDIR, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE,
	NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,

```
                          | NFS4ERR_STALE, NFS4ERR_SYMLINK          |
                          |                                         |
| CREATE                  | NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,    |
                          | NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE,     |
                          | NFS4ERR_BADNAME, NFS4ERR_BADOWNER,      |
                          | NFS4ERR_BADTYPE, NFS4ERR_BADXDR,        |
                          | NFS4ERR_DELAY, NFS4ERR_DQUOT,           |
                          | NFS4ERR_EXIST, NFS4ERR_FHEXPIRED,       |
                          | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,|
                          | NFS4ERR_NAMETOOLONG, NFS4ERR_NOFILEHANDLE,|
                          | NFS4ERR_NOSPC, NFS4ERR_NOTDIR,          |
                          | NFS4ERR_PERM, NFS4ERR_RESOURCE,         |
                          | NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,      |
                          | NFS4ERR_STALE                           |
                          |                                         |
| DELEGPURGE              | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
                          | NFS4ERR_NOTSUPP, NFS4ERR_LEASE_MOVED,   |
                          | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,  |
                          | NFS4ERR_STALE_CLIENTID                  |
                          |                                         |
| DELEGRETURN             | NFS4ERR_ADMIN_REVOKED, NFS4ERR_BAD_STATEID,|
                          | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
                          | NFS4ERR_EXPIRED, NFS4ERR_INVAL,         |
                          | NFS4ERR_LEASE_MOVED, NFS4ERR_MOVED,     |
                          | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTSUPP,  |
                          | NFS4ERR_OLD_STATEID, NFS4ERR_RESOURCE,  |
                          | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,     |
                          | NFS4ERR_STALE_STATEID                   |
                          |                                         |
| GETATTR                 | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,      |
                          | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
                          | NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,       |
                          | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,|
                          | NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE, |
                          | NFS4ERR_SERVERFAULT, NFS4ERR_STALE      |
                          |                                         |
| GETFH                   | NFS4ERR_BADHANDLE, NFS4ERR_FHEXPIRED,   |
                          | NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,    |
                          | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,  |
                          | NFS4ERR_STALE                           |
                          |                                         |
| ILLEGAL                 | NFS4ERR_BADXDR, NFS4ERR_OP_ILLEGAL      |
                          |                                         |
| LINK                    | NFS4ERR_ACCESS, NFS4ERR_BADCHAR,        |
                          | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,     |
                          | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
                          | NFS4ERR_DQUOT, NFS4ERR_EXIST,           |
                          | NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN,   |
```

```
|         |  NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR,   |
|         |  NFS4ERR_MLINK, NFS4ERR_MOVED,                |
|         |  NFS4ERR_NAMETOOLONG, NFS4ERR_NOENT,          |
|         |  NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,         |
|         |  NFS4ERR_NOTDIR, NFS4ERR_NOTSUPP,             |
|         |  NFS4ERR_RESOURCE, NFS4ERR_ROFS,              |
|         |  NFS4ERR_SERVERFAULT, NFS4ERR_STALE,          |
|         |  NFS4ERR_WRONGSEC, NFS4ERR_XDEV               |
|         |                                              |
| LOCK    |  NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,       |
|         |  NFS4ERR_BADHANDLE, NFS4ERR_BAD_RANGE,        |
|         |  NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,      |
|         |  NFS4ERR_BADXDR, NFS4ERR_DEADLOCK,            |
|         |  NFS4ERR_DELAY, NFS4ERR_DENIED,               |
|         |  NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,          |
|         |  NFS4ERR_GRACE, NFS4ERR_INVAL,                |
|         |  NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,          |
|         |  NFS4ERR_LOCK_NOTSUPP, NFS4ERR_LOCK_RANGE,    |
|         |  NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,         |
|         |  NFS4ERR_NO_GRACE, NFS4ERR_OLD_STATEID,       |
|         |  NFS4ERR_OPENMODE, NFS4ERR_RECLAIM_BAD,       |
|         |  NFS4ERR_RECLAIM_CONFLICT, NFS4ERR_RESOURCE,  |
|         |  NFS4ERR_SERVERFAULT, NFS4ERR_STALE,          |
|         |  NFS4ERR_STALE_CLIENTID,                      |
|         |  NFS4ERR_STALE_STATEID                        |
|         |                                              |
| LOCKT   |  NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,           |
|         |  NFS4ERR_BAD_RANGE, NFS4ERR_BADXDR,           |
|         |  NFS4ERR_DELAY, NFS4ERR_DENIED,               |
|         |  NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,          |
|         |  NFS4ERR_GRACE, NFS4ERR_INVAL,                |
|         |  NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,          |
|         |  NFS4ERR_LOCK_RANGE, NFS4ERR_MOVED,           |
|         |  NFS4ERR_NOFILEHANDLE, NFS4ERR_RESOURCE,      |
|         |  NFS4ERR_SERVERFAULT, NFS4ERR_STALE,          |
|         |  NFS4ERR_STALE_CLIENTID                       |
|         |                                              |
| LOCKU   |  NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,       |
|         |  NFS4ERR_BADHANDLE, NFS4ERR_BAD_RANGE,        |
|         |  NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,      |
|         |  NFS4ERR_BADXDR, NFS4ERR_DELAY,               |
|         |  NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,          |
|         |  NFS4ERR_GRACE, NFS4ERR_INVAL,                |
|         |  NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,          |
|         |  NFS4ERR_LOCK_RANGE, NFS4ERR_MOVED,           |
|         |  NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID,   |
|         |  NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,       |
|         |  NFS4ERR_STALE, NFS4ERR_STALE_STATEID         |
```

```
|                       |                                        |
| LOOKUP                | NFS4ERR_ACCESS, NFS4ERR_BADCHAR,        |
|                       | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,     |
|                       | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
|                       | NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,       |
|                       | NFS4ERR_IO, NFS4ERR_MOVED,              |
|                       | NFS4ERR_NAMETOOLONG, NFS4ERR_NOENT,     |
|                       | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,   |
|                       | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,  |
|                       | NFS4ERR_STALE, NFS4ERR_SYMLINK,         |
|                       | NFS4ERR_WRONGSEC                        |
|                       |                                        |
| LOOKUPP               | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,      |
|                       | NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,       |
|                       | NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOENT,|
|                       | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,   |
|                       | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,  |
|                       | NFS4ERR_STALE, NFS4ERR_SYMLINK,         |
|                       | NFS4ERR_WRONGSEC                        |
|                       |                                        |
| NVERIFY               | NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,    |
|                       | NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE,     |
|                       | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
|                       | NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,       |
|                       | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,|
|                       | NFS4ERR_NOFILEHANDLE, NFS4ERR_SAME,     |
|                       | NFS4ERR_SERVERFAULT, NFS4ERR_STALE      |
|                       |                                        |
| OPEN                  | NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,  |
|                       | NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR,   |
|                       | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,     |
|                       | NFS4ERR_BADOWNER, NFS4ERR_BADXDR,       |
|                       | NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID, |
|                       | NFS4ERR_DELAY, NFS4ERR_DQUOT,           |
|                       | NFS4ERR_EXIST, NFS4ERR_EXPIRED,         |
|                       | NFS4ERR_FBIG, NFS4ERR_FHEXPIRED,        |
|                       | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO,|
|                       | NFS4ERR_ISDIR, NFS4ERR_MOVED,           |
|                       | NFS4ERR_NAMETOOLONG, NFS4ERR_NOENT,     |
|                       | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,    |
|                       | NFS4ERR_NOTDIR, NFS4ERR_NOTSUP,         |
|                       | NFS4ERR_NO_GRACE, NFS4ERR_OLD_STATEID,  |
|                       | NFS4ERR_PERM, NFS4ERR_RECLAIM_BAD,      |
|                       | NFS4ERR_RECLAIM_CONFLICT, NFS4ERR_RESOURCE,|
|                       | NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,      |
|                       | NFS4ERR_SHARE_DENIED, NFS4ERR_STALE,    |
|                       | NFS4ERR_STALE_CLIENTID, NFS4ERR_SYMLINK,|
|                       | NFS4ERR_WRONGSEC                        |
```

```
|                    |                                            |
|  OPENATTR          | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,         |
|                    | NFS4ERR_BADXDR, NFS4ERR_DELAY,             |
|                    | NFS4ERR_DQUOT, NFS4ERR_FHEXPIRED,          |
|                    | NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOENT,  |
|                    | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,       |
|                    | NFS4ERR_NOTSUPP, NFS4ERR_RESOURCE,         |
|                    | NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,         |
|                    | NFS4ERR_STALE                              |
|                    |                                            |
|  OPEN_CONFIRM      | NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADHANDLE,  |
|                    | NFS4ERR_BAD_SEQID, NFS4ERR_BAD_STATEID,    |
|                    | NFS4ERR_BADXDR, NFS4ERR_EXPIRED,           |
|                    | NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,          |
|                    | NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,        |
|                    | NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,       |
|                    | NFS4ERR_OLD_STATEID, NFS4ERR_RESOURCE,     |
|                    | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,        |
|                    | NFS4ERR_STALE_STATEID                      |
|                    |                                            |
|  OPEN_DOWNGRADE    | NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADHANDLE,  |
|                    | NFS4ERR_BADXDR, NFS4ERR_BAD_SEQID,         |
|                    | NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,        |
|                    | NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,        |
|                    | NFS4ERR_INVAL, NFS4ERR_LEASE_MOVED,        |
|                    | NFS4ERR_LOCKS_HELD, NFS4ERR_MOVED,         |
|                    | NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID, |
|                    | NFS4ERR_RESOURCE, NFS4ERR_ROFS,            |
|                    | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,        |
|                    | NFS4ERR_STALE_STATEID                      |
|                    |                                            |
|  PUTFH             | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,         |
|                    | NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,          |
|                    | NFS4ERR_MOVED, NFS4ERR_SERVERFAULT,        |
|                    | NFS4ERR_STALE, NFS4ERR_WRONGSEC            |
|                    |                                            |
|  PUTPUBFH          | NFS4ERR_DELAY, NFS4ERR_SERVERFAULT,        |
|                    | NFS4ERR_WRONGSEC                           |
|                    |                                            |
|  PUTROOTFH         | NFS4ERR_DELAY, NFS4ERR_SERVERFAULT,        |
|                    | NFS4ERR_WRONGSEC                           |
|                    |                                            |
|  READ              | NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,     |
|                    | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,         |
|                    | NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,        |
|                    | NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,        |
|                    | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO,  |
|                    | NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,        |
```

```
|                      | NFS4ERR_LOCKED, NFS4ERR_MOVED,          |
|                      | NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID, |
|                      | NFS4ERR_OPENMODE, NFS4ERR_RESOURCE,     |
|                      | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,     |
|                      | NFS4ERR_STALE_STATEID, NFS4ERR_SYMLINK  |
|                      |                                         |
| READDIR              | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,      |
|                      | NFS4ERR_BADXDR, NFS4ERR_BAD_COOKIE,     |
|                      | NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,       |
|                      | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, |
|                      | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,   |
|                      | NFS4ERR_NOT_SAME, NFS4ERR_RESOURCE,     |
|                      | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,     |
|                      | NFS4ERR_TOOSMALL                        |
|                      |                                         |
| READLINK             | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,      |
|                      | NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,       |
|                      | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR, |
|                      | NFS4ERR_MOVED, NFS4ERR_NOTSUP,          |
|                      | NFS4ERR_RESOURCE, NFS4ERR_NOFILEHANDLE, |
|                      | NFS4ERR_SERVERFAULT, NFS4ERR_STALE      |
|                      |                                         |
| RELEASE_LOCKOWNER    | NFS4ERR_BADXDR, NFS4ERR_EXPIRED,        |
|                      | NFS4ERR_LEASE_MOVED, NFS4ERR_LOCKS_HELD, |
|                      | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,  |
|                      | NFS4ERR_STALE_CLIENTID                  |
|                      |                                         |
| REMOVE               | NFS4ERR_ACCESS, NFS4ERR_BADCHAR,        |
|                      | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,     |
|                      | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
|                      | NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN,   |
|                      | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO, |
|                      | NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,     |
|                      | NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE,    |
|                      | NFS4ERR_NOTDIR, NFS4ERR_NOTEMPTY,       |
|                      | NFS4ERR_RESOURCE, NFS4ERR_ROFS,         |
|                      | NFS4ERR_SERVERFAULT, NFS4ERR_STALE      |
|                      |                                         |
| RENAME               | NFS4ERR_ACCESS, NFS4ERR_BADCHAR,        |
|                      | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,     |
|                      | NFS4ERR_BADXDR, NFS4ERR_DELAY,          |
|                      | NFS4ERR_DQUOT, NFS4ERR_EXIST,           |
|                      | NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN,   |
|                      | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO, |
|                      | NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,     |
|                      | NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE,    |
|                      | NFS4ERR_NOSPC, NFS4ERR_NOTDIR,          |
|                      | NFS4ERR_NOTEMPTY, NFS4ERR_RESOURCE,     |
```

```
|                     | NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,   |
|                     | NFS4ERR_STALE, NFS4ERR_WRONGSEC,     |
|                     | NFS4ERR_XDEV                         |
|                     |                                      |
| RENEW               | NFS4ERR_ACCESS, NFS4ERR_BADXDR,      |
|                     | NFS4ERR_CB_PATH_DOWN, NFS4ERR_EXPIRED, |
|                     | NFS4ERR_LEASE_MOVED, NFS4ERR_RESOURCE, |
|                     | NFS4ERR_SERVERFAULT, NFS4ERR_STALE_CLIENTID |
|                     |                                      |
| RESTOREFH           | NFS4ERR_BADHANDLE, NFS4ERR_FHEXPIRED, |
|                     | NFS4ERR_MOVED, NFS4ERR_RESOURCE,     |
|                     | NFS4ERR_RESTOREFH, NFS4ERR_SERVERFAULT, |
|                     | NFS4ERR_STALE, NFS4ERR_WRONGSEC      |
|                     |                                      |
| SAVEFH              | NFS4ERR_BADHANDLE, NFS4ERR_FHEXPIRED, |
|                     | NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, |
|                     | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT, |
|                     | NFS4ERR_STALE                        |
|                     |                                      |
| SECINFO             | NFS4ERR_ACCESS, NFS4ERR_BADCHAR,     |
|                     | NFS4ERR_BADHANDLE, NFS4ERR_BADNAME,  |
|                     | NFS4ERR_BADXDR, NFS4ERR_DELAY,       |
|                     | NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,    |
|                     | NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,  |
|                     | NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE, |
|                     | NFS4ERR_NOTDIR, NFS4ERR_RESOURCE,    |
|                     | NFS4ERR_SERVERFAULT, NFS4ERR_STALE   |
|                     |                                      |
| SETATTR             | NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, |
|                     | NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR, |
|                     | NFS4ERR_BADHANDLE, NFS4ERR_BADOWNER, |
|                     | NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, |
|                     | NFS4ERR_DELAY, NFS4ERR_DQUOT,        |
|                     | NFS4ERR_EXPIRED, NFS4ERR_FBIG,       |
|                     | NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,    |
|                     | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR, |
|                     | NFS4ERR_LEASE_MOVED, NFS4ERR_LOCKED, |
|                     | NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, |
|                     | NFS4ERR_NOSPC, NFS4ERR_OLD_STATEID,  |
|                     | NFS4ERR_OPENMODE, NFS4ERR_PERM,      |
|                     | NFS4ERR_RESOURCE, NFS4ERR_ROFS,      |
|                     | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,  |
|                     | NFS4ERR_STALE_STATEID                |
|                     |                                      |
| SETCLIENTID         | NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,  |
|                     | NFS4ERR_DELAY, NFS4ERR_INVAL,        |
|                     | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT |
|                     |                                      |
```

```
| SETCLIENTID_CONFIRM | NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,        |
|                     | NFS4ERR_DELAY, NFS4ERR_RESOURCE,           |
|                     | NFS4ERR_SERVERFAULT, NFS4ERR_STALE_CLIENTID |
|                     |                                            |
| VERIFY              | NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,       |
|                     | NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE,        |
|                     | NFS4ERR_BADXDR, NFS4ERR_DELAY,             |
|                     | NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,          |
|                     | NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,  |
|                     | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOT_SAME,    |
|                     | NFS4ERR_RESOURCE, NFS4ERR_SERVERFAULT,     |
|                     | NFS4ERR_STALE                              |
|                     |                                            |
| WRITE               | NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,     |
|                     | NFS4ERR_BADXDR, NFS4ERR_BADHANDLE,         |
|                     | NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,        |
|                     | NFS4ERR_DQUOT, NFS4ERR_EXPIRED,            |
|                     | NFS4ERR_FBIG, NFS4ERR_FHEXPIRED,           |
|                     | NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO,  |
|                     | NFS4ERR_ISDIR, NFS4ERR_LEASE_MOVED,        |
|                     | NFS4ERR_LOCKED, NFS4ERR_MOVED,             |
|                     | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,       |
|                     | NFS4ERR_NXIO, NFS4ERR_OLD_STATEID,         |
|                     | NFS4ERR_OPENMODE, NFS4ERR_RESOURCE,        |
|                     | NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,         |
|                     | NFS4ERR_STALE, NFS4ERR_STALE_STATEID,      |
|                     | NFS4ERR_SYMLINK                            |
|                     |                                            |
+---------------------+--------------------------------------------+
```

                               Table 7

13.3. Callback operations and their valid errors

   This section contains a table which gives the valid error returns for
   each callback operation. The error code NFS4_OK (indicating no
   error) is not listed but should be understood to be returnable by all
   callback operations with the exception of CB_ILLEGAL.

Valid error returns for each protocol callback operation

```
+------------+------------------------------------------------+
| Callback   | Errors                                         |
| Operation  |                                                |
+------------+------------------------------------------------+
| CB_GETATTR | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR, NFS4ERR_DELAY, |
|            | NFS4ERR_INVAL, NFS4ERR_SERVERFAULT             |
|            |                                                |
| CB_ILLEGAL | NFS4ERR_BADXDR, NFS4ERR_OP_ILLEGAL             |
|            |                                                |
| CB_RECALL  | NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,             |
|            | NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,            |
|            | NFS4ERR_SERVERFAULT                            |
|            |                                                |
+------------+------------------------------------------------+
```

Table 8

13.4. Errors and the operations that use them

Errors and the operations that use them

```
+------------------------+---------------------------------------+
| Error                  | Operations                            |
+------------------------+---------------------------------------+
| NFS4ERR_ACCESS         | ACCESS, COMMIT, CREATE, GETATTR, LINK, |
|                        | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP,  |
|                        | NVERIFY, OPEN, OPENATTR, READ,        |
|                        | READDIR, READLINK, REMOVE, RENAME,    |
|                        | RENEW, SECINFO, SETATTR, VERIFY, WRITE |
|                        |                                       |
| NFS4ERR_ADMIN_REVOKED  | CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, |
|                        | OPEN_CONFIRM, OPEN_DOWNGRADE, READ,   |
|                        | SETATTR, WRITE                        |
|                        |                                       |
| NFS4ERR_ATTRNOTSUPP    | CREATE, NVERIFY, OPEN, SETATTR, VERIFY |
|                        |                                       |
| NFS4ERR_BADCHAR        | CREATE, LINK, LOOKUP, NVERIFY, OPEN,  |
|                        | REMOVE, RENAME, SECINFO, SETATTR,     |
|                        | VERIFY                                |
|                        |                                       |
| NFS4ERR_BADHANDLE      | ACCESS, CB_GETATTR, CB_RECALL, CLOSE, |
|                        | COMMIT, CREATE, GETATTR, GETFH, LINK, |
|                        | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP,  |
|                        | NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, |
|                        | OPEN_DOWNGRADE, PUTFH, READ, READDIR, |
|                        | READLINK, REMOVE, RENAME, RESTOREFH,  |
```

	SAVEFH, SECINFO, SETATTR, VERIFY, WRITE
NFS4ERR_BADNAME	CREATE, LINK, LOOKUP, OPEN, REMOVE, RENAME, SECINFO
NFS4ERR_BADOWNER	CREATE, OPEN, SETATTR
NFS4ERR_BADTYPE	CREATE
NFS4ERR_BADXDR	ACCESS, CB_GETATTR, CB_ILLEGAL, CB_RECALL, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, ILLEGAL, LINK, LOCK, LOCKT, LOCKU, LOOKUP, NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, OPEN_DOWNGRADE, PUTFH, READ, READDIR, RELEASE_LOCKOWNER, REMOVE, RENAME, RENEW, SECINFO, SETATTR, SETCLIENTID, SETCLIENTID_CONFIRM, VERIFY, WRITE
NFS4ERR_BAD_COOKIE	READDIR
NFS4ERR_BAD_RANGE	LOCK, LOCKT, LOCKU
NFS4ERR_BAD_SEQID	CLOSE, LOCK, LOCKU, OPEN, OPEN_CONFIRM, OPEN_DOWNGRADE
NFS4ERR_BAD_STATEID	CB_RECALL, CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, SETATTR, WRITE
NFS4ERR_CB_PATH_DOWN	RENEW
NFS4ERR_CLID_INUSE	SETCLIENTID, SETCLIENTID_CONFIRM
NFS4ERR_DEADLOCK	LOCK
NFS4ERR_DELAY	ACCESS, CB_GETATTR, CB_RECALL, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, REMOVE, RENAME, SECINFO, SETATTR, SETCLIENTID, SETCLIENTID_CONFIRM, VERIFY, WRITE

NFS4ERR_DENIED	LOCK, LOCKT
NFS4ERR_DQUOT	CREATE, LINK, OPEN, OPENATTR, RENAME, SETATTR, WRITE
NFS4ERR_EXIST	CREATE, LINK, OPEN, RENAME
NFS4ERR_EXPIRED	CLOSE, DELEGRETURN, LOCK, LOCKT, LOCKU, OPEN, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, RELEASE_LOCKOWNER, RENEW, SETATTR, WRITE
NFS4ERR_FBIG	OPEN, SETATTR, WRITE
NFS4ERR_FHEXPIRED	ACCESS, CLOSE, COMMIT, CREATE, GETATTR, GETFH, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SETATTR, VERIFY, WRITE
NFS4ERR_FILE_OPEN	LINK, REMOVE, RENAME
NFS4ERR_GRACE	GETATTR, LOCK, LOCKT, LOCKU, NVERIFY, OPEN, READ, REMOVE, RENAME, SETATTR, VERIFY, WRITE
NFS4ERR_INVAL	ACCESS, CB_GETATTR, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, LINK, LOCK, LOCKT, LOCKU, LOOKUP, NVERIFY, OPEN, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, READDIR, READLINK, REMOVE, RENAME, SECINFO, SETATTR, SETCLIENTID, VERIFY, WRITE
NFS4ERR_IO	ACCESS, COMMIT, CREATE, GETATTR, LINK, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, READ, READDIR, READLINK, REMOVE, RENAME, SETATTR, VERIFY, WRITE
NFS4ERR_ISDIR	CLOSE, COMMIT, LINK, LOCK, LOCKT, LOCKU, OPEN, OPEN_CONFIRM, READ, READLINK, SETATTR, WRITE

NFS4ERR_LEASE_MOVED	CLOSE, DELEGPURGE, DELEGRETURN, LOCK, LOCKT, LOCKU, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, RELEASE_LOCKOWNER, RENEW, SETATTR, WRITE
NFS4ERR_LOCKED	READ, SETATTR, WRITE
NFS4ERR_LOCKS_HELD	CLOSE, OPEN_DOWNGRADE, RELEASE_LOCKOWNER
NFS4ERR_LOCK_NOTSUPP	LOCK
NFS4ERR_LOCK_RANGE	LOCK, LOCKT, LOCKU
NFS4ERR_MLINK	LINK
NFS4ERR_MOVED	ACCESS, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, GETFH, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SETATTR, VERIFY, WRITE
NFS4ERR_NAMETOOLONG	CREATE, LINK, LOOKUP, OPEN, REMOVE, RENAME, SECINFO
NFS4ERR_NOENT	LINK, LOOKUP, LOOKUPP, OPEN, OPENATTR, REMOVE, RENAME, SECINFO
NFS4ERR_NOFILEHANDLE	ACCESS, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, GETFH, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, OPEN_DOWNGRADE, READ, READDIR, READLINK, REMOVE, RENAME, SAVEFH, SECINFO, SETATTR, VERIFY, WRITE
NFS4ERR_NOSPC	CREATE, LINK, OPEN, OPENATTR, RENAME, SETATTR, WRITE
NFS4ERR_NOTDIR	CREATE, LINK, LOOKUP, LOOKUPP, OPEN, READDIR, REMOVE, RENAME, SECINFO
NFS4ERR_NOTEMPTY	REMOVE, RENAME

```
    | NFS4ERR_NOTSUP            | OPEN, READLINK                    |
    |                          |                                   |
    | NFS4ERR_NOTSUPP          | DELEGPURGE, DELEGRETURN, LINK,     |
    |                          | OPENATTR                          |
    |                          |                                   |
    | NFS4ERR_NOT_SAME         | READDIR, VERIFY                   |
    |                          |                                   |
    | NFS4ERR_NO_GRACE         | LOCK, OPEN                        |
    |                          |                                   |
    | NFS4ERR_NXIO             | WRITE                             |
    |                          |                                   |
    | NFS4ERR_OLD_STATEID      | CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, |
    |                          | OPEN_CONFIRM, OPEN_DOWNGRADE, READ, |
    |                          | SETATTR, WRITE                    |
    |                          |                                   |
    | NFS4ERR_OPENMODE         | LOCK, READ, SETATTR, WRITE        |
    |                          |                                   |
    | NFS4ERR_OP_ILLEGAL       | CB_ILLEGAL, ILLEGAL               |
    |                          |                                   |
    | NFS4ERR_PERM             | CREATE, OPEN, SETATTR             |
    |                          |                                   |
    | NFS4ERR_RECLAIM_BAD      | LOCK, OPEN                        |
    |                          |                                   |
    | NFS4ERR_RECLAIM_CONFLICT | LOCK, OPEN                        |
    |                          |                                   |
    | NFS4ERR_RESOURCE         | ACCESS, CLOSE, COMMIT, CREATE,    |
    |                          | DELEGPURGE, DELEGRETURN, GETATTR, |
    |                          | GETFH, LINK, LOCK, LOCKT, LOCKU,  |
    |                          | LOOKUP, LOOKUPP, OPEN, OPENATTR,  |
    |                          | OPEN_CONFIRM, OPEN_DOWNGRADE, READ, |
    |                          | READDIR, READLINK, RELEASE_LOCKOWNER, |
    |                          | REMOVE, RENAME, RENEW, RESTOREFH, |
    |                          | SAVEFH, SECINFO, SETATTR, SETCLIENTID, |
    |                          | SETCLIENTID_CONFIRM, VERIFY, WRITE |
    |                          |                                   |
    | NFS4ERR_RESTOREFH        | RESTOREFH                         |
    |                          |                                   |
    | NFS4ERR_ROFS             | COMMIT, CREATE, LINK, OPEN, OPENATTR, |
    |                          | OPEN_DOWNGRADE, REMOVE, RENAME,    |
    |                          | SETATTR, WRITE                    |
    |                          |                                   |
    | NFS4ERR_SAME             | NVERIFY                           |
    |                          |                                   |
    | NFS4ERR_SERVERFAULT      | ACCESS, CB_GETATTR, CB_RECALL, CLOSE, |
    |                          | COMMIT, CREATE, DELEGPURGE,        |
    |                          | DELEGRETURN, GETATTR, GETFH, LINK, |
    |                          | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, |
    |                          | NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, |
```

```
|                           | OPEN_DOWNGRADE, PUTFH, PUTPUBFH, |
|                           | PUTROOTFH, READ, READDIR, READLINK, |
|                           | RELEASE_LOCKOWNER, REMOVE, RENAME, |
|                           | RENEW, RESTOREFH, SAVEFH, SECINFO, |
|                           | SETATTR, SETCLIENTID, |
|                           | SETCLIENTID_CONFIRM, VERIFY, WRITE |
|                           | |
| NFS4ERR_SHARE_DENIED      | OPEN |
|                           | |
| NFS4ERR_STALE             | ACCESS, CLOSE, COMMIT, CREATE, |
|                           | DELEGRETURN, GETATTR, GETFH, LINK, |
|                           | LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, |
|                           | NVERIFY, OPEN, OPENATTR, OPEN_CONFIRM, |
|                           | OPEN_DOWNGRADE, PUTFH, READ, READDIR, |
|                           | READLINK, REMOVE, RENAME, RESTOREFH, |
|                           | SAVEFH, SECINFO, SETATTR, VERIFY, |
|                           | WRITE |
|                           | |
| NFS4ERR_STALE_CLIENTID    | DELEGPURGE, LOCK, LOCKT, OPEN, |
|                           | RELEASE_LOCKOWNER, RENEW, |
|                           | SETCLIENTID_CONFIRM |
|                           | |
| NFS4ERR_STALE_STATEID     | CLOSE, DELEGRETURN, LOCK, LOCKU, |
|                           | OPEN_CONFIRM, OPEN_DOWNGRADE, READ, |
|                           | SETATTR, WRITE |
|                           | |
| NFS4ERR_SYMLINK           | COMMIT, LOOKUP, LOOKUPP, OPEN, READ, |
|                           | WRITE |
|                           | |
| NFS4ERR_TOOSMALL          | READDIR |
|                           | |
| NFS4ERR_WRONGSEC          | LINK, LOOKUP, LOOKUPP, OPEN, PUTFH, |
|                           | PUTPUBFH, PUTROOTFH, RENAME, RESTOREFH |
|                           | |
| NFS4ERR_XDEV              | LINK, RENAME |
|                           | |
+---------------------------+-------------------------------------+
```

                              Table 9

14. NFSv4 Requests

   For the NFSv4 RPC program, there are two traditional RPC procedures:
   NULL and COMPOUND. All other functionality is defined as a set of
   operations and these operations are defined in normal XDR/RPC syntax
   and semantics. However, these operations are encapsulated within the
   COMPOUND procedure. This requires that the client combine one or
   more of the NFSv4 operations into a single request.

The NFS4_CALLBACK program is used to provide server to client
signaling and is constructed in a similar fashion as the NFSv4
program. The procedures CB_NULL and CB_COMPOUND are defined in the
same way as NULL and COMPOUND are within the NFS program. The
CB_COMPOUND request also encapsulates the remaining operations of the
NFS4_CALLBACK program. There is no predefined RPC program number for
the NFS4_CALLBACK program. It is up to the client to specify a
program number in the "transient" program range. The program and
port number of the NFS4_CALLBACK program are provided by the client
as part of the SETCLIENTID/SETCLIENTID_CONFIRM sequence. The program
and port can be changed by another SETCLIENTID/SETCLIENTID_CONFIRM
sequence, and it is possible to use the sequence to change them
within a client incarnation without removing relevant leased client
state.

14.1. Compound Procedure

The COMPOUND procedure provides the opportunity for better
performance within high latency networks. The client can avoid
cumulative latency of multiple RPCs by combining multiple dependent
operations into a single COMPOUND procedure. A compound operation
may provide for protocol simplification by allowing the client to
combine basic procedures into a single request that is customized for
the client's environment.

The CB_COMPOUND procedure precisely parallels the features of
COMPOUND as described above.

The basic structure of the COMPOUND procedure is:

```
+-----+--------------+--------+-----------+-----------+-----------+--
| tag | minorversion | numops | op + args | op + args | op + args |
+-----+--------------+--------+-----------+-----------+-----------+--
```

and the reply's structure is:

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

The numops and numres fields, used in the depiction above, represent
the count for the counted array encoding use to signify the number of
arguments or results encoded in the request and response. As per the
XDR encoding, these counts must match exactly the number of operation
arguments or results encoded.

14.2. Evaluation of a Compound Request

   The server will process the COMPOUND procedure by evaluating each of
   the operations within the COMPOUND procedure in order. Each
   component operation consists of a 32 bit operation code, followed by
   the argument of length determined by the type of operation. The
   results of each operation are encoded in sequence into a reply
   buffer. The results of each operation are preceded by the opcode and
   a status code (normally zero). If an operation results in a non-zero
   status code, the status will be encoded and evaluation of the
   compound sequence will halt and the reply will be returned. Note
   that evaluation stops even in the event of "non error" conditions
   such as NFS4ERR_SAME.

   There are no atomicity requirements for the operations contained
   within the COMPOUND procedure. The operations being evaluated as
   part of a COMPOUND request may be evaluated simultaneously with other
   COMPOUND requests that the server receives.

   A COMPOUND is not a transaction and it is the client's responsibility
   for recovering from any partially completed COMPOUND procedure.
   These may occur at any point due to errors such as NFS4ERR_RESOURCE
   and NFS4ERR_DELAY. Note that these errors can occur in an otherwise
   valid operation string. Further, a server reboot which occurs in the
   middle of processing a COMPOUND procedure may leave the client with
   the difficult task of determining how far COMPOUND processing has
   proceeded. Therefore, the client should avoid overly complex
   COMPOUND procedures in the event of the failure of an operation
   within the procedure.

   Each operation assumes a "current" and "saved" filehandle that is
   available as part of the execution context of the compound request.
   Operations may set, change, or return the current filehandle. The
   "saved" filehandle is used for temporary storage of a filehandle
   value and as operands for the RENAME and LINK operations.

14.3. Synchronous Modifying Operations

   NFSv4 operations that modify the file system are synchronous. When
   an operation is successfully completed at the server, the client can
   depend that any data associated with the request is now on stable
   storage (the one exception is in the case of the file data in a WRITE
   operation with the UNSTABLE4 option specified).

   This implies that any previous operations within the same compound
   request are also reflected in stable storage. This behavior enables
   the client's ability to recover from a partially executed compound
   request which may resulted from the failure of the server. For

example, if a compound request contains operations A and B and the
server is unable to send a response to the client, depending on the
progress the server made in servicing the request the result of both
operations may be reflected in stable storage or just operation A may
be reflected. The server must not have just the results of operation
B in stable storage.

14.4. Operation Values

The operations encoded in the COMPOUND procedure are identified by
operation values. To avoid overlap with the RPC procedure numbers,
operations 0 (zero) and 1 are not defined. Operation 2 is not
defined but reserved for future use with minor versioning.

15. NFSv4 Procedures

[RFC Editor: prior to publishing this document as an RFC, please have
every Section that has a title of "Procedure X:" or "Operation Y:"
start at the top of a new page.]

15.1. Procedure 0: NULL - No Operation

15.1.1. SYNOPSIS

   <null>

15.1.2. ARGUMENT

      void;

15.1.3. RESULT

      void;

15.1.4. DESCRIPTION

   Standard NULL procedure. Void argument, void response. This
   procedure has no functionality associated with it. Because of this
   it is sometimes used to measure the overhead of processing a service
   request. Therefore, the server should ensure that no unnecessary
   work is done in servicing this procedure.

15.2. Procedure 1: COMPOUND - Compound Operations

15.2.1. SYNOPSIS

    compoundargs -> compoundres

15.2.2. ARGUMENT

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

    struct COMPOUND4args {
            utf8str_cs   tag;
            uint32_t   minorversion;
            nfs_argop4   argarray<>;
    };

15.2.3. RESULT

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

    struct COMPOUND4res {
            nfsstat4   status;
            utf8str_cs   tag;
            nfs_resop4   resarray<>;
    };

15.2.4. DESCRIPTION

   The COMPOUND procedure is used to combine one or more of the NFS
   operations into a single RPC request. The main NFS RPC program has
   two main procedures: NULL and COMPOUND. All other operations use the
   COMPOUND procedure as a wrapper.

   The COMPOUND procedure is used to combine individual operations into
   a single RPC request. The server interprets each of the operations
   in turn. If an operation is executed by the server and the status of
   that operation is NFS4_OK, then the next operation in the COMPOUND
   procedure is executed. The server continues this process until there
   are no more operations to be executed or one of the operations has a
   status value other than NFS4_OK.

   In the processing of the COMPOUND procedure, the server may find that
   it does not have the available resources to execute any or all of the
   operations within the COMPOUND sequence. In this case, the error

NFS4ERR_RESOURCE will be returned for the particular operation within
the COMPOUND procedure where the resource exhaustion occurred. This
assumes that all previous operations within the COMPOUND sequence
have been evaluated successfully. The results for all of the
evaluated operations must be returned to the client.

The server will generally choose between two methods of decoding the
client's request. The first would be the traditional one-pass XDR
decode, in which decoding of the entire COMPOUND precedes execution
of any operation within it. If there is an XDR decoding error in
this case, an RPC XDR decode error would be returned. The second
method would be to make an initial pass to decode the basic COMPOUND
request and then to XDR decode each of the individual operations, as
the server is ready to execute it. In this case, the server may
encounter an XDR decode error during such an operation decode, after
previous operations within the COMPOUND have been executed. In this
case, the server would return the error NFS4ERR_BADXDR to signify the
decode error.

The COMPOUND arguments contain a "minorversion" field. The initial
and default value for this field is 0 (zero). This field will be
used by future minor versions such that the client can communicate to
the server what minor version is being requested. If the server
receives a COMPOUND procedure with a minorversion field value that it
does not support, the server MUST return an error of
NFS4ERR_MINOR_VERS_MISMATCH and a zero length resultdata array.

Contained within the COMPOUND results is a "status" field. If the
results array length is non-zero, this status must be equivalent to
the status of the last operation that was executed within the
COMPOUND procedure. Therefore, if an operation incurred an error
then the "status" value will be the same error value as is being
returned for the operation that failed.

Note that operations, 0 (zero), 1 (one), and 2 (two) are not defined
for the COMPOUND procedure. It is possible that the server receives
a request that contains an operation that is less than the first
legal operation (OP_ACCESS) or greater than the last legal operation
(OP_RELEASE_LOCKOWNER). In this case, the server's response will
encode the opcode OP_ILLEGAL rather than the illegal opcode of the
request. The status field in the ILLEGAL return results will set to
NFS4ERR_OP_ILLEGAL. The COMPOUND procedure's return results will
also be NFS4ERR_OP_ILLEGAL.

The definition of the "tag" in the request is left to the
implementer. It may be used to summarize the content of the compound
request for the benefit of packet sniffers and engineers debugging
implementations. However, the value of "tag" in the response SHOULD

be the same value as provided in the request. This applies to the
tag field of the CB_COMPOUND procedure as well.

15.2.4.1. Current Filehandle

The current and saved filehandle are used throughout the protocol.
Most operations implicitly use the current filehandle as a argument
and many set the current filehandle as part of the results. The
combination of client specified sequences of operations and current
and saved filehandle arguments and results allows for greater
protocol flexibility. The best or easiest example of current
filehandle usage is a sequence like the following:

```
  PUTFH fh1                 {fh1}
  LOOKUP "compA"            {fh2}
  GETATTR                   {fh2}
  LOOKUP "compB"            {fh3}
  GETATTR                   {fh3}
  LOOKUP "compC"            {fh4}
  GETATTR                   {fh4}
  GETFH
```

                             Figure 1

In this example, the PUTFH (Section 15.22) operation explicitly sets
the current filehandle value while the result of each LOOKUP
operation sets the current filehandle value to the resultant file
system object. Also, the client is able to insert GETATTR operations
using the current filehandle as an argument.

The PUTROOTFH (Section 15.24) and PUTPUBFH (Section 15.24) operations
also set the current filehandle. The above example would replace
"PUTFH fh1" with PUTROOTFH or PUTPUBFH with no filehandle argument in
order to achieve the same effect (on the assumption that "compA" is
directly below the root of the namespace).

Along with the current filehandle, there is a saved filehandle.
While the current filehandle is set as the result of operations like
LOOKUP, the saved filehandle must be set directly with the use of the
SAVEFH operation. The SAVEFH operations copies the current
filehandle value to the saved value. The saved filehandle value is
used in combination with the current filehandle value for the LINK
and RENAME operations. The RESTOREFH operation will copy the saved
filehandle value to the current filehandle value; as a result, the
saved filehandle value may be used a sort of "scratch" area for the
client's series of operations.

15.2.5. IMPLEMENTATION

   Since an error of any type may occur after only a portion of the
   operations have been evaluated, the client must be prepared to
   recover from any failure. If the source of an NFS4ERR_RESOURCE error
   was a complex or lengthy set of operations, it is likely that if the
   number of operations were reduced the server would be able to
   evaluate them successfully. Therefore, the client is responsible for
   dealing with this type of complexity in recovery.

   A single compound should not contain multiple operations that have
   different values for the clientid field used in OPEN, LOCK, RENEW.
   This can cause confusion in cases in which operations that do not
   contain clientids have potential interactions with operations that
   do. When only a single clientid has been used, it is clear what
   client is being referenced. For a particular example involving the
   interaction of OPEN and GETATTR, see Section 15.18.6.

15.3. Operation 3: ACCESS - Check Access Rights

15.3.1. SYNOPSIS

     (cfh), accessreq -> supported, accessrights

15.3.2. ARGUMENT

   const ACCESS4_READ   = 0x00000001;
   const ACCESS4_LOOKUP   = 0x00000002;
   const ACCESS4_MODIFY   = 0x00000004;
   const ACCESS4_EXTEND   = 0x00000008;
   const ACCESS4_DELETE   = 0x00000010;
   const ACCESS4_EXECUTE   = 0x00000020;

   struct ACCESS4args {
           /* CURRENT_FH: object */
           uint32_t   access;
   };

15.3.3. RESULT

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

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

15.3.4. DESCRIPTION

   ACCESS determines the access rights that a user, as identified by the
   credentials in the RPC request, has with respect to the file system
   object specified by the current filehandle. The client encodes the
   set of access rights that are to be checked in the bit mask "access".
   The server checks the permissions encoded in the bit mask. If a
   status of NFS4_OK is returned, two bit masks are included in the
   response. The first, "supported", represents the access rights for
   which the server can verify reliably. The second, "access",
   represents the access rights available to the user for the filehandle
   provided. On success, the current filehandle retains its value.

   Note that the supported field will contain only as many values as
   were originally sent in the arguments. For example, if the client
   sends an ACCESS operation with only the ACCESS4_READ value set and
   the server supports this value, the server will return only
   ACCESS4_READ even if it could have reliably checked other values.

   The results of this operation are necessarily advisory in nature. A
   return status of NFS4_OK and the appropriate bit set in the bit mask
   does not imply that such access will be allowed to the file system
   object in the future. This is because access rights can be revoked
   by the server at any time.

   The following access permissions may be requested:

   ACCESS4_READ: Read data from file or read a directory.

   ACCESS4_LOOKUP: Look up a name in a directory (no meaning for non-
      directory objects).

   ACCESS4_MODIFY: Rewrite existing file data or modify existing
      directory entries.

ACCESS4_EXTEND: Write new data or add directory entries.

ACCESS4_DELETE: Delete an existing directory entry.

ACCESS4_EXECUTE: Execute file (no meaning for a directory).

On success, the current filehandle retains its value.

15.3.5. IMPLEMENTATION

In general, it is not sufficient for the client to attempt to deduce
access permissions by inspecting the uid, gid, and mode fields in the
file attributes or by attempting to interpret the contents of the ACL
attribute. This is because the server may perform uid or gid mapping
or enforce additional access control restrictions. It is also
possible that the server may not be in the same ID space as the
client. In these cases (and perhaps others), the client cannot
reliably perform an access check with only current file attributes.

In the NFSv2 protocol, the only reliable way to determine whether an
operation was allowed was to try it and see if it succeeded or
failed. Using the ACCESS operation in the NFSv4 protocol, the client
can ask the server to indicate whether or not one or more classes of
operations are permitted. The ACCESS operation is provided to allow
clients to check before doing a series of operations which might
result in an access failure. The OPEN operation provides a point
where the server can verify access to the file object and method to
return that information to the client. The ACCESS operation is still
useful for directory operations or for use in the case the UNIX API
"access" is used on the client.

The information returned by the server in response to an ACCESS call
is not permanent. It was correct at the exact time that the server
performed the checks, but not necessarily afterward. The server can
revoke access permission at any time.

The client should use the effective credentials of the user to build
the authentication information in the ACCESS request used to
determine access rights. It is the effective user and group
credentials that are used in subsequent read and write operations.

Many implementations do not directly support the ACCESS4_DELETE
permission. Operating systems like UNIX will ignore the
ACCESS4_DELETE bit if set on an access request on a non-directory
object. In these systems, delete permission on a file is determined
by the access permissions on the directory in which the file resides,
instead of being determined by the permissions of the file itself.
Therefore, the mask returned enumerating which access rights can be

supported will have the ACCESS4_DELETE value set to 0. This
indicates to the client that the server was unable to check that
particular access right. The ACCESS4_DELETE bit in the access mask
returned will then be ignored by the client.

15.4. Operation 4: CLOSE - Close File

15.4.1. SYNOPSIS

   (cfh), seqid, open_stateid -> open_stateid

15.4.2. ARGUMENT

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

15.4.3. RESULT

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

15.4.4. DESCRIPTION

   The CLOSE operation releases share reservations for the regular or
   named attribute file as specified by the current filehandle. The
   share reservations and other state information released at the server
   as a result of this CLOSE is only associated with the supplied
   stateid. The sequence id provides for the correct ordering. State
   associated with other OPENs is not affected.

   If byte-range locks are held, the client SHOULD release all locks
   before issuing a CLOSE. The server MAY free all outstanding locks on
   CLOSE but some servers may not support the CLOSE of a file that still
   has byte-range locks held. The server MUST return failure if any
   locks would exist after the CLOSE.

   On success, the current filehandle retains its value.

15.4.5. IMPLEMENTATION

   Even though CLOSE returns a stateid, this stateid is not useful to
   the client and should be treated as deprecated. CLOSE "shuts down"
   the state associated with all OPENs for the file by a single open-
   owner. As noted above, CLOSE will either release all file locking
   state or return an error. Therefore, the stateid returned by CLOSE
   is not useful for operations that follow.

15.5. Operation 5: COMMIT - Commit Cached Data

15.5.1. SYNOPSIS

      (cfh), offset, count -> verifier

15.5.2. ARGUMENT

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

15.5.3. RESULT

   struct COMMIT4resok {
           verifier4   writeverf;
   };

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

15.5.4. DESCRIPTION

   The COMMIT operation forces or flushes data to stable storage for the
   file specified by the current filehandle. The flushed data is that
   which was previously written with a WRITE operation which had the
   stable field set to UNSTABLE4.

   The offset specifies the position within the file where the flush is
   to begin. An offset value of 0 (zero) means to flush data starting
   at the beginning of the file. The count specifies the number of

bytes of data to flush. If count is 0 (zero), a flush from offset to
the end of the file is done.

The server returns a write verifier upon successful completion of the
COMMIT. The write verifier is used by the client to determine if the
server has restarted or rebooted between the initial WRITE(s) and the
COMMIT. The client does this by comparing the write verifier
returned from the initial writes and the verifier returned by the
COMMIT operation. The server must vary the value of the write
verifier at each server event or instantiation that may lead to a
loss of uncommitted data. Most commonly this occurs when the server
is rebooted; however, other events at the server may result in
uncommitted data loss as well.

On success, the current filehandle retains its value.

15.5.5. IMPLEMENTATION

The COMMIT operation is similar in operation and semantics to the
POSIX fsync() [fsync] system call that synchronizes a file's state
with the disk (file data and metadata is flushed to disk or stable
storage). COMMIT performs the same operation for a client, flushing
any unsynchronized data and metadata on the server to the server's
disk or stable storage for the specified file. Like fsync(), it may
be that there is some modified data or no modified data to
synchronize. The data may have been synchronized by the server's
normal periodic buffer synchronization activity. COMMIT should
return NFS4_OK, unless there has been an unexpected error.

COMMIT differs from fsync() in that it is possible for the client to
flush a range of the file (most likely triggered by a buffer-
reclamation scheme on the client before file has been completely
written).

The server implementation of COMMIT is reasonably simple. If the
server receives a full file COMMIT request, that is starting at
offset 0 and count 0, it should do the equivalent of fsync()'ing the
file. Otherwise, it should arrange to have the cached data in the
range specified by offset and count to be flushed to stable storage.
In both cases, any metadata associated with the file must be flushed
to stable storage before returning. It is not an error for there to
be nothing to flush on the server. This means that the data and
metadata that needed to be flushed have already been flushed or lost
during the last server failure.

The client implementation of COMMIT is a little more complex. There
are two reasons for wanting to commit a client buffer to stable
storage. The first is that the client wants to reuse a buffer. In

this case, the offset and count of the buffer are sent to the server
in the COMMIT request. The server then flushes any cached data based
on the offset and count, and flushes any metadata associated with the
file. It then returns the status of the flush and the write
verifier. The other reason for the client to generate a COMMIT is
for a full file flush, such as may be done at close. In this case,
the client would gather all of the buffers for this file that contain
uncommitted data, do the COMMIT operation with an offset of 0 and
count of 0, and then free all of those buffers. Any other dirty
buffers would be sent to the server in the normal fashion.

After a buffer is written by the client with the stable parameter set
to UNSTABLE4, the buffer must be considered as modified by the client
until the buffer has either been flushed via a COMMIT operation or
written via a WRITE operation with stable parameter set to FILE_SYNC4
or DATA_SYNC4. This is done to prevent the buffer from being freed
and reused before the data can be flushed to stable storage on the
server.

When a response is returned from either a WRITE or a COMMIT operation
and it contains a write verifier that is different than previously
returned by the server, the client will need to retransmit all of the
buffers containing uncommitted cached data to the server. How this
is to be done is up to the implementer. If there is only one buffer
of interest, then it should probably be sent back over in a WRITE
request with the appropriate stable parameter. If there is more than
one buffer, it might be worthwhile retransmitting all of the buffers
in WRITE requests with the stable parameter set to UNSTABLE4 and then
retransmitting the COMMIT operation to flush all of the data on the
server to stable storage. The timing of these retransmissions is
left to the implementer.

The above description applies to page-cache-based systems as well as
buffer-cache-based systems. In those systems, the virtual memory
system will need to be modified instead of the buffer cache.

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

15.6.1. SYNOPSIS

   (cfh), name, type, attrs -> (cfh), cinfo, attrset

15.6.2. ARGUMENT

```
   union createtype4 switch (nfs_ftype4 type) {
    case NF4LNK:
            linktext4 linkdata;
    case NF4BLK:
    case NF4CHR:
            specdata4 devdata;
    case NF4SOCK:
    case NF4FIFO:
    case NF4DIR:
            void;
    default:
            void;  /* server should return NFS4ERR_BADTYPE */
   };

   struct CREATE4args {
           /* CURRENT_FH: directory for creation */
           createtype4     objtype;
           component4      objname;
           fattr4          createattrs;
   };
```

15.6.3. RESULT

```
   struct CREATE4resok {
           change_info4    cinfo;
           bitmap4         attrset;         /* attributes set */
   };

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

15.6.4. DESCRIPTION

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

   The objname specifies the name for the new object. The objtype
   determines the type of object to be created: directory, symlink, etc.

   If an object of the same name already exists in the directory, the
   server will return the error NFS4ERR_EXIST.

For the directory where the new file object was created, the server returns change_info4 information in cinfo. With the atomic field of the change_info4 struct, the server will indicate if the before and after change attributes were obtained atomically with respect to the file object creation.

If the objname is of zero length, NFS4ERR_INVAL will be returned. The objname is also subject to the normal UTF-8, character support, and name checks. See Section 12.7 for further discussion.

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

The createattrs specifies the initial set of attributes for the object. The set of attributes may include any writable attribute valid for the object type. When the operation is successful, the server will return to the client an attribute mask signifying which attributes were successfully set for the object.

If createattrs includes neither the owner attribute nor an ACL with an ACE for the owner, and if the server's file system both supports and requires an owner attribute (or an owner ACE) then the server MUST derive the owner (or the owner ACE). This would typically be from the principal indicated in the RPC credentials of the call, but the server's operating environment or file system semantics may dictate other methods of derivation. Similarly, if createattrs includes neither the group attribute nor a group ACE, and if the server's file system both supports and requires the notion of a group attribute (or group ACE), the server MUST derive the group attribute (or the corresponding owner ACE) for the file. This could be from the RPC call's credentials, such as the group principal if the credentials include it (such as with AUTH_SYS), from the group identifier associated with the principal in the credentials (e.g., POSIX systems have a user database [getpwnam] that has the group identifier for every user identifier), inherited from directory the object is created in, or whatever else the server's operating environment or file system semantics dictate. This applies to the OPEN operation too.

Conversely, it is possible the client will specify in createattrs an owner attribute or group attribute or ACL that the principal indicated the RPC call's credentials does not have permissions to create files for. The error to be returned in this instance is NFS4ERR_PERM. This applies to the OPEN operation too.

15.6.5. IMPLEMENTATION

   If the client desires to set attribute values after the create, a
   SETATTR operation can be added to the COMPOUND request so that the
   appropriate attributes will be set.

15.7. Operation 7: DELEGPURGE - Purge Delegations Awaiting Recovery

15.7.1. SYNOPSIS

     clientid ->

15.7.2. ARGUMENT

   struct DELEGPURGE4args {
           clientid4   clientid;
   };

15.7.3. RESULT

   struct DELEGPURGE4res {
           nfsstat4   status;
   };

15.7.4. DESCRIPTION

   Purges all of the delegations awaiting recovery for a given client.
   This is useful for clients which do not commit delegation information
   to stable storage to indicate that conflicting requests need not be
   delayed by the server awaiting recovery of delegation information.

   This operation is provided to support clients that record delegation
   information on stable storage on the client. In this case,
   DELEGPURGE should be issued immediately after doing delegation
   recovery (using CLAIM_DELEGATE_PREV) on all delegations known to the
   client. Doing so will notify the server that no additional
   delegations for the client will be recovered allowing it to free
   resources, and avoid delaying other clients who make requests that
   conflict with the unrecovered delegations. All client SHOULD use
   DELEGPURGE as part of recovery once it is known that no further
   CLAIM_DELEGATE_PREV recovery will be done. This includes clients
   that do not record delegation information on stable storage, who
   would then do a DELEGPURGE immediately after SETCLIENTID_CONFIRM.

   The set of delegations known to the server and the client may be
   different. The reasons for this include:

o A client may fail after making a request which resulted in
   delegation but before it received the results and committed them
   to the client's stable storage.

o A client may fail after deleting its indication that a delegation
   exists but before the delegation return is fully processed by the
   server.

o In the case in which the server and the client restart, the server
   may have limited persistent recording of delegation to a subset of
   those in existence.

o A client may have only persistently recorded information about a
   subset of delegations.

The server MAY support DELEGPURGE, but its support or non-support
should match that of CLAIM_DELEGATE_PREV:

o A server may support both DELEGPURGE and CLAIM_DELEGATE_PREV.

o A server may support neither DELEGPURGE nor CLAIM_DELEGATE_PREV.

This fact allows a client starting up to determine if the server is
prepared to support persistent storage of delegation information and
thus whether it may use write-back caching to local persistent
storage, relying on CLAIM_DELEGATE_PREV recovery to allow such
changed data to be flushed safely to the server in the event of
client restart.

15.8. Operation 8: DELEGRETURN - Return Delegation

15.8.1. SYNOPSIS

   (cfh), stateid ->

15.8.2. ARGUMENT

   struct DELEGRETURN4args {
           /* CURRENT_FH: delegated file */
           stateid4   deleg_stateid;
   };

15.8.3. RESULT

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

15.8.4. DESCRIPTION

   Returns the delegation represented by the current filehandle and
   stateid.

   Delegations may be returned when recalled or voluntarily (i.e.,
   before the server has recalled them). In either case the client must
   properly propagate state changed under the context of the delegation
   to the server before returning the delegation.

15.9. Operation 9: GETATTR - Get Attributes

15.9.1. SYNOPSIS

      (cfh), attrbits -> attrbits, attrvals

15.9.2. ARGUMENT

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

15.9.3. RESULT

```
struct GETATTR4resok {
        fattr4           obj_attributes;
};

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

15.9.4. DESCRIPTION

   The GETATTR operation will obtain attributes for the file system
   object specified by the current filehandle. The client sets a bit in
   the bitmap argument for each attribute value that it would like the

server to return. The server returns an attribute bitmap that
indicates the attribute values for which it was able to return
values, followed by the attribute values ordered lowest attribute
number first.

The server MUST return a value for each attribute that the client
requests if the attribute is supported by the server. If the server
does not support an attribute or cannot approximate a useful value
then it MUST NOT return the attribute value and MUST NOT set the
attribute bit in the result bitmap. The server MUST return an error
if it supports an attribute on the target but cannot obtain its
value. In that case no attribute values will be returned.

File systems which are absent should be treated as having support for
a very small set of attributes as described in GETATTR Within an
Absent File System (Section 8.3.1), even if previously, when the file
system was present, more attributes were supported.

All servers MUST support the REQUIRED attributes as specified in the
section File Attributes (Section 5), for all file systems, with the
exception of absent file systems.

On success, the current filehandle retains its value.

15.9.5. IMPLEMENTATION

Suppose there is a OPEN_DELEGATE_WRITE delegation held by another
client for file in question and size and/or change are among the set
of attributes being interrogated. The server has two choices.
First, the server can obtain the actual current value of these
attributes from the client holding the delegation by using the
CB_GETATTR callback. Second, the server, particularly when the
delegated client is unresponsive, can recall the delegation in
question. The GETATTR MUST NOT proceed until one of the following
occurs:

o The requested attribute values are returned in the response to
   CB_GETATTR.

o The OPEN_DELEGATE_WRITE delegation is returned.

o The OPEN_DELEGATE_WRITE delegation is revoked.

Unless one of the above happens very quickly, one or more
NFS4ERR_DELAY errors will be returned while a delegation is
outstanding.

15.10. Operation 10: GETFH - Get Current Filehandle

15.10.1. SYNOPSIS

   (cfh) -> filehandle

15.10.2. ARGUMENT

   /* CURRENT_FH: */
   void;

15.10.3. RESULT

   struct GETFH4resok {
         nfs_fh4   object;
   };

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

15.10.4. DESCRIPTION

   This operation returns the current filehandle value.

   On success, the current filehandle retains its value.

15.10.5. IMPLEMENTATION

   Operations that change the current filehandle like LOOKUP or CREATE
   do not automatically return the new filehandle as a result. For
   instance, if a client needs to lookup a directory entry and obtain
   its filehandle then the following request is needed.

     PUTFH (directory filehandle)
     LOOKUP (entry name)
     GETFH

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

15.11.1. SYNOPSIS

     (sfh), (cfh), newname -> (cfh), cinfo

15.11.2. ARGUMENT

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

15.11.3. RESULT

```
struct LINK4resok {
        change_info4    cinfo;
};

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

15.11.4. DESCRIPTION

   The LINK operation creates an additional newname for the file
   represented by the saved filehandle, as set by the SAVEFH operation,
   in the directory represented by the current filehandle. The existing
   file and the target directory must reside within the same file system
   on the server. On success, the current filehandle will continue to
   be the target directory. If an object exists in the target directory
   with the same name as newname, the server must return NFS4ERR_EXIST.

   For the target directory, the server returns change_info4 information
   in cinfo. With the atomic field of the change_info4 struct, the
   server will indicate if the before and after change attributes were
   obtained atomically with respect to the link creation.

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

15.11.5. IMPLEMENTATION

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

   The statement "file and the target directory must reside within the
   same file system on the server" means that the fsid fields in the
   attributes for the objects are the same. If they reside on different
   file systems, the error NFS4ERR_XDEV is returned. This error may be
   returned by some servers when there is an internal partitioning of a
   file system that the LINK operation would violate.

   On some servers, "." and ".." are illegal values for newname and the
   error NFS4ERR_BADNAME will be returned if they are specified.

   When the current filehandle designates a named attribute directory
   and the object to be linked (the saved filehandle) is not a named
   attribute for the same object, the error NFS4ERR_XDEV MUST be
   returned. When the saved filehandle designates a named attribute and
   the current filehandle is not the appropriate named attribute
   directory, the error NFS4ERR_XDEV MUST also be returned.

   When the current filehandle designates a named attribute directory
   and the object to be linked (the saved filehandle) is a named
   attribute within that directory, the server MAY return the error
   NFS4ERR_NOTSUPP.

   In the case that newname is already linked to the file represented by
   the saved filehandle, the server will return NFS4ERR_EXIST.

   Note that symbolic links are created with the CREATE operation.

15.12. Operation 12: LOCK - Create Lock

15.12.1. SYNOPSIS

      (cfh) locktype, reclaim, offset, length, locker -> stateid

15.12.2. ARGUMENT

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

```
   /*
    * For LOCK, transition from open_owner to new lock_owner
    */
   struct open_to_lock_owner4 {
           seqid4          open_seqid;
           stateid4        open_stateid;
           seqid4          lock_seqid;
           lock_owner4     lock_owner;
   };

   /*
    * For LOCK, existing lock_owner continues to request file locks
    */
   struct exist_lock_owner4 {
           stateid4        lock_stateid;
           seqid4          lock_seqid;
   };

   union locker4 switch (bool new_lock_owner) {
    case TRUE:
           open_to_lock_owner4     open_owner;
    case FALSE:
           exist_lock_owner4       lock_owner;
   };

   /*
    * LOCK/LOCKT/LOCKU: Record lock management
    */
   struct LOCK4args {
           /* CURRENT_FH: file */
           nfs_lock_type4  locktype;
           bool            reclaim;
           offset4         offset;
           length4         length;
           locker4         locker;
   };
```

15.12.3. RESULT

```
struct LOCK4denied {
        offset4         offset;
        length4         length;
        nfs_lock_type4  locktype;
        lock_owner4     owner;
};

struct LOCK4resok {
        stateid4        lock_stateid;
};

union LOCK4res switch (nfsstat4 status) {
 case NFS4_OK:
        LOCK4resok      resok4;
 case NFS4ERR_DENIED:
        LOCK4denied     denied;
 default:
        void;
};
```

15.12.4. DESCRIPTION

   The LOCK operation requests a byte-range lock for the byte range
   specified by the offset and length parameters. The lock type is also
   specified to be one of the nfs_lock_type4s. If this is a reclaim
   request, the reclaim parameter will be TRUE;

   Bytes in a file may be locked even if those bytes are not currently
   allocated to the file. To lock the file from a specific offset
   through the end-of-file (no matter how long the file actually is) use
   a length field with all bits set to 1 (one). If the length is zero,
   or if a length which is not all bits set to one is specified, and
   length when added to the offset exceeds the maximum 64-bit unsigned
   integer value, the error NFS4ERR_INVAL will result.

   Some servers may only support locking for byte offsets that fit
   within 32 bits. If the client specifies a range that includes a byte
   beyond the last byte offset of the 32-bit range, but does not include
   the last byte offset of the 32-bit and all of the byte offsets beyond
   it, up to the end of the valid 64-bit range, such a 32-bit server
   MUST return the error NFS4ERR_BAD_RANGE.

   In the case that the lock is denied, the owner, offset, and length of
   a conflicting lock are returned.

   On success, the current filehandle retains its value.

15.12.5. IMPLEMENTATION

   If the server is unable to determine the exact offset and length of
   the conflicting lock, the same offset and length that were provided
   in the arguments should be returned in the denied results. Section 9
   contains a full description of this and the other file locking
   operations.

   LOCK operations are subject to permission checks and to checks
   against the access type of the associated file. However, the
   specific right and modes required for various type of locks, reflect
   the semantics of the server-exported file system, and are not
   specified by the protocol. For example, Windows 2000 allows a write
   lock of a file open for READ, while a POSIX-compliant system does
   not.

   When the client makes a lock request that corresponds to a range that
   the lock-owner has locked already (with the same or different lock
   type), or to a sub-region of such a range, or to a region which
   includes multiple locks already granted to that lock-owner, in whole
   or in part, and the server does not support such locking operations
   (i.e., does not support POSIX locking semantics), the server will
   return the error NFS4ERR_LOCK_RANGE. In that case, the client may
   return an error, or it may emulate the required operations, using
   only LOCK for ranges that do not include any bytes already locked by
   that lock-owner and LOCKU of locks held by that lock-owner
   (specifying an exactly-matching range and type). Similarly, when the
   client makes a lock request that amounts to upgrading (changing from
   a read lock to a write lock) or downgrading (changing from write lock
   to a read lock) an existing record lock, and the server does not
   support such a lock, the server will return NFS4ERR_LOCK_NOTSUPP.
   Such operations may not perfectly reflect the required semantics in
   the face of conflicting lock requests from other clients.

   When a client holds an OPEN_DELEGATE_WRITE delegation, the client
   holding that delegation is assured that there are no opens by other
   clients. Thus, there can be no conflicting LOCK operations from such
   clients. Therefore, the client may be handling locking requests
   locally, without doing LOCK operations on the server. If it does
   that, it must be prepared to update the lock status on the server, by
   sending appropriate LOCK and LOCKU operations before returning the
   delegation.

   When one or more clients hold OPEN_DELEGATE_READ delegations, any
   LOCK operation where the server is implementing mandatory locking
   semantics MUST result in the recall of all such delegations. The
   LOCK operation may not be granted until all such delegations are
   returned or revoked. Except where this happens very quickly, one or

   more NFS4ERR_DELAY errors will be returned to requests made while the
   delegation remains outstanding.

   The locker argument specifies the lock-owner that is associated with
   the LOCK request. The locker4 structure is a switched union that
   indicates whether the client has already created byte-range locking
   state associated with the current open file and lock-owner. There
   are multiple cases to be considered, corresponding to possible
   combinations of whether locking state has been created for the
   current open file and lock-owner, and whether the boolean
   new_lock_owner is set. In all of the cases, there is a lock_seqid
   specified, whether the lock-owner is specified explicitly or
   implicitly. This seqid value is used for checking lock-owner
   sequencing/replay issues. When the given lock-owner is not known to
   the server, this establishes an initial sequence value for the new
   lock-owner.

   o In the case in which the state has been created and the boolean is
      false, the only part of the argument other than lock_seqid is just
      a stateid representing the set of locks associated with that open
      file and lock-owner.

   o In the case in which the state has been created and the boolean is
      true, the server rejects the request with the error
      NFS4ERR_BAD_SEQID. The only exception is where there is a
      retransmission of a previous request in which the boolean was
      true. In this case, the lock_seqid will match the original
      request and the response will reflect the final case, below.

   o In the case where no byte-range locking state has been established
      and the boolean is true, the argument contains an
      open_to_lock_owner structure which specifies the stateid of the
      open file and the lock-owner to be used for the lock. Note that
      although the open-owner is not given explicitly, the open_seqid
      associated with it is used to check for open-owner sequencing
      issues. This case provides a method to use the established state
      of the open_stateid to transition to the use of a lock stateid.

15.13. Operation 13: LOCKT - Test For Lock

15.13.1. SYNOPSIS

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

15.13.2. ARGUMENT

```
struct LOCKT4args {
        /* CURRENT_FH: file */
        nfs_lock_type4  locktype;
        offset4         offset;
        length4         length;
        lock_owner4     owner;
};
```

15.13.3. RESULT

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

15.13.4. DESCRIPTION

The LOCKT operation tests the lock as specified in the arguments. If
a conflicting lock exists, the owner, offset, length, and type of the
conflicting lock are returned; if no lock is held, nothing other than
NFS4_OK is returned. Lock types READ_LT and READW_LT are processed
in the same way in that a conflicting lock test is done without
regard to blocking or non-blocking. The same is true for WRITE_LT
and WRITEW_LT.

The ranges are specified as for LOCK. The NFS4ERR_INVAL and
NFS4ERR_BAD_RANGE errors are returned under the same circumstances as
for LOCK.

On success, the current filehandle retains its value.

15.13.5. IMPLEMENTATION

If the server is unable to determine the exact offset and length of
the conflicting lock, the same offset and length that were provided
in the arguments should be returned in the denied results. Section 9
contains further discussion of the file locking mechanisms.

LOCKT uses a lock_owner4 rather a stateid4, as is used in LOCK to
identify the owner. This is because the client does not have to open

the file to test for the existence of a lock, so a stateid may not be available.

The test for conflicting locks SHOULD exclude locks for the current lock-owner. Note that since such locks are not examined the possible existence of overlapping ranges may not affect the results of LOCKT. If the server does examine locks that match the lock-owner for the purpose of range checking, NFS4ERR_LOCK_RANGE may be returned. In the event that it returns NFS4_OK, clients may do a LOCK and receive NFS4ERR_LOCK_RANGE on the LOCK request because of the flexibility provided to the server.

When a client holds an OPEN_DELEGATE_WRITE delegation, it may choose (see Section 15.12.5)) to handle LOCK requests locally. In such a case, LOCKT requests will similarly be handled locally.

15.14. Operation 14: LOCKU - Unlock File

15.14.1. SYNOPSIS

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

15.14.2. ARGUMENT

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

15.14.3. RESULT

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

15.14.4. DESCRIPTION

   The LOCKU operation unlocks the byte-range lock specified by the
   parameters. The client may set the locktype field to any value that
   is legal for the nfs_lock_type4 enumerated type, and the server MUST
   accept any legal value for locktype. Any legal value for locktype
   has no effect on the success or failure of the LOCKU operation.

   The ranges are specified as for LOCK. The NFS4ERR_INVAL and
   NFS4ERR_BAD_RANGE errors are returned under the same circumstances as
   for LOCK.

   On success, the current filehandle retains its value.

15.14.5. IMPLEMENTATION

   If the area to be unlocked does not correspond exactly to a lock
   actually held by the lock-owner the server may return the error
   NFS4ERR_LOCK_RANGE. This includes the case in which the area is not
   locked, where the area is a sub-range of the area locked, where it
   overlaps the area locked without matching exactly or the area
   specified includes multiple locks held by the lock-owner. In all of
   these cases, allowed by POSIX locking [fcntl] semantics, a client
   receiving this error, should if it desires support for such
   operations, simulate the operation using LOCKU on ranges
   corresponding to locks it actually holds, possibly followed by LOCK
   requests for the sub-ranges not being unlocked.

   When a client holds an OPEN_DELEGATE_WRITE delegation, it may choose
   (see Section 15.12.5)) to handle LOCK requests locally. In such a
   case, LOCKU requests will similarly be handled locally.

15.15. Operation 15: LOOKUP - Lookup Filename

15.15.1. SYNOPSIS

     (cfh), component -> (cfh)

15.15.2. ARGUMENT

   struct LOOKUP4args {
           /* CURRENT_FH: directory */
           component4   objname;
   };

15.15.3. RESULT

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

15.15.4. DESCRIPTION

This operation LOOKUPs or finds a file system object using the
directory specified by the current filehandle. LOOKUP evaluates the
component and if the object exists the current filehandle is replaced
with the component's filehandle.

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

If the component is of zero length, NFS4ERR_INVAL will be returned.
The component is also subject to the normal UTF-8, character support,
and name checks. See Section 12.7 for further discussion.

15.15.5. IMPLEMENTATION

If the client wants to achieve the effect of a multi-component
lookup, it may construct a COMPOUND request such as (and obtain each
filehandle):

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

NFSv4 servers depart from the semantics of previous NFS versions in
allowing LOOKUP requests to cross mount points on the server. The
client can detect a mount point crossing by comparing the fsid
attribute of the directory with the fsid attribute of the directory
looked up. If the fsids are different then the new directory is a
server mount point. UNIX clients that detect a mount point crossing
will need to mount the server's file system. This needs to be done
to maintain the file object identity checking mechanisms common to
UNIX clients.

Servers that limit NFS access to "shares" or "exported" file systems
should provide a pseudo-file system into which the exported file
systems can be integrated, so that clients can browse the server's
name space. The clients' view of a pseudo file system will be
limited to paths that lead to exported file systems.

Note: previous versions of the protocol assigned special semantics to
the names "." and "..". NFSv4 assigns no special semantics to these
names. The LOOKUPP operator must be used to lookup a parent
directory.

Note that this operation does not follow symbolic links. The client
is responsible for all parsing of filenames including filenames that
are modified by symbolic links encountered during the lookup process.

If the current filehandle supplied is not a directory but a symbolic
link, the error NFS4ERR_SYMLINK is returned as the error. For all
other non-directory file types, the error NFS4ERR_NOTDIR is returned.

15.16. Operation 16: LOOKUPP - Lookup Parent Directory

15.16.1. SYNOPSIS

    (cfh) -> (cfh)

15.16.2. ARGUMENT

    /* CURRENT_FH: object */
    void;

15.16.3. RESULT

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

15.16.4. DESCRIPTION

The current filehandle is assumed to refer to a regular directory or
a named attribute directory. LOOKUPP assigns the filehandle for its
parent directory to be the current filehandle. If there is no parent
directory an NFS4ERR_NOENT error must be returned. Therefore,
NFS4ERR_NOENT will be returned by the server when the current
filehandle is at the root or top of the server's file tree.

15.16.5. IMPLEMENTATION

   As for LOOKUP, LOOKUPP will also cross mount points.

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

   If the current filehandle is a named attribute directory that is
   associated with a file system object via OPENATTR (i.e., not a sub-
   directory of a named attribute directory), LOOKUPP SHOULD return the
   filehandle of the associated file system object.

15.17. Operation 17: NVERIFY - Verify Difference in Attributes

15.17.1. SYNOPSIS

      (cfh), fattr -> -

15.17.2. ARGUMENT

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

15.17.3. RESULT

   struct NVERIFY4res {
           nfsstat4   status;
   };

15.17.4. DESCRIPTION

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

   On success, the current filehandle retains its value.

15.17.5. IMPLEMENTATION

   This operation is useful as a cache validation operator. If the
   object to which the attributes belong has changed then the following
   operations may obtain new data associated with that object. For

    instance, to check if a file has been changed and obtain new data if
    it has:

      PUTFH (public)
      LOOKUP "foobar"
      NVERIFY attrbits attrs
      READ 0 32767

    In the case that a RECOMMENDED attribute is specified in the NVERIFY
    operation and the server does not support that attribute for the file
    system object, the error NFS4ERR_ATTRNOTSUPP is returned to the
    client.

    When the attribute rdattr_error or any write-only attribute (e.g.,
    time_modify_set) is specified, the error NFS4ERR_INVAL is returned to
    the client.

15.18. Operation 18: OPEN - Open a Regular File

15.18.1. SYNOPSIS

    (cfh), seqid, share_access, share_deny, owner, openhow, claim ->
    (cfh), stateid, cinfo, rflags, attrset, delegation

15.18.2. ARGUMENT

    /*
     * Various definitions for OPEN
     */
    enum createmode4 {
            UNCHECKED4   = 0,
            GUARDED4   = 1,
            EXCLUSIVE4   = 2
    };

    union createhow4 switch (createmode4 mode) {
     case UNCHECKED4:
     case GUARDED4:
            fattr4   createattrs;
     case EXCLUSIVE4:
            verifier4   createverf;
    };

    enum opentype4 {
            OPEN4_NOCREATE = 0,
            OPEN4_CREATE   = 1
    };

```
   union openflag4 switch (opentype4 opentype) {
    case OPEN4_CREATE:
           createhow4      how;
    default:
           void;
   };

   /* Next definitions used for OPEN delegation */
   enum limit_by4 {
           NFS_LIMIT_SIZE          = 1,
           NFS_LIMIT_BLOCKS        = 2
           /* others as needed */
   };

   struct nfs_modified_limit4 {
           uint32_t        num_blocks;
           uint32_t        bytes_per_block;
   };

   union nfs_space_limit4 switch (limit_by4 limitby) {
    /* limit specified as file size */
    case NFS_LIMIT_SIZE:
           uint64_t                filesize;
    /* limit specified by number of blocks */
    case NFS_LIMIT_BLOCKS:
           nfs_modified_limit4    mod_blocks;
   } ;

   enum open_delegation_type4 {
           OPEN_DELEGATE_NONE      = 0,
           OPEN_DELEGATE_READ      = 1,
           OPEN_DELEGATE_WRITE     = 2
   };

   enum open_claim_type4 {
           CLAIM_NULL              = 0,
           CLAIM_PREVIOUS          = 1,
           CLAIM_DELEGATE_CUR      = 2,
           CLAIM_DELEGATE_PREV     = 3
   };

   struct open_claim_delegate_cur4 {
           stateid4        delegate_stateid;
           component4      file;
   };

   union open_claim4 switch (open_claim_type4 claim) {
    /*
```

```
              * No special rights to file.
              * Ordinary OPEN of the specified file.
              */
             case CLAIM_NULL:
                     /* CURRENT_FH: directory */
                     component4      file;
             /*
              * Right to the file established by an
              * open previous to server reboot. File
              * identified by filehandle obtained at
              * that time rather than by name.
              */
             case CLAIM_PREVIOUS:
                     /* CURRENT_FH: file being reclaimed */
                     open_delegation_type4   delegate_type;

             /*
              * Right to file based on a delegation
              * granted by the server. File is
              * specified by name.
              */
             case CLAIM_DELEGATE_CUR:
                     /* CURRENT_FH: directory */
                     open_claim_delegate_cur4        delegate_cur_info;

             /*
              * Right to file based on a delegation
              * granted to a previous boot instance
              * of the client.  File is specified by name.
              */
             case CLAIM_DELEGATE_PREV:
                     /* CURRENT_FH: directory */
                     component4      file_delegate_prev;
        };

        /*
         * OPEN: Open a file, potentially receiving an open delegation
         */
        struct OPEN4args {
                seqid4          seqid;
                uint32_t        share_access;
                uint32_t        share_deny;
                open_owner4     owner;
                openflag4       openhow;
                open_claim4     claim;
        };
```

15.18.3.   RESULT

```
struct open_read_delegation4 {
 stateid4 stateid;    /* Stateid for delegation*/
 bool     recall;     /* Pre-recalled flag for
                          delegations obtained
                          by reclaim (CLAIM_PREVIOUS) */

 nfsace4 permissions; /* Defines users who don't
                          need an ACCESS call to
                          open for read */
};

struct open_write_delegation4 {
 stateid4 stateid;        /* Stateid for delegation */
 bool     recall;         /* Pre-recalled flag for
                              delegations obtained
                              by reclaim
                              (CLAIM_PREVIOUS) */

 nfs_space_limit4
         space_limit; /* Defines condition that
                          the client must check to
                          determine whether the
                          file needs to be flushed
                          to the server on close.  */

 nfsace4   permissions; /* Defines users who don't
                            need an ACCESS call as
                            part of a delegated
                            open. */
};

union open_delegation4
switch (open_delegation_type4 delegation_type) {
        case OPEN_DELEGATE_NONE:
                void;
        case OPEN_DELEGATE_READ:
                open_read_delegation4 read;
        case OPEN_DELEGATE_WRITE:
                open_write_delegation4 write;
};

/*
 * Result flags
 */

/* Client must confirm open */
```

```
const OPEN4_RESULT_CONFIRM      = 0x00000002;
/* Type of file locking behavior at the server */
const OPEN4_RESULT_LOCKTYPE_POSIX = 0x00000004;

struct OPEN4resok {
 stateid4        stateid;       /* Stateid for open */
 change_info4    cinfo;         /* Directory Change Info */
 uint32_t        rflags;        /* Result flags */
 bitmap4         attrset;       /* attribute set for create*/
 open_delegation4 delegation; /* Info on any open
                                  delegation */
};

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

15.18.4. Warning to Client Implementers

OPEN resembles LOOKUP in that it generates a filehandle for the
client to use. Unlike LOOKUP though, OPEN creates server state on
the filehandle. In normal circumstances, the client can only release
this state with a CLOSE operation. CLOSE uses the current filehandle
to determine which file to close. Therefore, the client MUST follow
every OPEN operation with a GETFH operation in the same COMPOUND
procedure. This will supply the client with the filehandle such that
CLOSE can be used appropriately.

Simply waiting for the lease on the file to expire is insufficient
because the server may maintain the state indefinitely as long as
another client does not attempt to make a conflicting access to the
same file.

15.18.5. DESCRIPTION

The OPEN operation creates and/or opens a regular file in a directory
with the provided name. If the file does not exist at the server and
creation is desired, specification of the method of creation is
provided by the openhow parameter. The client has the choice of
three creation methods: UNCHECKED4, GUARDED4, or EXCLUSIVE4.

If the current filehandle is a named attribute directory, OPEN will
then create or open a named attribute file. Note that exclusive

create of a named attribute is not supported. If the createmode is
EXCLUSIVE4 and the current filehandle is a named attribute directory,
the server will return EINVAL.

UNCHECKED4 means that the file should be created if a file of that
name does not exist and encountering an existing regular file of that
name is not an error. For this type of create, createattrs specifies
the initial set of attributes for the file. The set of attributes
may include any writable attribute valid for regular files. When an
UNCHECKED4 create encounters an existing file, the attributes
specified by createattrs are not used, except that when an size of
zero is specified, the existing file is truncated. If GUARDED4 is
specified, the server checks for the presence of a duplicate object
by name before performing the create. If a duplicate exists, an
error of NFS4ERR_EXIST is returned as the status. If the object does
not exist, the request is performed as described for UNCHECKED4. For
each of these cases (UNCHECKED4 and GUARDED4) where the operation is
successful, the server will return to the client an attribute mask
signifying which attributes were successfully set for the object.

EXCLUSIVE4 specifies that the server is to follow exclusive creation
semantics, using the verifier to ensure exclusive creation of the
target. The server should check for the presence of a duplicate
object by name. If the object does not exist, the server creates the
object and stores the verifier with the object. If the object does
exist and the stored verifier matches the client provided verifier,
the server uses the existing object as the newly created object. If
the stored verifier does not match, then an error of NFS4ERR_EXIST is
returned. No attributes may be provided in this case, since the
server may use an attribute of the target object to store the
verifier. If the server uses an attribute to store the exclusive
create verifier, it will signify which attribute by setting the
appropriate bit in the attribute mask that is returned in the
results.

For the target directory, the server returns change_info4 information
in cinfo. With the atomic field of the change_info4 struct, the
server will indicate if the before and after change attributes were
obtained atomically with respect to the link creation.

Upon successful creation, the current filehandle is replaced by that
of the new object.

The OPEN operation provides for Windows share reservation capability
with the use of the share_access and share_deny fields of the OPEN
arguments. The client specifies at OPEN the required share_access
and share_deny modes. For clients that do not directly support
SHAREs (i.e., UNIX), the expected deny value is DENY_NONE. In the

case that there is a existing SHARE reservation that conflicts with
the OPEN request, the server returns the error NFS4ERR_SHARE_DENIED.
For a complete SHARE request, the client must provide values for the
owner and seqid fields for the OPEN argument. For additional
discussion of SHARE semantics see Section 9.9.

In the case that the client is recovering state from a server
failure, the claim field of the OPEN argument is used to signify that
the request is meant to reclaim state previously held.

The "claim" field of the OPEN argument is used to specify the file to
be opened and the state information which the client claims to
possess. There are four basic claim types which cover the various
situations for an OPEN. They are as follows:

CLAIM_NULL: For the client, this is a new OPEN request and there is
   no previous state associate with the file for the client.

CLAIM_PREVIOUS: The client is claiming basic OPEN state for a file
   that was held previous to a server reboot. Generally used when a
   server is returning persistent filehandles; the client may not
   have the file name to reclaim the OPEN.

CLAIM_DELEGATE_CUR: The client is claiming a delegation for OPEN as
   granted by the server. Generally this is done as part of
   recalling a delegation.

CLAIM_DELEGATE_PREV: The client is claiming a delegation granted to
   a previous client instance. This claim type is for use after a
   SETCLIENTID_CONFIRM and before the corresponding DELEGPURGE in two
   situations: after a client reboot and after a lease expiration
   that resulted in loss of all lock state. The server MAY support
   CLAIM_DELEGATE_PREV. If it does support CLAIM_DELEGATE_PREV,
   SETCLIENTID_CONFIRM MUST NOT remove the client's delegation state,
   and the server MUST support the DELEGPURGE operation.

The following errors apply to use of the CLAIM_DELEGATE_PREV claim
type:

o NFS4ERR_NOTSUPP is returned if the server does not support this
   claim type.

o NFS4ERR_INVAL is returned if the reclaim is done at an
   inappropriate time, e.g., after DELEGPURGE has been done.

o NFS4ERR_BAD_RECLAIM is returned if the other error conditions do
   not apply and the server has no record of the delegation whose
   reclaim is being attempted.

For OPEN requests whose claim type is other than CLAIM_PREVIOUS
(i.e., requests other than those devoted to reclaiming opens after a
server reboot) that reach the server during its grace or lease
expiration period, the server returns an error of NFS4ERR_GRACE.

For any OPEN request, the server may return an open delegation, which
allows further opens and closes to be handled locally on the client
as described in Section 10.4. Note that delegation is up to the
server to decide. The client should never assume that delegation
will or will not be granted in a particular instance. It should
always be prepared for either case. A partial exception is the
reclaim (CLAIM_PREVIOUS) case, in which a delegation type is claimed.
In this case, delegation will always be granted, although the server
may specify an immediate recall in the delegation structure.

The rflags returned by a successful OPEN allow the server to return
information governing how the open file is to be handled.

OPEN4_RESULT_CONFIRM indicates that the client MUST execute an
OPEN_CONFIRM operation before using the open file.
OPEN4_RESULT_LOCKTYPE_POSIX indicates the server's file locking
behavior supports the complete set of Posix locking techniques
[fcntl]. From this the client can choose to manage file locking
state in a way to handle a mis-match of file locking management.

If the component is of zero length, NFS4ERR_INVAL will be returned.
The component is also subject to the normal UTF-8, character support,
and name checks. See Section 12.7 for further discussion.

When an OPEN is done and the specified open-owner already has the
resulting filehandle open, the result is to "OR" together the new
share and deny status together with the existing status. In this
case, only a single CLOSE need be done, even though multiple OPENs
were completed. When such an OPEN is done, checking of share
reservations for the new OPEN proceeds normally, with no exception
for the existing OPEN held by the same owner. In this case, the
stateid returned as an "other" field that matches that of the
previous open while the "seqid" field is incremented to reflect the
change status due to the new open (Section 9.1.4).

If the underlying file system at the server is only accessible in a
read-only mode and the OPEN request has specified
OPEN4_SHARE_ACCESSS_WRITE or OPEN4_SHARE_ACCESS_BOTH, the server will
return NFS4ERR_ROFS to indicate a read-only file system.

As with the CREATE operation, the server MUST derive the owner, owner
ACE, group, or group ACE if any of the four attributes are required
and supported by the server's file system. For an OPEN with the

EXCLUSIVE4 createmode, the server has no choice, since such OPEN
calls do not include the createattrs field. Conversely, if
createattrs is specified, and includes owner or group (or
corresponding ACEs) that the principal in the RPC call's credentials
does not have authorization to create files for, then the server may
return NFS4ERR_PERM.

In the case of a OPEN which specifies a size of zero (e.g.,
truncation) and the file has named attributes, the named attributes
are left as is. They are not removed.

15.18.6. IMPLEMENTATION

The OPEN operation contains support for EXCLUSIVE4 create. The
mechanism is similar to the support in NFSv3 [RFC1813]. As in NFSv3,
this mechanism provides reliable exclusive creation. Exclusive
create is invoked when the how parameter is EXCLUSIVE4. In this
case, the client provides a verifier that can reasonably be expected
to be unique. A combination of a client identifier, perhaps the
client network address, and a unique number generated by the client,
perhaps the RPC transaction identifier, may be appropriate.

If the object does not exist, the server creates the object and
stores the verifier in stable storage. For file systems that do not
provide a mechanism for the storage of arbitrary file attributes, the
server may use one or more elements of the object meta-data to store
the verifier. The verifier must be stored in stable storage to
prevent erroneous failure on retransmission of the request. It is
assumed that an exclusive create is being performed because exclusive
semantics are critical to the application. Because of the expected
usage, exclusive CREATE does not rely solely on the normally volatile
duplicate request cache for storage of the verifier. The duplicate
request cache in volatile storage does not survive a crash and may
actually flush on a long network partition, opening failure windows.
In the UNIX local file system environment, the expected storage
location for the verifier on creation is the meta-data (time stamps)
of the object. For this reason, an exclusive object create may not
include initial attributes because the server would have nowhere to
store the verifier.

If the server cannot support these exclusive create semantics,
possibly because of the requirement to commit the verifier to stable
storage, it should fail the OPEN request with the error,
NFS4ERR_NOTSUPP.

During an exclusive CREATE request, if the object already exists, the
server reconstructs the object's verifier and compares it with the
verifier in the request. If they match, the server treats the

request as a success. The request is presumed to be a duplicate of
an earlier, successful request for which the reply was lost and that
the server duplicate request cache mechanism did not detect. If the
verifiers do not match, the request is rejected with the status,
NFS4ERR_EXIST.

Once the client has performed a successful exclusive create, it must
issue a SETATTR to set the correct object attributes. Until it does
so, it should not rely upon any of the object attributes, since the
server implementation may need to overload object meta-data to store
the verifier. The subsequent SETATTR must not occur in the same
COMPOUND request as the OPEN. This separation will guarantee that
the exclusive create mechanism will continue to function properly in
the face of retransmission of the request.

Use of the GUARDED4 attribute does not provide exactly-once
semantics. In particular, if a reply is lost and the server does not
detect the retransmission of the request, the operation can fail with
NFS4ERR_EXIST, even though the create was performed successfully.
The client would use this behavior in the case that the application
has not requested an exclusive create but has asked to have the file
truncated when the file is opened. In the case of the client timing
out and retransmitting the create request, the client can use
GUARDED4 to prevent against a sequence like: create, write, create
(retransmitted) from occurring.

For SHARE reservations (see Section 9.9), the client must specify a
value for share_access that is one of OPEN4_SHARE_ACCESS_READ,
OPEN4_SHARE_ACCESS_WRITE, or OPEN4_SHARE_ACCESS_BOTH. For
share_deny, the client must specify one of OPEN4_SHARE_DENY_NONE,
OPEN4_SHARE_DENY_READ, OPEN4_SHARE_DENY_WRITE, or
OPEN4_SHARE_DENY_BOTH. If the client fails to do this, the server
must return NFS4ERR_INVAL.

Based on the share_access value (OPEN4_SHARE_ACCESS_READ,
OPEN4_SHARE_ACCESS_WRITE, or OPEN4_SHARE_ACCESS_BOTH) the client
should check that the requester has the proper access rights to
perform the specified operation. This would generally be the results
of applying the ACL access rules to the file for the current
requester. However, just as with the ACCESS operation, the client
should not attempt to second-guess the server's decisions, as access
rights may change and may be subject to server administrative
controls outside the ACL framework. If the requester is not
authorized to READ or WRITE (depending on the share_access value),
the server must return NFS4ERR_ACCESS. Note that since the NFS
version 4 protocol does not impose any requirement that READs and
WRITEs issued for an open file have the same credentials as the OPEN

itself, the server still must do appropriate access checking on the READs and WRITEs themselves.

If the component provided to OPEN resolves to something other than a regular file (or a named attribute), an error will be returned to the client. If it is a directory, NFS4ERR_ISDIR is returned; otherwise, NFS4ERR_SYMLINK is returned. Note that NFS4ERR_SYMLINK is returned for both symlinks and for special files of other types; NFS4ERR_INVAL would be inappropriate, since the arguments provided by the client were correct, and the client cannot necessarily know at the time it sent the OPEN that the component would resolve to a non-regular file.

If the current filehandle is not a directory, the error NFS4ERR_NOTDIR will be returned.

If a COMPOUND contains an OPEN which establishes an OPEN_DELEGATE_WRITE delegation, then normally subsequent GETATTRs result in a CB_GETATTR being sent to the client holding the delegation. However, in the case in which the OPEN and GETATTR are part of the same COMPOUND, the server SHOULD understand that the operations are for the same client ID and avoid querying the client, which will not be able to respond. This sequence of OPEN, GETATTR SHOULD be understood as retrieving of the size and change attributes at the time of OPEN, Further, as explained in Section 15.2.5, the client should not construct a COMPOUND which mixes operations for different client IDs.

15.19. Operation 19: OPENATTR - Open Named Attribute Directory

15.19.1. SYNOPSIS

    (cfh) createdir -> (cfh)

15.19.2. ARGUMENT

```
struct OPENATTR4args {
        /* CURRENT_FH: object */
        bool    createdir;
};
```

15.19.3. RESULT

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

15.19.4. DESCRIPTION

   The OPENATTR operation is used to obtain the filehandle of the named
   attribute directory associated with the current filehandle. The
   result of the OPENATTR will be a filehandle to an object of type
   NF4ATTRDIR. From this filehandle, READDIR and LOOKUP operations can
   be used to obtain filehandles for the various named attributes
   associated with the original file system object. Filehandles
   returned within the named attribute directory will have a type of
   NF4NAMEDATTR.

   The createdir argument allows the client to signify if a named
   attribute directory should be created as a result of the OPENATTR
   operation. Some clients may use the OPENATTR operation with a value
   of FALSE for createdir to determine if any named attributes exist for
   the object. If none exist, then NFS4ERR_NOENT will be returned. If
   createdir has a value of TRUE and no named attribute directory
   exists, one is created. The creation of a named attribute directory
   assumes that the server has implemented named attribute support in
   this fashion and is not required to do so by this definition.

15.19.5. IMPLEMENTATION

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

15.20. Operation 20: OPEN_CONFIRM - Confirm Open

15.20.1. SYNOPSIS

      (cfh), seqid, stateid -> stateid

15.20.2. ARGUMENT

   struct OPEN_CONFIRM4args {
           /* CURRENT_FH: opened file */
           stateid4   open_stateid;
           seqid4   seqid;
   };

15.20.3. RESULT

```
struct OPEN_CONFIRM4resok {
        stateid4        open_stateid;
};

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

15.20.4. DESCRIPTION

   This operation is used to confirm the sequence id usage for the first
   time that a open-owner is used by a client. The stateid returned
   from the OPEN operation is used as the argument for this operation
   along with the next sequence id for the open-owner. The sequence id
   passed to the OPEN_CONFIRM must be 1 (one) greater than the seqid
   passed to the OPEN operation (Section 9.1.4). If the server receives
   an unexpected sequence id with respect to the original open, then the
   server assumes that the client will not confirm the original OPEN and
   all state associated with the original OPEN is released by the
   server.

   On success, the current filehandle retains its value.

15.20.5. IMPLEMENTATION

   A given client might generate many open_owner4 data structures for a
   given client ID. The client will periodically either dispose of its
   open_owner4s or stop using them for indefinite periods of time. The
   latter situation is why the NFSv4 protocol does not have an explicit
   operation to exit an open_owner4: such an operation is of no use in
   that situation. Instead, to avoid unbounded memory use, the server
   needs to implement a strategy for disposing of open_owner4s that have
   no current open state for any files and have not been used recently.
   The time period used to determine when to dispose of open_owner4s is
   an implementation choice. The time period should certainly be no
   less than the lease time plus any grace period the server wishes to
   implement beyond a lease time. The OPEN_CONFIRM operation allows the
   server to safely dispose of unused open_owner4 data structures.

   In the case that a client issues an OPEN operation and the server no
   longer has a record of the open_owner4, the server needs to ensure
   that this is a new OPEN and not a replay or retransmission.

Servers MUST NOT require confirmation on OPENs that grant delegations
or are doing reclaim operations. See Section 9.1.11 for details.
The server can easily avoid this by noting whether it has disposed of
one open_owner4 for the given client ID. If the server does not
support delegation, it might simply maintain a single bit that notes
whether any open_owner4 (for any client) has been disposed of.

The server must hold unconfirmed OPEN state until one of three events
occur. First, the client sends an OPEN_CONFIRM request with the
appropriate sequence id and stateid within the lease period. In this
case, the OPEN state on the server goes to confirmed, and the
open_owner4 on the server is fully established.

Second, the client sends another OPEN request with a sequence id that
is incorrect for the open_owner4 (out of sequence). In this case,
the server assumes the second OPEN request is valid and the first one
is a replay. The server cancels the OPEN state of the first OPEN
request, establishes an unconfirmed OPEN state for the second OPEN
request, and responds to the second OPEN request with an indication
that an OPEN_CONFIRM is needed. The process then repeats itself.
While there is a potential for a denial of service attack on the
client, it is mitigated if the client and server require the use of a
security flavor based on Kerberos V5 or some other flavor that uses
cryptography.

What if the server is in the unconfirmed OPEN state for a given
open_owner4, and it receives an operation on the open_owner4 that has
a stateid but the operation is not OPEN, or it is OPEN_CONFIRM but
with the wrong stateid? Then, even if the seqid is correct, the
server returns NFS4ERR_BAD_STATEID, because the server assumes the
operation is a replay: if the server has no established OPEN state,
then there is no way, for example, a LOCK operation could be valid.

Third, neither of the two aforementioned events occur for the
open_owner4 within the lease period. In this case, the OPEN state is
canceled and disposal of the open_owner4 can occur.

15.21. Operation 21: OPEN_DOWNGRADE - Reduce Open File Access

15.21.1. SYNOPSIS

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

15.21.2. ARGUMENT

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

15.21.3. RESULT

```
    struct OPEN_DOWNGRADE4resok {
            stateid4         open_stateid;
    };

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

15.21.4. DESCRIPTION

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

The share_access and share_deny bits specified in this operation replace the current ones for the specified open file. The share_access and share_deny bits specified must be exactly equal to the union of the share_access and share_deny bits specified for some subset of the OPENs in effect for current open-owner on the current file. If that constraint is not respected, the error NFS4ERR_INVAL should be returned. Since share_access and share_deny bits are subsets of those already granted, it is not possible for this request to be denied because of conflicting share reservations.

As the OPEN_DOWNGRADE may change a file to be not-open-for-write and a write byte-range lock might be held, the server may have to reject the OPEN_DOWNGRADE with a NFS4ERR_LOCKS_HELD.

On success, the current filehandle retains its value.

15.22. Operation 22: PUTFH - Set Current Filehandle

15.22.1. SYNOPSIS

    filehandle -> (cfh)

15.22.2. ARGUMENT

```
struct PUTFH4args {
        nfs_fh4        object;
};
```

15.22.3. RESULT

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

15.22.4. DESCRIPTION

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

   If the security mechanism used by the requester does not meet the
   requirements of the filehandle provided to this operation, the server
   MUST return NFS4ERR_WRONGSEC.

   See Section 15.2.4.1 for more details on the current filehandle.

15.22.5. IMPLEMENTATION

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

15.23. Operation 23: PUTPUBFH - Set Public Filehandle

15.23.1. SYNOPSIS

    - -> (cfh)

15.23.2. ARGUMENT

    void;

15.23.3. RESULT

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

15.23.4. DESCRIPTION

   Replaces the current filehandle with the filehandle that represents
   the public filehandle of the server's name space. This filehandle
   may be different from the "root" filehandle which may be associated
   with some other directory on the server.

   The public filehandle concept was introduced in [RFC2054], [RFC2055],
   [RFC2224]. The intent for NFSv4 is that the public filehandle
   (represented by the PUTPUBFH operation) be used as a method of
   providing compatibility with the WebNFS server of NFSv2 and NFSv3.

   The public filehandle and the root filehandle (represented by the
   PUTROOTFH operation) should be equivalent. If the public and root
   filehandles are not equivalent, then the public filehandle MUST be a
   descendant of the root filehandle.

15.23.5. IMPLEMENTATION

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

   With the NFSv2 and 3 public filehandle, the client is able to specify
   whether the path name provided in the LOOKUP should be evaluated as
   either an absolute path relative to the server's root or relative to
   the public filehandle. [RFC2224] contains further discussion of the
   functionality. With NFSv4, that type of specification is not
   directly available in the LOOKUP operation. The reason for this is
   because the component separators needed to specify absolute vs.
   relative are not allowed in NFSv4. Therefore, the client is
   responsible for constructing its request such that the use of either
   PUTROOTFH or PUTPUBFH are used to signify absolute or relative
   evaluation of an NFS URL respectively.

   Note that there are warnings mentioned in [RFC2224] with respect to
   the use of absolute evaluation and the restrictions the server may
   place on that evaluation with respect to how much of its namespace
   has been made available. These same warnings apply to NFSv4. It is
   likely, therefore that because of server implementation details, an

NFSv3 absolute public filehandle lookup may behave differently than an NFSv4 absolute resolution.

There is a form of security negotiation as described in [RFC2755] that uses the public filehandle as a method of employing Simple and Protected GSSAPI Negotiation Mechanism (SNEGO) [RFC4178]. This method is not available with NFSv4 as filehandles are not overloaded with special meaning and therefore do not provide the same framework as NFSv2 and NFSv3. Clients should therefore use the security negotiation mechanisms described in this RFC.

15.24. Operation 24: PUTROOTFH - Set Root Filehandle

15.24.1. SYNOPSIS

      - -> (cfh)

15.24.2. ARGUMENT

     void;

15.24.3. RESULT

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

15.24.4. DESCRIPTION

   Replaces the current filehandle with the filehandle that represents the root of the server's name space. From this filehandle a LOOKUP operation can locate any other filehandle on the server. This filehandle may be different from the "public" filehandle which may be associated with some other directory on the server.

   See Section 15.2.4.1 for more details on the current filehandle.

15.24.5. IMPLEMENTATION

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

15.25. Operation 25: READ - Read from File

15.25.1. SYNOPSIS

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

15.25.2. ARGUMENT

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

15.25.3. RESULT

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

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

15.25.4. DESCRIPTION

   The READ operation reads data from the regular file identified by the
   current filehandle.

   The client provides an offset of where the READ is to start and a
   count of how many bytes are to be read. An offset of 0 (zero) means
   to read data starting at the beginning of the file. If offset is
   greater than or equal to the size of the file, the status, NFS4_OK,
   is returned with a data length set to 0 (zero) and eof is set to
   TRUE. The READ is subject to access permissions checking.

   If the client specifies a count value of 0 (zero), the READ succeeds
   and returns 0 (zero) bytes of data again subject to access
   permissions checking. The server may choose to return fewer bytes
   than specified by the client. The client needs to check for this
   condition and handle the condition appropriately.

The stateid value for a READ request represents a value returned from
a previous byte-range lock or share reservation request or the
stateid associated with a delegation. The stateid is used by the
server to verify that the associated share reservation and any byte-
range locks are still valid and to update lease timeouts for the
client.

If the read ended at the end-of-file (formally, in a correctly formed
READ request, if offset + count is equal to the size of the file), or
the read request extends beyond the size of the file (if offset +
count is greater than the size of the file), eof is returned as TRUE;
otherwise it is FALSE. A successful READ of an empty file will
always return eof as TRUE.

If the current filehandle is not a regular file, an error will be
returned to the client. In the case the current filehandle
represents a directory, NFS4ERR_ISDIR is returned; otherwise,
NFS4ERR_INVAL is returned.

For a READ using the special anonymous stateid, the server MAY allow
the READ to be serviced subject to mandatory file locks or the
current share deny modes for the file. For a READ using the special
READ bypass stateid, the server MAY allow READ operations to bypass
locking checks at the server.

On success, the current filehandle retains its value.

15.25.5. IMPLEMENTATION

If the server returns a "short read" (i.e., fewer data than requested
and eof is set to FALSE), the client should send another READ to get
the remaining data. A server may return less data than requested
under several circumstances. The file may have been truncated by
another client or perhaps on the server itself, changing the file
size from what the requesting client believes to be the case. This
would reduce the actual amount of data available to the client. It
is possible that the server reduces the transfer size and so returns
a short read result. Server resource exhaustion may also result in a
short read.

If mandatory byte-range locking is in effect for the file, and if the
byte-range corresponding to the data to be read from the file is
WRITE_LT locked by an owner not associated with the stateid, the
server will return the NFS4ERR_LOCKED error. The client should try
to get the appropriate READ_LT via the LOCK operation before
reattempting the READ. When the READ completes, the client should
release the byte-range lock via LOCKU.

If another client has an OPEN_DELEGATE_WRITE delegation for the file
being read, the delegation must be recalled, and the operation cannot
proceed until that delegation is returned or revoked. Except where
this happens very quickly, one or more NFS4ERR_DELAY errors will be
returned to requests made while the delegation remains outstanding.
Normally, delegations will not be recalled as a result of a READ
operation since the recall will occur as a result of an earlier OPEN.
However, since it is possible for a READ to be done with a special
stateid, the server needs to check for this case even though the
client should have done an OPEN previously.

15.26. Operation 26: READDIR - Read Directory

15.26.1. SYNOPSIS

   (cfh), cookie, cookieverf, dircount, maxcount, attr_request ->
   cookieverf { cookie, name, attrs }

15.26.2. ARGUMENT

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

15.26.3. RESULT

```
struct entry4 {
        nfs_cookie4     cookie;
        component4      name;
        fattr4          attrs;
        entry4          *nextentry;
};

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

struct READDIR4resok {
        verifier4       cookieverf;
        dirlist4        reply;
};


union READDIR4res switch (nfsstat4 status) {
 case NFS4_OK:
        READDIR4resok  resok4;
 default:
        void;
};
```

15.26.4. DESCRIPTION

   The READDIR operation retrieves a variable number of entries from a
   file system directory and returns client requested attributes for
   each entry along with information to allow the client to request
   additional directory entries in a subsequent READDIR.

   The arguments contain a cookie value that represents where the
   READDIR should start within the directory. A value of 0 (zero) for
   the cookie is used to start reading at the beginning of the
   directory. For subsequent READDIR requests, the client specifies a
   cookie value that is provided by the server on a previous READDIR
   request.

   The cookieverf value should be set to 0 (zero) when the cookie value
   is 0 (zero) (first directory read). On subsequent requests, it
   should be a cookieverf as returned by the server. The cookieverf
   must match that returned by the READDIR in which the cookie was
   acquired. If the server determines that the cookieverf is no longer
   valid for the directory, the error NFS4ERR_NOT_SAME must be returned.

The dircount portion of the argument is a hint of the maximum number
of bytes of directory information that should be returned. This
value represents the length of the names of the directory entries and
the cookie value for these entries. This length represents the XDR
encoding of the data (names and cookies) and not the length in the
native format of the server.

The maxcount value of the argument is the maximum number of bytes for
the result. This maximum size represents all of the data being
returned within the READDIR4resok structure and includes the XDR
overhead. The server may return less data. If the server is unable
to return a single directory entry within the maxcount limit, the
error NFS4ERR_TOOSMALL will be returned to the client.

Finally, attr_request represents the list of attributes to be
returned for each directory entry supplied by the server.

On successful return, the server's response will provide a list of
directory entries. Each of these entries contains the name of the
directory entry, a cookie value for that entry, and the associated
attributes as requested. The "eof" flag has a value of TRUE if there
are no more entries in the directory.

The cookie value is only meaningful to the server and is used as a
"bookmark" for the directory entry. As mentioned, this cookie is
used by the client for subsequent READDIR operations so that it may
continue reading a directory. The cookie is similar in concept to a
READ offset but should not be interpreted as such by the client. The
server SHOULD try to accept cookie values issued with READDIR
responses even if the directory has been modified between the READDIR
calls but MAY return NFS4ERR_NOT_VALID if this is not possible as
might be the case if the server has rebooted in the interim.

In some cases, the server may encounter an error while obtaining the
attributes for a directory entry. Instead of returning an error for
the entire READDIR operation, the server can instead return the
attribute 'fattr4_rdattr_error'. With this, the server is able to
communicate the failure to the client and not fail the entire
operation in the instance of what might be a transient failure.
Obviously, the client must request the fattr4_rdattr_error attribute
for this method to work properly. If the client does not request the
attribute, the server has no choice but to return failure for the
entire READDIR operation.

For some file system environments, the directory entries "." and ".."
have special meaning and in other environments, they may not. If the
server supports these special entries within a directory, they should
not be returned to the client as part of the READDIR response. To

enable some client environments, the cookie values of 0, 1, and 2 are
to be considered reserved. Note that the UNIX client will use these
values when combining the server's response and local representations
to enable a fully formed UNIX directory presentation to the
application.

For READDIR arguments, cookie values of 1 and 2 SHOULD NOT be used
and for READDIR results cookie values of 0, 1, and 2 MUST NOT be
returned.

On success, the current filehandle retains its value.

15.26.5. IMPLEMENTATION

The server's file system directory representations can differ
greatly. A client's programming interfaces may also be bound to the
local operating environment in a way that does not translate well
into the NFS protocol. Therefore the use of the dircount and
maxcount fields are provided to allow the client the ability to
provide guidelines to the server. If the client is aggressive about
attribute collection during a READDIR, the server has an idea of how
to limit the encoded response. The dircount field provides a hint on
the number of entries based solely on the names of the directory
entries. Since it is a hint, it may be possible that a dircount
value is zero. In this case, the server is free to ignore the
dircount value and return directory information based on the
specified maxcount value.

As there is no way for the client to indicate that a cookie value
once received, will not be subsequently used, server implementations
should avoid schemes that allocate memory corresponding to a returned
cookie. Such allocation can be avoided if the server bases cookie
values on a value such as the offset within the directory where the
scan is to be resumed.

Cookies generated by such techniques should be designed to remain
valid despite modification of the associated directory. If a server
were to invalidate a cookie because of a directory modification,
READDIR's of large directories might never finish.

If a directory is deleted after the client has carried out one or
more READDIR operations on the directory, the cookies returned will
become invalid but the server does not need to be concerned as the
directory file handle used previously would have become stale and
would be reported as such on subsequent READDIR operations. The
server would not need to check the cookie verifier in this case.

However, certain re-organization operations on a directory (including
directory compaction) may invalidate READDDIR cookies previously
given out. When such a situation occurs, the server should modify
the cookie verifier so as to disallow use of cookies which would
otherwise no longer be valid.

The cookieverf may be used by the server to help manage cookie values
that may become stale. It should be a rare occurrence that a server
is unable to continue properly reading a directory with the provided
cookie/cookieverf pair. The server should make every effort to avoid
this condition since the application at the client may not be able to
properly handle this type of failure.

The use of the cookieverf will also protect the client from using
READDIR cookie values that may be stale. For example, if the file
system has been migrated, the server may or may not be able to use
the same cookie values to service READDIR as the previous server
used. With the client providing the cookieverf, the server is able
to provide the appropriate response to the client. This prevents the
case where the server may accept a cookie value but the underlying
directory has changed and the response is invalid from the client's
context of its previous READDIR.

Since some servers will not be returning "." and ".." entries as has
been done with previous versions of the NFS protocol, the client that
requires these entries be present in READDIR responses must fabricate
them.

15.27. Operation 27: READLINK - Read Symbolic Link

15.27.1. SYNOPSIS

   (cfh) -> linktext

15.27.2. ARGUMENT

   /* CURRENT_FH: symlink */
   void;

15.27.3. RESULT

```
struct READLINK4resok {
        linktext4       link;
};

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

15.27.4. DESCRIPTION

   READLINK reads the data associated with a symbolic link. The data is
   a UTF-8 string that is opaque to the server. That is, whether
   created by an NFS client or created locally on the server, the data
   in a symbolic link is not interpreted when created, but is simply
   stored.

   On success, the current filehandle retains its value.

15.27.5. IMPLEMENTATION

   A symbolic link is nominally a pointer to another file. The data is
   not necessarily interpreted by the server, just stored in the file.
   It is possible for a client implementation to store a path name that
   is not meaningful to the server operating system in a symbolic link.
   A READLINK operation returns the data to the client for
   interpretation. If different implementations want to share access to
   symbolic links, then they must agree on the interpretation of the
   data in the symbolic link.

   The READLINK operation is only allowed on objects of type NF4LNK.
   The server should return the error, NFS4ERR_INVAL, if the object is
   not of type, NF4LNK.

15.28. Operation 28: REMOVE - Remove Filesystem Object

15.28.1. SYNOPSIS

     (cfh), filename -> change_info

15.28.2. ARGUMENT

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

15.28.3. RESULT

```
struct REMOVE4resok {
        change_info4    cinfo;
};

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

15.28.4. DESCRIPTION

   The REMOVE operation removes (deletes) a directory entry named by
   filename from the directory corresponding to the current filehandle.
   If the entry in the directory was the last reference to the
   corresponding file system object, the object may be destroyed.

   For the directory where the filename was removed, the server returns
   change_info4 information in cinfo. With the atomic field of the
   change_info4 struct, the server will indicate if the before and after
   change attributes were obtained atomically with respect to the
   removal.

   If the target is of zero length, NFS4ERR_INVAL will be returned. The
   target is also subject to the normal UTF-8, character support, and
   name checks. See Section 12.7 for further discussion.

   On success, the current filehandle retains its value.

15.28.5. IMPLEMENTATION

   NFSv3 required a different operator RMDIR for directory removal and
   REMOVE for non-directory removal. This allowed clients to skip
   checking the file type when being passed a non-directory delete
   system call (e.g., unlink() [unlink] in POSIX) to remove a directory,
   as well as the converse (e.g., a rmdir() on a non-directory) because
   they knew the server would check the file type. NFSv4 REMOVE can be
   used to delete any directory entry independent of its file type. The

implementer of an NFSv4 client's entry points from the unlink() and
rmdir() system calls should first check the file type against the
types the system call is allowed to remove before issuing a REMOVE.
Alternatively, the implementer can produce a COMPOUND call that
includes a LOOKUP/VERIFY sequence to verify the file type before a
REMOVE operation in the same COMPOUND call.

The concept of last reference is server specific. However, if the
numlinks field in the previous attributes of the object had the value
1, the client should not rely on referring to the object via a
filehandle. Likewise, the client should not rely on the resources
(disk space, directory entry, and so on) formerly associated with the
object becoming immediately available. Thus, if a client needs to be
able to continue to access a file after using REMOVE to remove it,
the client should take steps to make sure that the file will still be
accessible. The usual mechanism used is to RENAME the file from its
old name to a new hidden name.

If the server finds that the file is still open when the REMOVE
arrives:

o The server SHOULD NOT delete the file's directory entry if the
   file was opened with OPEN4_SHARE_DENY_WRITE or
   OPEN4_SHARE_DENY_BOTH.

o If the file was not opened with OPEN4_SHARE_DENY_WRITE or
   OPEN4_SHARE_DENY_BOTH, the server SHOULD delete the file's
   directory entry. However, until last CLOSE of the file, the
   server MAY continue to allow access to the file via its
   filehandle.

15.29. Operation 29: RENAME - Rename Directory Entry

15.29.1. SYNOPSIS

   (sfh), oldname, (cfh), newname -> source_cinfo, target_cinfo

15.29.2. ARGUMENT

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

15.29.3. RESULT

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

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

15.29.4. DESCRIPTION

   The RENAME operation renames the object identified by oldname in the
   source directory corresponding to the saved filehandle, as set by the
   SAVEFH operation, to newname in the target directory corresponding to
   the current filehandle. The operation is required to be atomic to
   the client. Source and target directories must reside on the same
   file system on the server. On success, the current filehandle will
   continue to be the target directory.

   If the target directory already contains an entry with the name,
   newname, the source object must be compatible with the target: either
   both are non-directories or both are directories and the target must
   be empty. If compatible, the existing target is removed before the
   rename occurs (See Section 15.28 for client and server actions
   whenever a target is removed). If they are not compatible or if the
   target is a directory but not empty, the server will return the
   error, NFS4ERR_EXIST.

   If oldname and newname both refer to the same file (they might be
   hard links of each other), then RENAME should perform no action and
   return success.

   For both directories involved in the RENAME, the server returns
   change_info4 information. With the atomic field of the change_info4
   struct, the server will indicate if the before and after change
   attributes were obtained atomically with respect to the rename.

   If the oldname refers to a named attribute and the saved and current
   filehandles refer to the named attribute directories of different
   file system objects, the server will return NFS4ERR_XDEV just as if
   the saved and current filehandles represented directories on
   different file systems.

If the oldname or newname is of zero length, NFS4ERR_INVAL will be
returned. The oldname and newname are also subject to the normal
UTF-8, character support, and name checks. See Section 12.7 for
further discussion.

15.29.5. IMPLEMENTATION

The RENAME operation must be atomic to the client. The statement
"source and target directories must reside on the same file system on
the server" means that the fsid fields in the attributes for the
directories are the same. If they reside on different file systems,
the error, NFS4ERR_XDEV, is returned.

Based on the value of the fh_expire_type attribute for the object,
the filehandle may or may not expire on a RENAME. However, server
implementers are strongly encouraged to attempt to keep filehandles
from expiring in this fashion.

On some servers, the file names "." and ".." are illegal as either
oldname or newname, and will result in the error NFS4ERR_BADNAME. In
addition, on many servers the case of oldname or newname being an
alias for the source directory will be checked for. Such servers
will return the error NFS4ERR_INVAL in these cases.

If either of the source or target filehandles are not directories,
the server will return NFS4ERR_NOTDIR.

15.30. Operation 30: RENEW - Renew a Lease

15.30.1. SYNOPSIS

   clientid -> ()

15.30.2. ARGUMENT

   struct RENEW4args {
        clientid4   clientid;
   };

15.30.3. RESULT

   struct RENEW4res {
        nfsstat4   status;
   };

15.30.4. DESCRIPTION

   The RENEW operation is used by the client to renew leases which it
   currently holds at a server. In processing the RENEW request, the
   server renews all leases associated with the client. The associated
   leases are determined by the clientid provided via the SETCLIENTID
   operation.

15.30.5. IMPLEMENTATION

   When the client holds delegations, it needs to use RENEW to detect
   when the server has determined that the callback path is down. When
   the server has made such a determination, only the RENEW operation
   will renew the lease on delegations. If the server determines the
   callback path is down, it returns NFS4ERR_CB_PATH_DOWN. Even though
   it returns NFS4ERR_CB_PATH_DOWN, the server MUST renew the lease on
   the byte-range locks and share reservations that the client has
   established on the server. If for some reason the lock and share
   reservation lease cannot be renewed, then the server MUST return an
   error other than NFS4ERR_CB_PATH_DOWN, even if the callback path is
   also down. In the event that the server has conditions such that it
   could return either NFS4ERR_CB_PATH_DOWN or NFS4ERR_LEASE_MOVED,
   NFS4ERR_LEASE_MOVED MUST be handled first.

   The client that issues RENEW MUST choose the principal, RPC security
   flavor, and if applicable, GSS-API mechanism and service via one of
   the following algorithms:

   o The client uses the same principal, RPC security flavor -- and if
      the flavor was RPCSEC_GSS -- the same mechanism and service that
      was used when the client ID was established via
      SETCLIENTID_CONFIRM.

   o The client uses any principal, RPC security flavor mechanism and
      service combination that currently has an OPEN file on the server.
      I.e., the same principal had a successful OPEN operation, the file
      is still open by that principal, and the flavor, mechanism, and
      service of RENEW match that of the previous OPEN.

   The server MUST reject a RENEW that does not use one the
   aforementioned algorithms, with the error NFS4ERR_ACCESS.

15.31. Operation 31: RESTOREFH - Restore Saved Filehandle

15.31.1. SYNOPSIS

     (sfh) -> (cfh)

15.31.2. ARGUMENT

     /* SAVED_FH: */
     void;

15.31.3. RESULT

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

15.31.4. DESCRIPTION

   Set the current filehandle to the value in the saved filehandle. If
   there is no saved filehandle then return the error NFS4ERR_RESTOREFH.

15.31.5. IMPLEMENTATION

   Operations like OPEN and LOOKUP use the current filehandle to
   represent a directory and replace it with a new filehandle. Assuming
   the previous filehandle was saved with a SAVEFH operator, the
   previous filehandle can be restored as the current filehandle. This
   is commonly used to obtain post-operation attributes for the
   directory, e.g.,

     PUTFH (directory filehandle)
     SAVEFH
     GETATTR attrbits   (pre-op dir attrs)
     CREATE optbits "foo" attrs
     GETATTR attrbits   (file attributes)
     RESTOREFH
     GETATTR attrbits   (post-op dir attrs)

15.32. Operation 32: SAVEFH - Save Current Filehandle

15.32.1. SYNOPSIS

     (cfh) -> (sfh)

15.32.2. ARGUMENT

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

15.32.3. RESULT

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

15.32.4. DESCRIPTION

Save the current filehandle. If a previous filehandle was saved then
it is no longer accessible. The saved filehandle can be restored as
the current filehandle with the RESTOREFH operator.

On success, the current filehandle retains its value.

15.32.5. IMPLEMENTATION

15.33. Operation 33: SECINFO - Obtain Available Security

15.33.1. SYNOPSIS

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

15.33.2. ARGUMENT

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

15.33.3. RESULT

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

   struct rpcsec_gss_info {
           sec_oid4        oid;
           qop4            qop;
           rpc_gss_svc_t   service;
   };

   /* RPCSEC_GSS has a value of '6' - See RFC 2203 */
   union secinfo4 switch (uint32_t flavor) {
    case RPCSEC_GSS:
            rpcsec_gss_info         flavor_info;
    default:
            void;
   };

   typedef secinfo4 SECINFO4resok<>;

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

15.33.4. DESCRIPTION

   The SECINFO operation is used by the client to obtain a list of valid
   RPC authentication flavors for a specific directory filehandle, file
   name pair. SECINFO should apply the same access methodology used for
   LOOKUP when evaluating the name. Therefore, if the requester does
   not have the appropriate access to LOOKUP the name then SECINFO must
   behave the same way and return NFS4ERR_ACCESS.

   The result will contain an array which represents the security
   mechanisms available, with an order corresponding to server's
   preferences, the most preferred being first in the array. The client
   is free to pick whatever security mechanism it both desires and
   supports, or to pick in the server's preference order the first one
   it supports. The array entries are represented by the secinfo4

structure. The field 'flavor' will contain a value of AUTH_NONE,
AUTH_SYS (as defined in [RFC5531]), or RPCSEC_GSS (as defined in
[RFC2203]).

For the flavors AUTH_NONE and AUTH_SYS, no additional security
information is returned. For a return value of RPCSEC_GSS, a
security triple is returned that contains the mechanism object id (as
defined in [RFC2743]), the quality of protection (as defined in
[RFC2743]) and the service type (as defined in [RFC2203]). It is
possible for SECINFO to return multiple entries with flavor equal to
RPCSEC_GSS with different security triple values.

On success, the current filehandle retains its value.

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

15.33.5. IMPLEMENTATION

The SECINFO operation is expected to be used by the NFS client when
the error value of NFS4ERR_WRONGSEC is returned from another NFS
operation. This signifies to the client that the server's security
policy is different from what the client is currently using. At this
point, the client is expected to obtain a list of possible security
flavors and choose what best suits its policies.

As mentioned, the server's security policies will determine when a
client request receives NFS4ERR_WRONGSEC. The operations which may
receive this error are: LINK, LOOKUP, LOOKUPP, OPEN, PUTFH, PUTPUBFH,
PUTROOTFH, RENAME, RESTOREFH, and indirectly READDIR. LINK and
RENAME will only receive this error if the security used for the
operation is inappropriate for saved filehandle. With the exception
of READDIR, these operations represent the point at which the client
can instantiate a filehandle into the "current filehandle" at the
server. The filehandle is either provided by the client (PUTFH,
PUTPUBFH, PUTROOTFH) or generated as a result of a name to filehandle
translation (LOOKUP and OPEN). RESTOREFH is different because the
filehandle is a result of a previous SAVEFH. Even though the
filehandle, for RESTOREFH, might have previously passed the server's
inspection for a security match, the server will check it again on
RESTOREFH to ensure that the security policy has not changed.

If the client wants to resolve an error return of NFS4ERR_WRONGSEC,
the following will occur:

o For LOOKUP and OPEN, the client will use SECINFO with the same
   current filehandle and name as provided in the original LOOKUP or
   OPEN to enumerate the available security triples.

o For LINK, PUTFH, RENAME, and RESTOREFH, the client will use
   SECINFO and provide the parent directory filehandle and object
   name which corresponds to the filehandle originally provided by
   the PUTFH RESTOREFH, or for LINK and RENAME, the SAVEFH.

o For LOOKUPP, PUTROOTFH and PUTPUBFH, the client will be unable to
   use the SECINFO operation since SECINFO requires a current
   filehandle and none exist for these two operations. Therefore,
   the client must iterate through the security triples available at
   the client and reattempt the PUTROOTFH or PUTPUBFH operation. In
   the unfortunate event none of the MANDATORY security triples are
   supported by the client and server, the client SHOULD try using
   others that support integrity. Failing that, the client can try
   using AUTH_NONE, but because such forms lack integrity checks,
   this puts the client at risk. Nonetheless, the server SHOULD
   allow the client to use whatever security form the client requests
   and the server supports, since the risks of doing so are on the
   client.

The READDIR operation will not directly return the NFS4ERR_WRONGSEC
error. However, if the READDIR request included a request for
attributes, it is possible that the READDIR request's security triple
does not match that of a directory entry. If this is the case and
the client has requested the rdattr_error attribute, the server will
return the NFS4ERR_WRONGSEC error in rdattr_error for the entry.

Note that a server MAY use the AUTH_NONE flavor to signify that the
client is allowed to attempt to use authentication flavors that are
not explicitly listed in the SECINFO results. Instead of using a
listed flavor, the client might then, for instance opt to use an
otherwise unlisted RPCSEC_GSS mechanism instead of AUTH_NONE. It may
wish to do so in order to meet an application requirement for data
integrity or privacy. In choosing to use an unlisted flavor, the
client SHOULD always be prepared to handle a failure by falling back
to using AUTH_NONE or another listed flavor. It cannot assume that
identity mapping is supported, and should be prepared for the fact
that its identity is squashed.

See Section 17 for a discussion on the recommendations for security
flavor used by SECINFO.

15.34. Operation 34: SETATTR - Set Attributes

15.34.1. SYNOPSIS

   (cfh), stateid, attrmask, attr_vals -> attrsset

15.34.2. ARGUMENT

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

15.34.3. RESULT

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

15.34.4. DESCRIPTION

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

   The stateid argument for SETATTR is used to provide byte-range
   locking context that is necessary for SETATTR requests that set the
   size attribute. Since setting the size attribute modifies the file's
   data, it has the same locking requirements as a corresponding WRITE.
   Any SETATTR that sets the size attribute is incompatible with a share
   reservation that specifies OPEN4_SHARE_DENY_WRITE. The area between
   the old end-of-file and the new end-of-file is considered to be
   modified just as would have been the case had the area in question
   been specified as the target of WRITE, for the purpose of checking
   conflicts with byte-range locks, for those cases in which a server is
   implementing mandatory byte-range locking behavior. A valid stateid
   SHOULD always be specified. When the file size attribute is not set,
   the special anonymous stateid MAY be passed.

   On either success or failure of the operation, the server will return
   the attrsset bitmask to represent what (if any) attributes were
   successfully set. The attrsset in the response is a subset of the
   bitmap4 that is part of the obj_attributes in the argument.

   On success, the current filehandle retains its value.

15.34.5. IMPLEMENTATION

   If the request specifies the owner attribute to be set, the server
   SHOULD allow the operation to succeed if the current owner of the
   object matches the value specified in the request. Some servers may
   be implemented in a way as to prohibit the setting of the owner
   attribute unless the requester has privilege to do so. If the server
   is lenient in this one case of matching owner values, the client
   implementation may be simplified in cases of creation of an object
   (e.g., an exclusive create via OPEN) followed by a SETATTR.

   The file size attribute is used to request changes to the size of a
   file. A value of zero causes the file to be truncated, a value less
   than the current size of the file causes data from new size to the
   end of the file to be discarded, and a size greater than the current
   size of the file causes logically zeroed data bytes to be added to
   the end of the file. Servers are free to implement this using holes
   or actual zero data bytes. Clients should not make any assumptions
   regarding a server's implementation of this feature, beyond that the
   bytes returned will be zeroed. Servers MUST support extending the
   file size via SETATTR.

   SETATTR is not guaranteed atomic. A failed SETATTR may partially
   change a file's attributes, hence the reason why the reply always
   includes the status and the list of attributes that were set.

   If the object whose attributes are being changed has a file
   delegation that is held by a client other than the one doing the
   SETATTR, the delegation(s) must be recalled, and the operation cannot
   proceed to actually change an attribute until each such delegation is
   returned or revoked. In all cases in which delegations are recalled,
   the server is likely to return one or more NFS4ERR_DELAY errors while
   the delegation(s) remains outstanding, although it might not do that
   if the delegations are returned quickly.

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

   The attributes time_access_set and time_modify_set are write-only
   attributes constructed as a switched union so the client can direct
   the server in setting the time values. If the switched union
   specifies SET_TO_CLIENT_TIME4, the client has provided an nfstime4 to
   be used for the operation. If the switch union does not specify
   SET_TO_CLIENT_TIME4, the server is to use its current time for the
   SETATTR operation.

If server and client times differ, programs that compare client time
to file times can break. A time maintenance protocol should be used
to limit client/server time skew.

Use of a COMPOUND containing a VERIFY operation specifying only the
change attribute, immediately followed by a SETATTR, provides a means
whereby a client may specify a request that emulates the
functionality of the SETATTR guard mechanism of NFSv3. Since the
function of the guard mechanism is to avoid changes to the file
attributes based on stale information, delays between checking of the
guard condition and the setting of the attributes have the potential
to compromise this function, as would the corresponding delay in the
NFSv4 emulation. Therefore, NFSv4 servers should take care to avoid
such delays, to the degree possible, when executing such a request.

If the server does not support an attribute as requested by the
client, the server should return NFS4ERR_ATTRNOTSUPP.

A mask of the attributes actually set is returned by SETATTR in all
cases. That mask MUST NOT include attribute bits not requested to be
set by the client. If the attribute masks in the request and reply
are equal, the status field in the reply MUST be NFS4_OK.

15.35. Operation 35: SETCLIENTID - Negotiate Client ID

15.35.1. SYNOPSIS

   client, callback, callback_ident -> clientid, setclientid_confirm

15.35.2. ARGUMENT

   struct SETCLIENTID4args {
           nfs_client_id4 client;
           cb_client4   callback;
           uint32_t   callback_ident;
   };

15.35.3. RESULT

```
struct SETCLIENTID4resok {
        clientid4       clientid;
        verifier4       setclientid_confirm;
};

union SETCLIENTID4res switch (nfsstat4 status) {
 case NFS4_OK:
        SETCLIENTID4resok       resok4;
 case NFS4ERR_CLID_INUSE:
        clientaddr4     client_using;
 default:
        void;
};
```

15.35.4. DESCRIPTION

The client uses the SETCLIENTID operation to notify the server of its
intention to use a particular client identifier, callback, and
callback_ident for subsequent requests that entail creating lock,
share reservation, and delegation state on the server. Upon
successful completion the server will return a shorthand client ID
which, if confirmed via a separate step, will be used in subsequent
file locking and file open requests. Confirmation of the client ID
must be done via the SETCLIENTID_CONFIRM operation to return the
client ID and setclientid_confirm values, as verifiers, to the
server. The reason why two verifiers are necessary is that it is
possible to use SETCLIENTID and SETCLIENTID_CONFIRM to modify the
callback and callback_ident information but not the shorthand client
ID. In that event, the setclientid_confirm value is effectively the
only verifier.

The callback information provided in this operation will be used if
the client is provided an open delegation at a future point.
Therefore, the client must correctly reflect the program and port
numbers for the callback program at the time SETCLIENTID is used.

The callback_ident value is used by the server on the callback. The
client can leverage the callback_ident to eliminate the need for more
than one callback RPC program number, while still being able to
determine which server is initiating the callback.

15.35.5. IMPLEMENTATION

To understand how to implement SETCLIENTID, make the following
notations. Let:

x be the value of the client.id subfield of the SETCLIENTID4args
   structure.

v be the value of the client.verifier subfield of the
   SETCLIENTID4args structure.

c be the value of the client ID field returned in the
   SETCLIENTID4resok structure.

k represent the value combination of the fields callback and
   callback_ident fields of the SETCLIENTID4args structure.

s be the setclientid_confirm value returned in the SETCLIENTID4resok
   structure.

{ v, x, c, k, s } be a quintuple for a client record. A client
   record is confirmed if there has been a SETCLIENTID_CONFIRM
   operation to confirm it. Otherwise it is unconfirmed. An
   unconfirmed record is established by a SETCLIENTID call.

Since SETCLIENTID is a non-idempotent operation, let us assume that
the server is implementing the duplicate request cache (DRC).

When the server gets a SETCLIENTID { v, x, k } request, it processes
it in the following manner.

o It first looks up the request in the DRC. If there is a hit, it
   returns the result cached in the DRC. The server does NOT remove
   client state (locks, shares, delegations) nor does it modify any
   recorded callback and callback_ident information for client { x }.

   For any DRC miss, the server takes the client ID string x, and
   searches for client records for x that the server may have
   recorded from previous SETCLIENTID calls. For any confirmed
   record with the same id string x, if the recorded principal does
   not match that of SETCLIENTID call, then the server returns a
   NFS4ERR_CLID_INUSE error.

   For brevity of discussion, the remaining description of the
   processing assumes that there was a DRC miss, and that where the
   server has previously recorded a confirmed record for client x,
   the aforementioned principal check has successfully passed.

o The server checks if it has recorded a confirmed record for { v,
   x, c, l, s }, where l may or may not equal k. If so, and since the
   id verifier v of the request matches that which is confirmed and
   recorded, the server treats this as a probable callback
   information update and records an unconfirmed { v, x, c, k, t }

and leaves the confirmed { v, x, c, l, s } in place, such that t
!= s. It does not matter if k equals l or not. Any pre-existing
unconfirmed { v, x, c, *, * } is removed.

The server returns { c, t }. It is indeed returning the old
clientid4 value c, because the client apparently only wants to
update callback value k to value l. It's possible this request is
one from the Byzantine router that has stale callback information,
but this is not a problem. The callback information update is
only confirmed if followed up by a SETCLIENTID_CONFIRM { c, t }.

The server awaits confirmation of k via SETCLIENTID_CONFIRM { c, t
}.

The server does NOT remove client (lock/share/delegation) state
for x.

o   The server has previously recorded a confirmed { u, x, c, l, s }
    record such that v != u, l may or may not equal k, and has not
    recorded any unconfirmed { *, x, *, *, * } record for x. The
    server records an unconfirmed { v, x, d, k, t } (d != c, t != s).

    The server returns { d, t }.

    The server awaits confirmation of { d, k } via SETCLIENTID_CONFIRM
    { d, t }.

    The server does NOT remove client (lock/share/delegation) state
    for x.

o   The server has previously recorded a confirmed { u, x, c, l, s }
    record such that v != u, l may or may not equal k, and recorded an
    unconfirmed { w, x, d, m, t } record such that c != d, t != s, m
    may or may not equal k, m may or may not equal l, and k may or may
    not equal l. Whether w == v or w != v makes no difference. The
    server simply removes the unconfirmed { w, x, d, m, t } record and
    replaces it with an unconfirmed { v, x, e, k, r } record, such
    that e != d, e != c, r != t, r != s.

    The server returns { e, r }.

    The server awaits confirmation of { e, k } via SETCLIENTID_CONFIRM
    { e, r }.

    The server does NOT remove client (lock/share/delegation) state
    for x.

   o The server has no confirmed { *, x, *, *, * } for x. It may or may
      not have recorded an unconfirmed { u, x, c, l, s }, where l may or
      may not equal k, and u may or may not equal v. Any unconfirmed
      record { u, x, c, l, * }, regardless whether u == v or l == k, is
      replaced with an unconfirmed record { v, x, d, k, t } where d !=
      c, t != s.

      The server returns { d, t }.

      The server awaits confirmation of { d, k } via SETCLIENTID_CONFIRM
      { d, t }. The server does NOT remove client (lock/share/
      delegation) state for x.

   The server generates the clientid and setclientid_confirm values and
   must take care to ensure that these values are extremely unlikely to
   ever be regenerated.

15.36. Operation 36: SETCLIENTID_CONFIRM - Confirm Client ID

15.36.1. SYNOPSIS

      clientid, setclientid_confirm -> -

15.36.2. ARGUMENT

   struct SETCLIENTID_CONFIRM4args {
           clientid4   clientid;
           verifier4   setclientid_confirm;
   };

15.36.3. RESULT

   struct SETCLIENTID_CONFIRM4res {
           nfsstat4   status;
   };

15.36.4. DESCRIPTION

   This operation is used by the client to confirm the results from a
   previous call to SETCLIENTID. The client provides the server
   supplied (from a SETCLIENTID response) client ID. The server
   responds with a simple status of success or failure.

15.36.5. IMPLEMENTATION

   The client must use the SETCLIENTID_CONFIRM operation to confirm the
   following two distinct cases:

   o The client's use of a new shorthand client identifier (as returned
      from the server in the response to SETCLIENTID), a new callback
      value (as specified in the arguments to SETCLIENTID) and a new
      callback_ident (as specified in the arguments to SETCLIENTID)
      value. The client's use of SETCLIENTID_CONFIRM in this case also
      confirms the removal of any of the client's previous relevant
      leased state. Relevant leased client state includes byte-range
      locks, share reservations, and where the server does not support
      the CLAIM_DELEGATE_PREV claim type, delegations. If the server
      supports CLAIM_DELEGATE_PREV, then SETCLIENTID_CONFIRM MUST NOT
      remove delegations for this client; relevant leased client state
      would then just include byte-range locks and share reservations.

   o The client's re-use of an old, previously confirmed, shorthand
      client identifier, a new callback value, and a new callback_ident
      value. The client's use of SETCLIENTID_CONFIRM in this case MUST
      NOT result in the removal of any previous leased state (locks,
      share reservations, and delegations)

   We use the same notation and definitions for v, x, c, k, s, and
   unconfirmed and confirmed client records as introduced in the
   description of the SETCLIENTID operation. The arguments to
   SETCLIENTID_CONFIRM are indicated by the notation { c, s }, where c
   is a value of type clientid4, and s is a value of type verifier4
   corresponding to the setclientid_confirm field.

   As with SETCLIENTID, SETCLIENTID_CONFIRM is a non-idempotent
   operation, and we assume that the server is implementing the
   duplicate request cache (DRC).

   When the server gets a SETCLIENTID_CONFIRM { c, s } request, it
   processes it in the following manner.

   o It first looks up the request in the DRC. If there is a hit, it
      returns the result cached in the DRC. The server does not remove
      any relevant leased client state nor does it modify any recorded
      callback and callback_ident information for client { x } as
      represented by the shorthand value c.

   For a DRC miss, the server checks for client records that match the
   shorthand value c. The processing cases are as follows:

o The server has recorded an unconfirmed { v, x, c, k, s } record
   and a confirmed { v, x, c, l, t } record, such that s != t. If
   the principals of the records do not match that of the
   SETCLIENTID_CONFIRM, the server returns NFS4ERR_CLID_INUSE, and no
   relevant leased client state is removed and no recorded callback
   and callback_ident information for client { x } is changed.
   Otherwise, the confirmed { v, x, c, l, t } record is removed and
   the unconfirmed { v, x, c, k, s } is marked as confirmed, thereby
   modifying recorded and confirmed callback and callback_ident
   information for client { x }.

   The server does not remove any relevant leased client state.

   The server returns NFS4_OK.

o The server has not recorded an unconfirmed { v, x, c, *, * } and
   has recorded a confirmed { v, x, c, *, s }. If the principals of
   the record and of SETCLIENTID_CONFIRM do not match, the server
   returns NFS4ERR_CLID_INUSE without removing any relevant leased
   client state and without changing recorded callback and
   callback_ident values for client { x }.

   If the principals match, then what has likely happened is that the
   client never got the response from the SETCLIENTID_CONFIRM, and
   the DRC entry has been purged. Whatever the scenario, since the
   principals match, as well as { c, s } matching a confirmed record,
   the server leaves client x's relevant leased client state intact,
   leaves its callback and callback_ident values unmodified, and
   returns NFS4_OK.

o The server has not recorded a confirmed { *, *, c, *, * }, and has
   recorded an unconfirmed { *, x, c, k, s }. Even if this is a
   retry from client, nonetheless the client's first
   SETCLIENTID_CONFIRM attempt was not received by the server. Retry
   or not, the server doesn't know, but it processes it as if were a
   first try. If the principal of the unconfirmed { *, x, c, k, s }
   record mismatches that of the SETCLIENTID_CONFIRM request the
   server returns NFS4ERR_CLID_INUSE without removing any relevant
   leased client state.

   Otherwise, the server records a confirmed { *, x, c, k, s }. If
   there is also a confirmed { *, x, d, *, t }, the server MUST
   remove the client x's relevant leased client state, and overwrite
   the callback state with k. The confirmed record { *, x, d, *, t }
   is removed.

   Server returns NFS4_OK.

   o The server has no record of a confirmed or unconfirmed { *, *, c,
      *, s }. The server returns NFS4ERR_STALE_CLIENTID. The server
      does not remove any relevant leased client state, nor does it
      modify any recorded callback and callback_ident information for
      any client.

   The server needs to cache unconfirmed { v, x, c, k, s } client
   records and await for some time their confirmation. As should be
   clear from the record processing discussions for SETCLIENTID and
   SETCLIENTID_CONFIRM, there are cases where the server does not
   deterministically remove unconfirmed client records. To avoid
   running out of resources, the server is not required to hold
   unconfirmed records indefinitely. One strategy the server might use
   is to set a limit on how many unconfirmed client records it will
   maintain, and then when the limit would be exceeded, remove the
   oldest record. Another strategy might be to remove an unconfirmed
   record when some amount of time has elapsed. The choice of the
   amount of time is fairly arbitrary but it is surely no higher than
   the server's lease time period. Consider that leases need to be
   renewed before the lease time expires via an operation from the
   client. If the client cannot issue a SETCLIENTID_CONFIRM after a
   SETCLIENTID before a period of time equal to that of a lease expires,
   then the client is unlikely to be able maintain state on the server
   during steady state operation.

   If the client does send a SETCLIENTID_CONFIRM for an unconfirmed
   record that the server has already deleted, the client will get
   NFS4ERR_STALE_CLIENTID back. If so, the client should then start
   over, and send SETCLIENTID to reestablish an unconfirmed client
   record and get back an unconfirmed client ID and setclientid_confirm
   verifier. The client should then send the SETCLIENTID_CONFIRM to
   confirm the client ID.

   SETCLIENTID_CONFIRM does not establish or renew a lease. However, if
   SETCLIENTID_CONFIRM removes relevant leased client state, and that
   state does not include existing delegations, the server MUST allow
   the client a period of time no less than the value of lease_time
   attribute, to reclaim, (via the CLAIM_DELEGATE_PREV claim type of the
   OPEN operation) its delegations before removing unreclaimed
   delegations.

15.37. Operation 37: VERIFY - Verify Same Attributes

15.37.1. SYNOPSIS

   (cfh), fattr -> -

15.37.2. ARGUMENT

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

15.37.3. RESULT

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

15.37.4. DESCRIPTION

The VERIFY operation is used to verify that attributes have a value
assumed by the client before proceeding with following operations in
the compound request. If any of the attributes do not match then the
error NFS4ERR_NOT_SAME must be returned. The current filehandle
retains its value after successful completion of the operation.

15.37.5. IMPLEMENTATION

One possible use of the VERIFY operation is the following compound
sequence. With this the client is attempting to verify that the file
being removed will match what the client expects to be removed. This
sequence can help prevent the unintended deletion of a file.

```
  PUTFH (directory filehandle)
  LOOKUP (file name)
  VERIFY (filehandle == fh)
  PUTFH (directory filehandle)
  REMOVE (file name)
```

This sequence does not prevent a second client from removing and
creating a new file in the middle of this sequence but it does help
avoid the unintended result.

In the case that a RECOMMENDED attribute is specified in the VERIFY
operation and the server does not support that attribute for the file
system object, the error NFS4ERR_ATTRNOTSUPP is returned to the
client.

When the attribute rdattr_error or any write-only attribute (e.g.,
time_modify_set) is specified, the error NFS4ERR_INVAL is returned to
the client.

15.38. Operation 38: WRITE - Write to File

15.38.1. SYNOPSIS

    (cfh), stateid, offset, stable, data -> count, committed, writeverf

15.38.2. ARGUMENT

```
   enum stable_how4 {
           UNSTABLE4        = 0,
           DATA_SYNC4       = 1,
           FILE_SYNC4       = 2
   };

   struct WRITE4args {
           /* CURRENT_FH: file */
           stateid4         stateid;
           offset4          offset;
           stable_how4      stable;
           opaque           data<>;
   };
```

15.38.3. RESULT

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

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

15.38.4. DESCRIPTION

   The WRITE operation is used to write data to a regular file. The
   target file is specified by the current filehandle. The offset
   specifies the offset where the data should be written. An offset of
   0 (zero) specifies that the write should start at the beginning of
   the file. The count, as encoded as part of the opaque data
   parameter, represents the number of bytes of data that are to be
   written. If the count is 0 (zero), the WRITE will succeed and return

a count of 0 (zero) subject to permissions checking. The server may
choose to write fewer bytes than requested by the client.

Part of the write request is a specification of how the write is to
be performed. The client specifies with the stable parameter the
method of how the data is to be processed by the server. If stable
is FILE_SYNC4, the server must commit the data written plus all file
system metadata to stable storage before returning results. This
corresponds to the NFS version 2 protocol semantics. Any other
behavior constitutes a protocol violation. If stable is DATA_SYNC4,
then the server must commit all of the data to stable storage and
enough of the metadata to retrieve the data before returning. The
server implementer is free to implement DATA_SYNC4 in the same
fashion as FILE_SYNC4, but with a possible performance drop. If
stable is UNSTABLE4, the server is free to commit any part of the
data and the metadata to stable storage, including all or none,
before returning a reply to the client. There is no guarantee
whether or when any uncommitted data will subsequently be committed
to stable storage. The only guarantees made by the server are that
it will not destroy any data without changing the value of verf and
that it will not commit the data and metadata at a level less than
that requested by the client.

The stateid value for a WRITE request represents a value returned
from a previous byte-range lock or share reservation request or the
stateid associated with a delegation. The stateid is used by the
server to verify that the associated share reservation and any byte-
range locks are still valid and to update lease timeouts for the
client.

Upon successful completion, the following results are returned. The
count result is the number of bytes of data written to the file. The
server may write fewer bytes than requested. If so, the actual
number of bytes written starting at location, offset, is returned.

The server also returns an indication of the level of commitment of
the data and metadata via committed. If the server committed all
data and metadata to stable storage, committed should be set to
FILE_SYNC4. If the level of commitment was at least as strong as
DATA_SYNC4, then committed should be set to DATA_SYNC4. Otherwise,
committed must be returned as UNSTABLE4. If stable was FILE4_SYNC,
then committed must also be FILE_SYNC4: anything else constitutes a
protocol violation. If stable was DATA_SYNC4, then committed may be
FILE_SYNC4 or DATA_SYNC4: anything else constitutes a protocol
violation. If stable was UNSTABLE4, then committed may be either
FILE_SYNC4, DATA_SYNC4, or UNSTABLE4.

The final portion of the result is the write verifier. The write
verifier is a cookie that the client can use to determine whether the
server has changed instance (boot) state between a call to WRITE and
a subsequent call to either WRITE or COMMIT. This cookie must be
consistent during a single instance of the NFSv4 protocol service and
must be unique between instances of the NFSv4 protocol server, where
uncommitted data may be lost.

If a client writes data to the server with the stable argument set to
UNSTABLE4 and the reply yields a committed response of DATA_SYNC4 or
UNSTABLE4, the client will follow up some time in the future with a
COMMIT operation to synchronize outstanding asynchronous data and
metadata with the server's stable storage, barring client error. It
is possible that due to client crash or other error that a subsequent
COMMIT will not be received by the server.

For a WRITE using the special anonymous stateid, the server MAY allow
the WRITE to be serviced subject to mandatory file locks or the
current share deny modes for the file. For a WRITE using the special
READ bypass stateid, the server MUST NOT allow the WRITE operation to
bypass locking checks at the server and is treated exactly the same
as if the anonymous stateid were used.

On success, the current filehandle retains its value.

15.38.5. IMPLEMENTATION

It is possible for the server to write fewer bytes of data than
requested by the client. In this case, the server should not return
an error unless no data was written at all. If the server writes
less than the number of bytes specified, the client should issue
another WRITE to write the remaining data.

It is assumed that the act of writing data to a file will cause the
time_modified of the file to be updated. However, the time_modified
of the file should not be changed unless the contents of the file are
changed. Thus, a WRITE request with count set to 0 should not cause
the time_modified of the file to be updated.

The definition of stable storage has been historically a point of
contention. The following expected properties of stable storage may
help in resolving design issues in the implementation. Stable
storage is persistent storage that survives:

1. Repeated power failures.

2. Hardware failures (of any board, power supply, etc.).

3. Repeated software crashes, including reboot cycle.

This definition does not address failure of the stable storage module itself.

The verifier is defined to allow a client to detect different instances of an NFSv4 protocol server over which cached, uncommitted data may be lost. In the most likely case, the verifier allows the client to detect server reboots. This information is required so that the client can safely determine whether the server could have lost cached data. If the server fails unexpectedly and the client has uncommitted data from previous WRITE requests (done with the stable argument set to UNSTABLE4 and in which the result committed was returned as UNSTABLE4 as well) it may not have flushed cached data to stable storage. The burden of recovery is on the client and the client will need to retransmit the data to the server.

A suggested verifier would be to use the time that the server was booted or the time the server was last started (if restarting the server without a reboot results in lost buffers).

The committed field in the results allows the client to do more effective caching. If the server is committing all WRITE requests to stable storage, then it should return with committed set to FILE_SYNC4, regardless of the value of the stable field in the arguments. A server that uses an NVRAM accelerator may choose to implement this policy. The client can use this to increase the effectiveness of the cache by discarding cached data that has already been committed on the server.

Some implementations may return NFS4ERR_NOSPC instead of NFS4ERR_DQUOT when a user's quota is exceeded. In the case that the current filehandle is a directory, the server will return NFS4ERR_ISDIR. If the current filehandle is not a regular file or a directory, the server will return NFS4ERR_INVAL.

If mandatory file locking is on for the file, and corresponding record of the data to be written file is read or write locked by an owner that is not associated with the stateid, the server will return NFS4ERR_LOCKED. If so, the client must check if the owner corresponding to the stateid used with the WRITE operation has a conflicting read lock that overlaps with the region that was to be written. If the stateid's owner has no conflicting read lock, then the client should try to get the appropriate write byte-range lock via the LOCK operation before re-attempting the WRITE. When the WRITE completes, the client should release the byte-range lock via LOCKU.

   If the stateid's owner had a conflicting read lock, then the client
   has no choice but to return an error to the application that
   attempted the WRITE. The reason is that since the stateid's owner
   had a read lock, the server either attempted to temporarily
   effectively upgrade this read lock to a write lock, or the server has
   no upgrade capability. If the server attempted to upgrade the read
   lock and failed, it is pointless for the client to re-attempt the
   upgrade via the LOCK operation, because there might be another client
   also trying to upgrade. If two clients are blocked trying upgrade
   the same lock, the clients deadlock. If the server has no upgrade
   capability, then it is pointless to try a LOCK operation to upgrade.

15.39. Operation 39: RELEASE_LOCKOWNER - Release Lockowner State

15.39.1. SYNOPSIS

      lock-owner -> ()

15.39.2. ARGUMENT

      struct RELEASE_LOCKOWNER4args {
              lock_owner4   lock_owner;
      };

15.39.3. RESULT

      struct RELEASE_LOCKOWNER4res {
              nfsstat4   status;
      };

15.39.4. DESCRIPTION

   This operation is used to notify the server that the lock_owner is no
   longer in use by the client and that future client requests will not
   reference this lock_owner. This allows the server to release cached
   state related to the specified lock_owner. If file locks, associated
   with the lock_owner, are held at the server, the error
   NFS4ERR_LOCKS_HELD will be returned and no further action will be
   taken.

15.39.5. IMPLEMENTATION

   The client may choose to use this operation to ease the amount of
   server state that is held. Information that can be released when a
   RELEASE_LOCKOWNER is done includes the specified lock-owner string,

the seqid associated with the lock-owner, any saved reply for the lock-owner, and any lock stateids associated with that lock-owner.

Depending on the behavior of applications at the client, it may be important for the client to use this operation since the server has certain obligations with respect to holding a reference to lock-owner-associated state as long as an associated file is open. Therefore, if the client knows for certain that the lock_owner will no longer be used, either to reference existing lock stateids associated with the lock-owner to create new ones, it should use RELEASE_LOCKOWNER.

15.40. Operation 10044: ILLEGAL - Illegal operation

15.40.1. SYNOPSIS

```
<null> -> ()
```

15.40.2. ARGUMENT

```
void;
```

15.40.3. RESULT

```
struct ILLEGAL4res {
        nfsstat4        status;
};
```

15.40.4. DESCRIPTION

This operation is a place holder for encoding a result to handle the case of the client sending an operation code within COMPOUND that is not supported. See Section 15.2.4 for more details.

The status field of ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

15.40.5. IMPLEMENTATION

A client will probably not send an operation with code OP_ILLEGAL but if it does, the response will be ILLEGAL4res just as it would be with any other invalid operation code. Note that if the server gets an illegal operation code that is not OP_ILLEGAL, and if the server checks for legal operation codes during the XDR decode phase, then the ILLEGAL4res would not be returned.

16. NFSv4 Callback Procedures

   The procedures used for callbacks are defined in the following
   sections. In the interest of clarity, the terms "client" and
   "server" refer to NFS clients and servers, despite the fact that for
   an individual callback RPC, the sense of these terms would be
   precisely the opposite.

   [RFC Editor: prior to publishing this document as an RFC, please have
   every Section that has a title of "Procedure X:" or "Operation Y:"
   start at the top of a new page.]

16.1. Procedure 0: CB_NULL - No Operation

16.1.1. SYNOPSIS

   <null>

16.1.2. ARGUMENT

   void;

16.1.3. RESULT

   void;

16.1.4. DESCRIPTION

   Standard NULL procedure. Void argument, void response. Even though
   there is no direct functionality associated with this procedure, the
   server will use CB_NULL to confirm the existence of a path for RPCs
   from server to client.

16.2. Procedure 1: CB_COMPOUND - Compound Operations

16.2.1. SYNOPSIS

   compoundargs -> compoundres

16.2.2. ARGUMENT

```
   enum nfs_cb_opnum4 {
           OP_CB_GETATTR          = 3,
           OP_CB_RECALL           = 4,
           OP_CB_ILLEGAL          = 10044
   };
```

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

   struct CB_COMPOUND4args {
           utf8str_cs       tag;
           uint32_t         minorversion;
           uint32_t         callback_ident;
           nfs_cb_argop4    argarray<>;
   };
```

16.2.3. RESULT

```
   union nfs_cb_resop4 switch (unsigned resop) {
    case OP_CB_GETATTR:   CB_GETATTR4res  opcbgetattr;
    case OP_CB_RECALL:    CB_RECALL4res   opcbrecall;
    case OP_CB_ILLEGAL:   CB_ILLEGAL4res  opcbillegal;
   };

   struct CB_COMPOUND4res {
           nfsstat4         status;
           utf8str_cs       tag;
           nfs_cb_resop4    resarray<>;
   };
```

16.2.4. DESCRIPTION

   The CB_COMPOUND procedure is used to combine one or more of the
   callback procedures into a single RPC request. The main callback RPC
   program has two main procedures: CB_NULL and CB_COMPOUND. All other
   operations use the CB_COMPOUND procedure as a wrapper.

   In the processing of the CB_COMPOUND procedure, the client may find
   that it does not have the available resources to execute any or all
   of the operations within the CB_COMPOUND sequence. In this case, the
   error NFS4ERR_RESOURCE will be returned for the particular operation
   within the CB_COMPOUND procedure where the resource exhaustion
   occurred. This assumes that all previous operations within the
   CB_COMPOUND sequence have been evaluated successfully.

   Contained within the CB_COMPOUND results is a 'status' field. This
   status must be equivalent to the status of the last operation that
   was executed within the CB_COMPOUND procedure. Therefore, if an

operation incurred an error then the 'status' value will be the same
error value as is being returned for the operation that failed.

For the definition of the "tag" field, see Section 15.2.

The value of callback_ident is supplied by the client during
SETCLIENTID. The server must use the client supplied callback_ident
during the CB_COMPOUND to allow the client to properly identify the
server.

Illegal operation codes are handled in the same way as they are
handled for the COMPOUND procedure.

16.2.5. IMPLEMENTATION

The CB_COMPOUND procedure is used to combine individual operations
into a single RPC request. The client interprets each of the
operations in turn. If an operation is executed by the client and
the status of that operation is NFS4_OK, then the next operation in
the CB_COMPOUND procedure is executed. The client continues this
process until there are no more operations to be executed or one of
the operations has a status value other than NFS4_OK.

16.2.6. Operation 3: CB_GETATTR - Get Attributes

16.2.6.1. SYNOPSIS

   fh, attr_request -> attrmask, attr_vals

16.2.6.2. ARGUMENT

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

16.2.6.3. RESULT

```
struct CB_GETATTR4resok {
        fattr4  obj_attributes;
};

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

16.2.6.4. DESCRIPTION

   The CB_GETATTR operation is used by the server to obtain the current
   modified state of a file that has been OPEN_DELEGATE_WRITE delegated.
   The attributes size and change are the only ones guaranteed to be
   serviced by the client. See Section 10.4.3 for a full description of
   how the client and server are to interact with the use of CB_GETATTR.

   If the filehandle specified is not one for which the client holds a
   OPEN_DELEGATE_WRITE delegation, an NFS4ERR_BADHANDLE error is
   returned.

16.2.6.5. IMPLEMENTATION

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

16.2.7. Operation 4: CB_RECALL - Recall an Open Delegation

16.2.7.1. SYNOPSIS

     stateid, truncate, fh -> ()

16.2.7.2. ARGUMENT

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

16.2.7.3. RESULT

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

16.2.7.4. DESCRIPTION

   The CB_RECALL operation is used to begin the process of recalling an
   open delegation and returning it to the server.

   The truncate flag is used to optimize recall for a file which is
   about to be truncated to zero. When it is set, the client is freed
   of obligation to propagate modified data for the file to the server,
   since this data is irrelevant.

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

   If the stateid specified is not one corresponding to an open
   delegation for the file specified by the filehandle, an
   NFS4ERR_BAD_STATEID is returned.

16.2.7.5. IMPLEMENTATION

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

16.2.8. Operation 10044: CB_ILLEGAL - Illegal Callback Operation

16.2.8.1. SYNOPSIS

   <null> -> ()

16.2.8.2. ARGUMENT

   void;

16.2.8.3. RESULT

```
   /*
    * CB_ILLEGAL: Response for illegal operation numbers
    */
   struct CB_ILLEGAL4res {
           nfsstat4        status;
   };
```

16.2.8.4. DESCRIPTION

   This operation is a place-holder for encoding a result to handle the
   case of the client sending an operation code within COMPOUND that is
   not supported. See Section 15.2.4 for more details.

   The status field of CB_ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

16.2.8.5. IMPLEMENTATION

   A server will probably not send an operation with code OP_CB_ILLEGAL
   but if it does, the response will be CB_ILLEGAL4res just as it would
   be with any other invalid operation code. Note that if the client
   gets an illegal operation code that is not OP_ILLEGAL, and if the
   client checks for legal operation codes during the XDR decode phase,
   then the CB_ILLEGAL4res would not be returned.

17. Security Considerations

   NFS has historically used a model where, from an authentication
   perspective, the client was the entire machine, or at least the
   source IP address of the machine. The NFS server relied on the NFS
   client to make the proper authentication of the end-user. The NFS
   server in turn shared its files only to specific clients, as
   identified by the client's source IP address. Given this model, the
   AUTH_SYS RPC security flavor simply identified the end-user using the
   client to the NFS server. When processing NFS responses, the client
   ensured that the responses came from the same IP address and port
   number that the request was sent to. While such a model is easy to
   implement and simple to deploy and use, it is certainly not a safe
   model. Thus, NFSv4 mandates that implementations support a security
   model that uses end to end authentication, where an end-user on a
   client mutually authenticates (via cryptographic schemes that do not
   expose passwords or keys in the clear on the network) to a principal
   on an NFS server. Consideration should also be given to the
   integrity and privacy of NFS requests and responses. The issues of
   end to end mutual authentication, integrity, and privacy are
   discussed as part of Section 3.

When an NFSv4 mandated security model is used and a security
principal or an NFSv4 name in user@dns_domain form needs to be
translated to or from a local representation as described in
Section 5.9, the translation SHOULD be done in a secure manner that
preserves the integrity of the translation. For communication with a
name service such as LDAP ([RFC4511]), this means employing a
security service that uses authentication and data integrity.
Kerberos and Transport Layer Security (TLS) ([RFC5246]) are examples
of such a security service.

Note that being REQUIRED to implement does not mean REQUIRED to use;
AUTH_SYS can be used by NFSv4 clients and servers. However, AUTH_SYS
is merely an OPTIONAL security flavor in NFSv4, and so
interoperability via AUTH_SYS is not assured.

For reasons of reduced administration overhead, better performance
and/or reduction of CPU utilization, users of NFSv4 implementations
may choose to not use security mechanisms that enable integrity
protection on each remote procedure call and response. The use of
mechanisms without integrity leaves the customer vulnerable to an
attacker in between the NFS client and server that modifies the RPC
request and/or the response. While implementations are free to
provide the option to use weaker security mechanisms, there are two
operations in particular that warrant the implementation overriding
user choices.

The first such operation is SECINFO. It is recommended that the
client issue the SECINFO call such that it is protected with a
security flavor that has integrity protection, such as RPCSEC_GSS
with a security triple that uses either rpc_gss_svc_integrity or
rpc_gss_svc_privacy (rpc_gss_svc_privacy includes integrity
protection) service. Without integrity protection encapsulating
SECINFO and therefore its results, an attacker in the middle could
modify results such that the client might select a weaker algorithm
in the set allowed by server, making the client and/or server
vulnerable to further attacks.

The second operation that SHOULD use integrity protection is any
GETATTR for the fs_locations attribute. The attack has two steps.
First the attacker modifies the unprotected results of some operation
to return NFS4ERR_MOVED. Second, when the client follows up with a
GETATTR for the fs_locations attribute, the attacker modifies the
results to cause the client to migrate its traffic to a server
controlled by the attacker.

Because the operations SETCLIENTID/SETCLIENTID_CONFIRM are
responsible for the release of client state, it is imperative that
the principal used for these operations is checked against and match

with the previous use of these operations. See Section 9.1.1 for
further discussion.

Unicode in the form of UTF-8 is used for file component names (i.e.,
both directory and file components), as well as the owner and
owner_group attributes; other character sets may also be allowed for
file component names. String processing (e.g., Unicode
normalization) raises security concerns for string comparison - see
Sections 5.9 and 12 for further discussion and see [RFC6943] for
related identifier comparison security considerations. File
component names are identifiers with respect to the identifier
comparison discussion in [RFC6943] because they are used to identify
the objects to which ACLs are applied, see Section 6.

18. IANA Considerations

This section uses terms that are defined in [RFC5226].

18.1. Named Attribute Definitions

IANA has created a registry called the "NFSv4 Named Attribute
Definitions Registry" for [RFC3530] and [RFC5661]. This section
introduces no new changes, but it does recap the intent.

The NFSv4 protocol supports the association of a file with zero or
more named attributes. The name space identifiers for these
attributes are defined as string names. The protocol does not define
the specific assignment of the name space for these file attributes.
The IANA registry promotes interoperability where common interests
exist. While application developers are allowed to define and use
attributes as needed, they are encouraged to register the attributes
with IANA.

Such registered named attributes are presumed to apply to all minor
versions of NFSv4, including those defined subsequently to the
registration. Where the named attribute is intended to be limited
with regard to the minor versions for which they are not be used, the
assignment in registry will clearly state the applicable limits.

The registry is to be maintained using the Specification Required
policy as defined in Section 4.1 of [RFC5226].

Under the NFSv4 specification, the name of a named attribute can in
theory be up to $2^{32} - 1$ bytes in length, but in practice NFSv4
clients and servers will be unable to handle a string that long.
IANA should reject any assignment request with a named attribute that
exceeds 128 UTF-8 characters. To give IESG the flexibility to set up
bases of assignment of Experimental Use and Standards Action, the

prefixes of "EXPE" and "STDS" are Reserved. The zero length named
attribute name is Reserved.

The prefix "PRIV" is allocated for Private Use. A site that wants to
make use of unregistered named attributes without risk of conflicting
with an assignment in IANA's registry should use the prefix "PRIV" in
all of its named attributes.

Because some NFSv4 clients and servers have case insensitive
semantics, the fifteen additional lower case and mixed case
permutations of each of "EXPE", "PRIV", and "STDS", are Reserved
(e.g. "expe", "expE", "exPe", etc. are Reserved). Similarly, IANA
must not allow two assignments that would conflict if both named
attributes were converted to a common case.

The registry of named attributes is a list of assignments, each
containing three fields for each assignment.

1. A US-ASCII string name that is the actual name of the attribute.
    This name must be unique. This string name can be 1 to 128 UTF-8
    characters long.

2. A reference to the specification of the named attribute. The
    reference can consume up to 256 bytes (or more if IANA permits).

3. The point of contact of the registrant. The point of contact can
    consume up to 256 bytes (or more if IANA permits).

18.1.1. Initial Registry

There is no initial registry.

18.1.2. Updating Registrations

The registrant is always permitted to update the point of contact
field. To make any other change will require Expert Review or IESG
Approval.

19. References

19.1. Normative References

[RFC20]   Cerf, V., "ASCII format for network interchange", RFC 20,
            October 1969.

[RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
            Requirement Levels", March 1997.

   [RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol
              Specification", RFC 2203, September 1997.

   [RFC2743] Linn, J., "Generic Security Service Application Program
              Interface Version 2, Update 1", RFC 2743, January 2000.

   [RFC3490] Faltstrom, P., Hoffman, P., and A. Costello,
              "Internationalizing Domain Names in Applications (IDNA)",
              RFC 3490, March 2003.

   [RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode
              for Internationalized Domain Names in Applications
              (IDNA)", RFC 3492, March 2003.

   [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO
              10646", STD 63, RFC 3629, November 2003.

   [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
              IANA Considerations Section in RFCs", BCP 26, RFC 5226,
              May 2008.

   [RFC5403] Eisler, M., "RPCSEC_GSS Version 2", RFC 5403, February
              2009.

   [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol
              Specification Version 2", RFC 5531, May 2009.

   [RFC5665] Eisler, M., Ed., "IANA Considerations for Remote Procedure
              Call (RPC) Network Identifiers and Universal Address
              Formats", RFC 5665, January 2010.

   [RFC5890] Klensin, J., "Internationalized Domain Names in
              Applications (IDNA): Definitions and Document Framework",
              RFC 5890, August 2010.

   [RFC5891] Klensin, J., "Internationalized Domain Names in
              Applications (IDNA): Protocol", RFC 5891, August 2010.

   [RFC6649] Astrand, L. and T. Yu, "Deprecate DES, RC4-HMAC-EXP, and
              Other Weak Cryptographic Algorithms in Kerberos", RFC
              6649, July 2012.

   [RFCNFSv4XDR]
              Haynes, T. and D. Noveck, "NFSv4 Version 0 XDR
              Description", draft-ietf-nfsv4-rfc3530bis-dot-x-23 (work
              in progress), Dec 2014.

[SPECIALCASING]
          The Unicode Consortium, "SpecialCasing-6.3.0.txt", Unicode
          Character Database , September 2013,
          <http://www.unicode.org/Public/6.3.0/ucd/
          SpecialCasing.txt>.

[UNICODE] The Unicode Consortium, "The Unicode Standard, Version
          6.3.0", September 2013,
          <http://www.unicode.org/versions/Unicode6.3.0/>.

[openg_symlink]
          The Open Group, "Section 3.372 of Chapter 3 of Base
          Definitions of The Open Group Base Specifications Issue 6,
          IEEE Std 1003.1, 2004 Edition, HTML Version
          (www.opengroup.org), ISBN 1931624232", 2004.

19.2. Informative References

[Chet]   Juszczak, C., "Improving the Performance and Correctness
          of an NFS Server", USENIX Conference Proceedings , June
          1990.

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

[IESG_ERRATA]
          IESG, "IESG Processing of RFC Errata for the IETF Stream",
          July 2008.

[MS-SMB]   Microsoft Corporation, , "Server Message Block (SMB)
          Protocol Specification", MS-SMB 17.0, November 2009.

[P1003.1e]
          Institute of Electrical and Electronics Engineers, Inc.,
          "IEEE Draft P1003.1e", 1997.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC
          793, September 1981.

[RFC1094] Nowicki, B., "NFS: Network File System Protocol
          specification", RFC 1094, March 1989.

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

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

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

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

   [RFC2224] Callaghan, B., "NFS URL Scheme", RFC 2224, October 1997.

   [RFC2623] Eisler, M., "NFS Version 2 and Version 3 Security Issues
              and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5",
              RFC 2623, June 1999.

   [RFC2624] Shepler, S., "NFS Version 4 Design Considerations", RFC
              2624, June 1999.

   [RFC2755] Chiu, A., Eisler, M., and B. Callaghan, "Security
              Negotiation for WebNFS", RFC 2755, January 2000.

   [RFC3010] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
              Beame, C., Eisler, M., and D. Noveck, "Network File System
              (NFS) version 4 Protocol", RFC 3010, December 2000.

   [RFC3232] Reynolds, J., "Assigned Numbers: RFC 1700 is Replaced by
              an On-line Database", RFC 3232, January 2002.

   [RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
              Beame, C., Eisler, M., and D. Noveck, "Network File System
              (NFS) version 4 Protocol", RFC 3530, April 2003.

   [RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos
              Version 5 Generic Security Service Application Program
              Interface (GSS-API) Mechanism: Version 2", RFC 4121, July
              2005.

   [RFC4178] Zhu, L., Leach, P., Jaganathan, K., and W. Ingersoll, "The
              Simple and Protected Generic Security Service Application
              Program Interface (GSS-API) Negotiation Mechanism", RFC
              4178, October 2005.

   [RFC4506] Eisler, M., "XDR: External Data Representation Standard",
              RFC 4506, May 2006.

   [RFC4511] Sermersheim, J., "Lightweight Directory Access Protocol
              (LDAP): The Protocol", RFC 4511, June 2006.

   [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
              (TLS) Protocol Version 1.2", RFC 5246, August 2008.

      [RFC5661] Shepler, S., Eisler, M., and D. Noveck, "Network File
                 System (NFS) Version 4 Minor Version 1 Protocol", RFC
                 5661, January 2010.

      [RFC6365] Hoffman, P. and J. Klensin, "Terminology Used in
                 Internationalization in the IETF", BCP 166, RFC 6365,
                 September 2011.

      [RFC6943] Thaler, D., "Issues in Identifier Comparison for Security
                 Purposes", RFC 6943, May 2013.

      [fcntl]   The Open Group, "Section 'fcntl()' of System Interfaces of
                 The Open Group Base Specifications Issue 6 IEEE Std
                 1003.1, 2004 Edition, HTML Version (www.opengroup.org),
                 ISBN 1931624232", 2004.

      [fsync]   The Open Group, "Section 'fsync()' of System Interfaces of
                 The Open Group Base Specifications Issue 6 IEEE Std
                 1003.1, 2004 Edition, HTML Version (www.opengroup.org),
                 ISBN 1931624232", 2004.

      [getpwnam]
                 The Open Group, "Section 'getpwnam()' of System Interfaces
                 of The Open Group Base Specifications Issue 6 IEEE Std
                 1003.1, 2004 Edition, HTML Version (www.opengroup.org),
                 ISBN 1931624232", 2004.

      [read_api]
                 The Open Group, "Section 'read()' of System Interfaces of
                 The Open Group Base Specifications Issue 6, IEEE Std
                 1003.1, 2004 Edition", 2004.

      [readdir_api]
                 The Open Group, "Section 'readdir()' of System Interfaces
                 of The Open Group Base Specifications Issue 6, IEEE Std
                 1003.1, 2004 Edition", 2004.

      [unlink]   The Open Group, "Section 'unlink()' of System Interfaces
                 of The Open Group Base Specifications Issue 6 IEEE Std
                 1003.1, 2004 Edition, HTML Version (www.opengroup.org),
                 ISBN 1931624232", 2004.

      [write_api]
                 The Open Group, "Section 'write()' of System Interfaces of
                 The Open Group Base Specifications Issue 6, IEEE Std
                 1003.1, 2004 Edition", 2004.

   [xnfs]   The Open Group, "Protocols for Interworking: XNFS, Version
               3W, ISBN 1-85912-184-5", February 1998.

Appendix A.   Acknowledgments

   A bis is certainly built on the shoulders of the first attempt.
   Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow,
   Carl Beame, Mike Eisler, and David Noveck are responsible for a great
   deal of the effort in this work.

   Tom Haynes would like to thank NetApp, Inc. for its funding of his
   time on this project.

   Rob Thurlow clarified how a client should contact a new server if a
   migration has occurred.

   David Black, Nico Williams, Mike Eisler, Trond Myklebust, James
   Lentini, and Mike Kupfer read many drafts of Section 12 and
   contributed numerous useful suggestions, without which the necessary
   revision of that section for this document would not have been
   possible.

   Peter Staubach read almost all of the drafts of Section 12 leading to
   the published result and his numerous comments were always useful and
   contributed substantially to improving the quality of the final
   result.

   Peter Saint-Andre was gracious enough to read the last draft of
   Section 12 and provided some key insight as to the concerns of the
   Internationalization community.

   James Lentini graciously read the rewrite of Section 8 and his
   comments were vital in improving the quality of that effort.

   Rob Thurlow, Sorin Faibish, James Lentini, Bruce Fields, and Trond
   Myklebust were faithful attendants of the biweekly triage meeting and
   accepted many an action item.

   Bruce Fields was a good sounding board for both the Third Edge
   Condition and Courtesy Locks in general. He was also the leading
   advocate of stamping out backport issues from [RFC5661].

   Marcel Telka was a champion of straightening out the difference
   between a lock-owner and an open-owner. He has also been diligent in
   reviewing the final document.

   Benjamin Kaduk reminded us that DES is dead and Nico Williams helped
   us close the lid on the coffin.

   Elwyn Davies provided a very thorough and engaging Gen-ART review,
   thanks!

Appendix B. RFC Editor Notes

   [RFC Editor: please remove this section prior to publishing this
   document as an RFC]

   [RFC Editor: prior to publishing this document as an RFC, please
   replace all occurrences of RFCNFSv4XDR with RFCxxxx where xxxx is the
   RFC number assigned to the XDR document.]

   [RFC Editor: Please note that there is also a reference entry that
   needs to be modified for the companion document.]

   [RFC Editor: prior to publishing this document as an RFC, please have
   every top level subsection of both Section 15 and Section 16 that has
   a title of "Procedure X:" or "Operation Y:" start at the top of a new
   page.]

Authors' Addresses

   Thomas Haynes (editor)
   Primary Data, Inc.
   4300 El Camino Real Ste 100
   Los Altos, CA 94022
   USA

   Phone: +1 408 215 1519
   Email: thomas.haynes@primarydata.com

   David Noveck (editor)
   Dell
   300 Innovative Way
   Nashua, NH 03062
   US

   Phone: +1 781 572 8038
   Email: dave_noveck@dell.com

Network File System (NFS) Version 4
External Data Representation Standard (XDR) Description
draft-ietf-nfsv4-rfc3530bis-dot-x-24.txt

Abstract

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

   RFC3530bis formally obsoleting RFC 3530. This document, together
   with RFC3530bis replaces RFC 3530 as the definition of the NFS
   version 4 protocol.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on June 7, 2015.

Copyright Notice

Table of Contents

1. XDR Description of NFSv4.0

   This document contains the XDR ([RFC4506]) description of NFSv4.0
   protocol ([RFCNFSv4]).

   The XDR description is provided in this document in a way that makes
   it simple for the reader to extract it into ready to compile form.
   The reader can feed this document in the following shell script to
   produce the machine readable XDR description of NFSv4.0:

```
    #!/bin/sh
    grep "^  *///" | sed 's?^  */// ??' | sed 's?^  *///$??'
```

I.e. if the above script is stored in a file called "extract.sh", and
this document is in a file called "spec.txt", then the reader can do:

```
 sh extract.sh < spec.txt > nfs4_prot.x
```

The effect of the script is to remove leading white space from each
line, plus a sentinel sequence of "///".

The XDR description, with the sentinel sequence follows:

```
 /// /*
 ///  * This file was machine generated for
 ///  *  [RFCNFSv4]
 ///  *  [RFC Editor: please update the citation on the line above]
 ///  * Last updated Thu Dec  4 11:31:21 PST 2014
 ///  */
 /// /*
 ///  *  Copyright (C) The IETF Trust (2009-2011)
 ///  *  All Rights Reserved.
 ///  *
 ///  *  Copyright (C) The Internet Society (1998-2011).
 ///  *  All Rights Reserved.
 ///  */
 ///
 /// /*
 ///  *       nfs4_prot.x
 ///  *
 ///  */
 ///
 /// /*
 ///  * Basic typedefs for RFC 1832 data type definitions
 ///  */
 /// /*
 ///  * typedef int           int32_t;
 ///  * typedef unsigned int uint32_t;
 ///  * typedef hyper                 int64_t;
 ///  * typedef unsigned hyper       uint64_t;
 ///  */
 ///
 /// /*
 ///  * Sizes
 ///  */
 /// const NFS4_FHSIZE             = 128;
 /// const NFS4_VERIFIER_SIZE       = 8;
```

```
///  const NFS4_OTHER_SIZE          = 12;
///  const NFS4_OPAQUE_LIMIT        = 1024;
///
///  const NFS4_INT64_MAX           = 0x7fffffffffffffff;
///  const NFS4_UINT64_MAX          = 0xffffffffffffffff;
///  const NFS4_INT32_MAX           = 0x7fffffff;
///  const NFS4_UINT32_MAX          = 0xffffffff;
///
///
///  /*
///   * File types
///   */
///  enum nfs_ftype4 {
///          NF4REG = 1,      /* Regular File */
///          NF4DIR = 2,      /* Directory */
///          NF4BLK = 3,      /* Special File - block device */
///          NF4CHR = 4,      /* Special File - character device */
///          NF4LNK = 5,      /* Symbolic Link */
///          NF4SOCK = 6,     /* Special File - socket */
///          NF4FIFO = 7,     /* Special File - fifo */
///          NF4ATTRDIR
///                  = 8,     /* Attribute Directory */
///          NF4NAMEDATTR
///                  = 9      /* Named Attribute */
///  };
///
///  /*
///   * Error status
///   */
///  enum nfsstat4 {
///   NFS4_OK               = 0,    /* everything is okay     */
///   NFS4ERR_PERM          = 1,    /* caller not privileged  */
///   NFS4ERR_NOENT         = 2,    /* no such file/directory */
///   NFS4ERR_IO            = 5,    /* hard I/O error         */
///   NFS4ERR_NXIO          = 6,    /* no such device         */
///   NFS4ERR_ACCESS        = 13,   /* access denied          */
///   NFS4ERR_EXIST         = 17,   /* file already exists    */
///   NFS4ERR_XDEV          = 18,   /* different filesystems  */
///   /* Unused/reserved       19 */
///   NFS4ERR_NOTDIR        = 20,   /* should be a directory  */
///   NFS4ERR_ISDIR         = 21,   /* should not be directory */
///   NFS4ERR_INVAL         = 22,   /* invalid argument       */
///   NFS4ERR_FBIG          = 27,   /* file exceeds server max */
///   NFS4ERR_NOSPC         = 28,   /* no space on filesystem */
///   NFS4ERR_ROFS          = 30,   /* read-only filesystem   */
///   NFS4ERR_MLINK         = 31,   /* too many hard links    */
///   NFS4ERR_NAMETOOLONG   = 63,   /* name exceeds server max */
///   NFS4ERR_NOTEMPTY      = 66,   /* directory not empty    */
```

```
///   NFS4ERR_DQUOT             = 69,   /* hard quota limit reached*/
///   NFS4ERR_STALE             = 70,   /* file no longer exists   */
///   NFS4ERR_BADHANDLE         = 10001,/* Illegal filehandle      */
///   NFS4ERR_BAD_COOKIE        = 10003,/* READDIR cookie is stale */
///   NFS4ERR_NOTSUPP           = 10004,/* operation not supported */
///   NFS4ERR_TOOSMALL          = 10005,/* response limit exceeded */
///   NFS4ERR_SERVERFAULT       = 10006,/* undefined server error  */
///   NFS4ERR_BADTYPE           = 10007,/* type invalid for CREATE */
///   NFS4ERR_DELAY             = 10008,/* file "busy" - retry     */
///   NFS4ERR_SAME              = 10009,/* nverify says attrs same */
///   NFS4ERR_DENIED            = 10010,/* lock unavailable        */
///   NFS4ERR_EXPIRED           = 10011,/* lock lease expired      */
///   NFS4ERR_LOCKED            = 10012,/* I/O failed due to lock  */
///   NFS4ERR_GRACE             = 10013,/* in grace period         */
///   NFS4ERR_FHEXPIRED         = 10014,/* filehandle expired      */
///   NFS4ERR_SHARE_DENIED      = 10015,/* share reserve denied    */
///   NFS4ERR_WRONGSEC          = 10016,/* wrong security flavor   */
///   NFS4ERR_CLID_INUSE        = 10017,/* clientid in use         */
///   NFS4ERR_RESOURCE          = 10018,/* resource exhaustion     */
///   NFS4ERR_MOVED             = 10019,/* filesystem relocated    */
///   NFS4ERR_NOFILEHANDLE      = 10020,/* current FH is not set   */
///   NFS4ERR_MINOR_VERS_MISMATCH = 10021,/* minor vers not supp */
///   NFS4ERR_STALE_CLIENTID    = 10022,/* server has rebooted     */
///   NFS4ERR_STALE_STATEID     = 10023,/* server has rebooted     */
///   NFS4ERR_OLD_STATEID       = 10024,/* state is out of sync    */
///   NFS4ERR_BAD_STATEID       = 10025,/* incorrect stateid       */
///   NFS4ERR_BAD_SEQID         = 10026,/* request is out of seq.  */
///   NFS4ERR_NOT_SAME          = 10027,/* verify - attrs not same */
///   NFS4ERR_LOCK_RANGE        = 10028,/* lock range not supported*/
///   NFS4ERR_SYMLINK           = 10029,/* should be file/directory*/
///   NFS4ERR_RESTOREFH         = 10030,/* no saved filehandle     */
///   NFS4ERR_LEASE_MOVED       = 10031,/* some filesystem moved   */
///   NFS4ERR_ATTRNOTSUPP       = 10032,/* recommended attr not sup*/
///   NFS4ERR_NO_GRACE          = 10033,/* reclaim outside of grace*/
///   NFS4ERR_RECLAIM_BAD       = 10034,/* reclaim error at server */
///   NFS4ERR_RECLAIM_CONFLICT  = 10035,/* conflict on reclaim     */
///   NFS4ERR_BADXDR            = 10036,/* XDR decode failed       */
///   NFS4ERR_LOCKS_HELD        = 10037,/* file locks held at CLOSE*/
///   NFS4ERR_OPENMODE          = 10038,/* conflict in OPEN and I/O*/
///   NFS4ERR_BADOWNER          = 10039,/* owner translation bad   */
///   NFS4ERR_BADCHAR           = 10040,/* utf-8 char not supported*/
///   NFS4ERR_BADNAME           = 10041,/* name not supported      */
///   NFS4ERR_BAD_RANGE         = 10042,/* lock range not supported*/
///   NFS4ERR_LOCK_NOTSUPP      = 10043,/* no atomic up/downgrade  */
///   NFS4ERR_OP_ILLEGAL        = 10044,/* undefined operation     */
///   NFS4ERR_DEADLOCK          = 10045,/* file locking deadlock   */
///   NFS4ERR_FILE_OPEN         = 10046,/* open file blocks op.    */
///   NFS4ERR_ADMIN_REVOKED     = 10047,/* lockowner state revoked */
```

```
///  NFS4ERR_CB_PATH_DOWN   = 10048 /* callback path down        */
/// };
///
/// /*
///  * Basic data types
///  */
/// typedef opaque  attrlist4<>;
/// typedef uint32_t        bitmap4<>;
/// typedef uint64_t        changeid4;
/// typedef uint64_t        clientid4;
/// typedef uint32_t        count4;
/// typedef uint64_t        length4;
/// typedef uint32_t        mode4;
/// typedef uint64_t        nfs_cookie4;
/// typedef opaque  nfs_fh4<NFS4_FHSIZE>;
/// typedef uint32_t        nfs_lease4;
/// typedef uint64_t        offset4;
/// typedef uint32_t        qop4;
/// typedef opaque  sec_oid4<>;
/// typedef uint32_t        seqid4;
/// typedef opaque  utf8string<>;
/// typedef utf8string      utf8str_cis;
/// typedef utf8string      utf8str_cs;
/// typedef utf8string      utf8str_mixed;
/// typedef utf8str_cs      component4;
/// typedef opaque  linktext4<>;
/// typedef utf8string      ascii_REQUIRED4;
/// typedef component4      pathname4<>;
/// typedef uint64_t        nfs_lockid4;
/// typedef opaque  verifier4[NFS4_VERIFIER_SIZE];
///
///
/// /*
///  * Timeval
///  */
/// struct nfstime4 {
///         int64_t         seconds;
///         uint32_t        nseconds;
/// };
///
/// enum time_how4 {
///         SET_TO_SERVER_TIME4 = 0,
///         SET_TO_CLIENT_TIME4 = 1
/// };
///
/// union settime4 switch (time_how4 set_it) {
///  case SET_TO_CLIENT_TIME4:
///         nfstime4         time;
```

```
///   default:
///           void;
/// };
///
///
/// /*
///  * File attribute definitions
///  */
///
/// /*
///  * FSID structure for major/minor
///  */
/// struct fsid4 {
///           uint64_t        major;
///           uint64_t        minor;
/// };
///
///
/// /*
///  * Filesystem locations attribute for relocation/migration
///  */
/// struct fs_location4 {
///           utf8str_cis             server<>;
///           pathname4               rootpath;
/// };
///
/// struct fs_locations4 {
///           pathname4       fs_root;
///           fs_location4    locations<>;
/// };
///
///
/// /*
///  * Various Access Control Entry definitions
///  */
///
/// /*
///  * Mask that indicates which Access Control Entries
///  * are supported. Values for the fattr4_aclsupport attribute.
///  */
/// const ACL4_SUPPORT_ALLOW_ACL    = 0x00000001;
/// const ACL4_SUPPORT_DENY_ACL     = 0x00000002;
/// const ACL4_SUPPORT_AUDIT_ACL    = 0x00000004;
/// const ACL4_SUPPORT_ALARM_ACL    = 0x00000008;
///
///
/// typedef uint32_t        acetype4;
///
```

```
///
/// /*
///  * acetype4 values, others can be added as needed.
///  */
/// const ACE4_ACCESS_ALLOWED_ACE_TYPE     = 0x00000000;
/// const ACE4_ACCESS_DENIED_ACE_TYPE      = 0x00000001;
/// const ACE4_SYSTEM_AUDIT_ACE_TYPE       = 0x00000002;
/// const ACE4_SYSTEM_ALARM_ACE_TYPE       = 0x00000003;
///
///
///
/// /*
///  * ACE flag
///  */
/// typedef uint32_t        aceflag4;
///
///
/// /*
///  * ACE flag values
///  */
/// const ACE4_FILE_INHERIT_ACE            = 0x00000001;
/// const ACE4_DIRECTORY_INHERIT_ACE       = 0x00000002;
/// const ACE4_NO_PROPAGATE_INHERIT_ACE    = 0x00000004;
/// const ACE4_INHERIT_ONLY_ACE            = 0x00000008;
/// const ACE4_SUCCESSFUL_ACCESS_ACE_FLAG  = 0x00000010;
/// const ACE4_FAILED_ACCESS_ACE_FLAG      = 0x00000020;
/// const ACE4_IDENTIFIER_GROUP            = 0x00000040;
///
///
///
/// /*
///  * ACE mask
///  */
/// typedef uint32_t        acemask4;
///
///
/// /*
///  * ACE mask values
///  */
/// const ACE4_READ_DATA         = 0x00000001;
/// const ACE4_LIST_DIRECTORY    = 0x00000001;
/// const ACE4_WRITE_DATA        = 0x00000002;
/// const ACE4_ADD_FILE          = 0x00000002;
/// const ACE4_APPEND_DATA       = 0x00000004;
/// const ACE4_ADD_SUBDIRECTORY  = 0x00000004;
/// const ACE4_READ_NAMED_ATTRS  = 0x00000008;
/// const ACE4_WRITE_NAMED_ATTRS = 0x00000010;
/// const ACE4_EXECUTE           = 0x00000020;
```

```
    /// const ACE4_DELETE_CHILD         = 0x00000040;
    /// const ACE4_READ_ATTRIBUTES      = 0x00000080;
    /// const ACE4_WRITE_ATTRIBUTES     = 0x00000100;
    ///
    /// const ACE4_DELETE               = 0x00010000;
    /// const ACE4_READ_ACL             = 0x00020000;
    /// const ACE4_WRITE_ACL            = 0x00040000;
    /// const ACE4_WRITE_OWNER          = 0x00080000;
    /// const ACE4_SYNCHRONIZE          = 0x00100000;
    ///
    ///
    /// /*
    ///  * ACE4_GENERIC_READ -- defined as combination of
    ///  *      ACE4_READ_ACL |
    ///  *      ACE4_READ_DATA |
    ///  *      ACE4_READ_ATTRIBUTES |
    ///  *      ACE4_SYNCHRONIZE
    ///  */
    ///
    /// const ACE4_GENERIC_READ = 0x00120081;
    ///
    /// /*
    ///  * ACE4_GENERIC_WRITE -- defined as combination of
    ///  *      ACE4_READ_ACL |
    ///  *      ACE4_WRITE_DATA |
    ///  *      ACE4_WRITE_ATTRIBUTES |
    ///  *      ACE4_WRITE_ACL |
    ///  *      ACE4_APPEND_DATA |
    ///  *      ACE4_SYNCHRONIZE
    ///  */
    /// const ACE4_GENERIC_WRITE = 0x00160106;
    ///
    ///
    /// /*
    ///  * ACE4_GENERIC_EXECUTE -- defined as combination of
    ///  *      ACE4_READ_ACL
    ///  *      ACE4_READ_ATTRIBUTES
    ///  *      ACE4_EXECUTE
    ///  *      ACE4_SYNCHRONIZE
    ///  */
    /// const ACE4_GENERIC_EXECUTE = 0x001200A0;
    ///
    ///
    /// /*
    ///  * Access Control Entry definition
    ///  */
    /// struct nfsace4 {
    ///         acetype4                type;
```

```
///          aceflag4                  flag;
///          acemask4                  access_mask;
///          utf8str_mixed             who;
/// };
///
///
/// /*
///  * Field definitions for the fattr4_mode attribute
///  */
/// const MODE4_SUID = 0x800;  /* set user id on execution */
/// const MODE4_SGID = 0x400;  /* set group id on execution */
/// const MODE4_SVTX = 0x200;  /* save text even after use */
/// const MODE4_RUSR = 0x100;  /* read permission: owner */
/// const MODE4_WUSR = 0x080;  /* write permission: owner */
/// const MODE4_XUSR = 0x040;  /* execute permission: owner */
/// const MODE4_RGRP = 0x020;  /* read permission: group */
/// const MODE4_WGRP = 0x010;  /* write permission: group */
/// const MODE4_XGRP = 0x008;  /* execute permission: group */
/// const MODE4_ROTH = 0x004;  /* read permission: other */
/// const MODE4_WOTH = 0x002;  /* write permission: other */
/// const MODE4_XOTH = 0x001;  /* execute permission: other */
///
///
/// /*
///  * Special data/attribute associated with
///  * file types NF4BLK and NF4CHR.
///  */
/// struct specdata4 {
/// uint32_t specdata1; /* major device number */
/// uint32_t specdata2; /* minor device number */
/// };
///
///
/// /*
///  * Values for fattr4_fh_expire_type
///  */
/// const   FH4_PERSISTENT          = 0x00000000;
/// const   FH4_NOEXPIRE_WITH_OPEN  = 0x00000001;
/// const   FH4_VOLATILE_ANY        = 0x00000002;
/// const   FH4_VOL_MIGRATION       = 0x00000004;
/// const   FH4_VOL_RENAME          = 0x00000008;
///
///
/// typedef bitmap4                 fattr4_supported_attrs;
/// typedef nfs_ftype4              fattr4_type;
/// typedef uint32_t                fattr4_fh_expire_type;
/// typedef changeid4               fattr4_change;
/// typedef uint64_t                fattr4_size;
```

```
    /// typedef bool                 fattr4_link_support;
    /// typedef bool                 fattr4_symlink_support;
    /// typedef bool                 fattr4_named_attr;
    /// typedef fsid4                fattr4_fsid;
    /// typedef bool                 fattr4_unique_handles;
    /// typedef nfs_lease4           fattr4_lease_time;
    /// typedef nfsstat4             fattr4_rdattr_error;
    ///
    /// typedef nfsace4              fattr4_acl<>;
    /// typedef uint32_t             fattr4_aclsupport;
    /// typedef bool                 fattr4_archive;
    /// typedef bool                 fattr4_cansettime;
    /// typedef bool                 fattr4_case_insensitive;
    /// typedef bool                 fattr4_case_preserving;
    /// typedef bool                 fattr4_chown_restricted;
    /// typedef uint64_t             fattr4_fileid;
    /// typedef uint64_t             fattr4_files_avail;
    /// typedef nfs_fh4              fattr4_filehandle;
    /// typedef uint64_t             fattr4_files_free;
    /// typedef uint64_t             fattr4_files_total;
    /// typedef fs_locations4        fattr4_fs_locations;
    /// typedef bool                 fattr4_hidden;
    /// typedef bool                 fattr4_homogeneous;
    /// typedef uint64_t             fattr4_maxfilesize;
    /// typedef uint32_t             fattr4_maxlink;
    /// typedef uint32_t             fattr4_maxname;
    /// typedef uint64_t             fattr4_maxread;
    /// typedef uint64_t             fattr4_maxwrite;
    /// typedef ascii_REQUIRED4      fattr4_mimetype;
    /// typedef mode4                fattr4_mode;
    /// typedef uint64_t             fattr4_mounted_on_fileid;
    /// typedef bool                 fattr4_no_trunc;
    /// typedef uint32_t             fattr4_numlinks;
    /// typedef utf8str_mixed        fattr4_owner;
    /// typedef utf8str_mixed        fattr4_owner_group;
    /// typedef uint64_t             fattr4_quota_avail_hard;
    /// typedef uint64_t             fattr4_quota_avail_soft;
    /// typedef uint64_t             fattr4_quota_used;
    /// typedef specdata4            fattr4_rawdev;
    /// typedef uint64_t             fattr4_space_avail;
    /// typedef uint64_t             fattr4_space_free;
    /// typedef uint64_t             fattr4_space_total;
    /// typedef uint64_t             fattr4_space_used;
    /// typedef bool                 fattr4_system;
    /// typedef nfstime4             fattr4_time_access;
    /// typedef settime4             fattr4_time_access_set;
    /// typedef nfstime4             fattr4_time_backup;
    /// typedef nfstime4             fattr4_time_create;
```

```
///  typedef nfstime4                  fattr4_time_delta;
///  typedef nfstime4                  fattr4_time_metadata;
///  typedef nfstime4                  fattr4_time_modify;
///  typedef settime4                  fattr4_time_modify_set;
///
///
///  /*
///   * Mandatory Attributes
///   */
///  const FATTR4_SUPPORTED_ATTRS   = 0;
///  const FATTR4_TYPE              = 1;
///  const FATTR4_FH_EXPIRE_TYPE    = 2;
///  const FATTR4_CHANGE            = 3;
///  const FATTR4_SIZE              = 4;
///  const FATTR4_LINK_SUPPORT      = 5;
///  const FATTR4_SYMLINK_SUPPORT   = 6;
///  const FATTR4_NAMED_ATTR        = 7;
///  const FATTR4_FSID              = 8;
///  const FATTR4_UNIQUE_HANDLES    = 9;
///  const FATTR4_LEASE_TIME        = 10;
///  const FATTR4_RDATTR_ERROR      = 11;
///  const FATTR4_FILEHANDLE        = 19;
///
///  /*
///   * Recommended Attributes
///   */
///  const FATTR4_ACL               = 12;
///  const FATTR4_ACLSUPPORT        = 13;
///  const FATTR4_ARCHIVE           = 14;
///  const FATTR4_CANSETTIME        = 15;
///  const FATTR4_CASE_INSENSITIVE  = 16;
///  const FATTR4_CASE_PRESERVING   = 17;
///  const FATTR4_CHOWN_RESTRICTED  = 18;
///  const FATTR4_FILEID            = 20;
///  const FATTR4_FILES_AVAIL       = 21;
///  const FATTR4_FILES_FREE        = 22;
///  const FATTR4_FILES_TOTAL       = 23;
///  const FATTR4_FS_LOCATIONS      = 24;
///  const FATTR4_HIDDEN            = 25;
///  const FATTR4_HOMOGENEOUS       = 26;
///  const FATTR4_MAXFILESIZE       = 27;
///  const FATTR4_MAXLINK           = 28;
///  const FATTR4_MAXNAME           = 29;
///  const FATTR4_MAXREAD           = 30;
///  const FATTR4_MAXWRITE          = 31;
///  const FATTR4_MIMETYPE          = 32;
///  const FATTR4_MODE              = 33;
///  const FATTR4_NO_TRUNC          = 34;
```

```
/// const FATTR4_NUMLINKS          = 35;
/// const FATTR4_OWNER             = 36;
/// const FATTR4_OWNER_GROUP       = 37;
/// const FATTR4_QUOTA_AVAIL_HARD  = 38;
/// const FATTR4_QUOTA_AVAIL_SOFT  = 39;
/// const FATTR4_QUOTA_USED        = 40;
/// const FATTR4_RAWDEV            = 41;
/// const FATTR4_SPACE_AVAIL       = 42;
/// const FATTR4_SPACE_FREE        = 43;
/// const FATTR4_SPACE_TOTAL       = 44;
/// const FATTR4_SPACE_USED        = 45;
/// const FATTR4_SYSTEM            = 46;
/// const FATTR4_TIME_ACCESS       = 47;
/// const FATTR4_TIME_ACCESS_SET   = 48;
/// const FATTR4_TIME_BACKUP       = 49;
/// const FATTR4_TIME_CREATE       = 50;
/// const FATTR4_TIME_DELTA        = 51;
/// const FATTR4_TIME_METADATA     = 52;
/// const FATTR4_TIME_MODIFY       = 53;
/// const FATTR4_TIME_MODIFY_SET   = 54;
/// const FATTR4_MOUNTED_ON_FILEID = 55;
///
/// /*
///  * File attribute container
///  */
/// struct fattr4 {
///         bitmap4         attrmask;
///         attrlist4       attr_vals;
/// };
///
///
/// /*
///  * Change info for the client
///  */
/// struct change_info4 {
///         bool            atomic;
///         changeid4       before;
///         changeid4       after;
/// };
///
///
/// struct clientaddr4 {
///         /* see struct rpcb in RFC 1833 */
///         string r_netid<>;       /* network id */
///         string r_addr<>;        /* universal address */
/// };
///
///
```

```
/// /*
///  * Callback program info as provided by the client
/// */
/// struct cb_client4 {
///         unsigned int    cb_program;
///         clientaddr4     cb_location;
/// };
///
///
/// /*
///  * Stateid
/// */
/// struct stateid4 {
///         uint32_t        seqid;
///         opaque          other[NFS4_OTHER_SIZE];
/// };
///
/// /*
///  * Client ID
/// */
/// struct nfs_client_id4 {
///         verifier4       verifier;
///         opaque          id<NFS4_OPAQUE_LIMIT>;
/// };
///
///
/// struct open_owner4 {
///         clientid4       clientid;
///         opaque          owner<NFS4_OPAQUE_LIMIT>;
/// };
///
///
/// struct lock_owner4 {
///         clientid4       clientid;
///         opaque          owner<NFS4_OPAQUE_LIMIT>;
/// };
///
///
/// enum nfs_lock_type4 {
///         READ_LT         = 1,
///         WRITE_LT        = 2,
///         READW_LT        = 3,    /* blocking read */
///         WRITEW_LT       = 4     /* blocking write */
/// };
///
///
/// const ACCESS4_READ      = 0x00000001;
/// const ACCESS4_LOOKUP    = 0x00000002;
```

```
/// const ACCESS4_MODIFY   = 0x00000004;
/// const ACCESS4_EXTEND   = 0x00000008;
/// const ACCESS4_DELETE   = 0x00000010;
/// const ACCESS4_EXECUTE  = 0x00000020;
///
/// struct ACCESS4args {
///         /* CURRENT_FH: object */
///         uint32_t        access;
/// };
///
/// struct ACCESS4resok {
///         uint32_t        supported;
///         uint32_t        access;
/// };
///
/// union ACCESS4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         ACCESS4resok    resok4;
///  default:
///         void;
/// };
///
/// struct CLOSE4args {
///         /* CURRENT_FH: object */
///         seqid4          seqid;
///         stateid4        open_stateid;
/// };
///
/// union CLOSE4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         stateid4        open_stateid;
///  default:
///         void;
/// };
///
/// struct COMMIT4args {
///         /* CURRENT_FH: file */
///         offset4         offset;
///         count4          count;
/// };
///
/// struct COMMIT4resok {
///         verifier4       writeverf;
/// };
///
/// union COMMIT4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         COMMIT4resok    resok4;
```

```
///  default:
///          void;
/// };
///
/// union createtype4 switch (nfs_ftype4 type) {
///  case NF4LNK:
///          linktext4 linkdata;
///  case NF4BLK:
///  case NF4CHR:
///          specdata4 devdata;
///  case NF4SOCK:
///  case NF4FIFO:
///  case NF4DIR:
///          void;
///  default:
///          void;  /* server should return NFS4ERR_BADTYPE */
/// };
///
/// struct CREATE4args {
///          /* CURRENT_FH: directory for creation */
///          createtype4     objtype;
///          component4      objname;
///          fattr4          createattrs;
/// };
///
/// struct CREATE4resok {
///          change_info4    cinfo;
///          bitmap4         attrset;        /* attributes set */
/// };
///
/// union CREATE4res switch (nfsstat4 status) {
///  case NFS4_OK:
///          CREATE4resok resok4;
///  default:
///          void;
/// };
///
/// struct DELEGPURGE4args {
///          clientid4       clientid;
/// };
///
/// struct DELEGPURGE4res {
///          nfsstat4        status;
/// };
///
/// struct DELEGRETURN4args {
///          /* CURRENT_FH: delegated file */
///          stateid4        deleg_stateid;
```

```
/// };
///
/// struct DELEGRETURN4res {
///         nfsstat4        status;
/// };
///
/// struct GETATTR4args {
///         /* CURRENT_FH: directory or file */
///         bitmap4         attr_request;
/// };
///
/// struct GETATTR4resok {
///         fattr4          obj_attributes;
/// };
///
/// union GETATTR4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         GETATTR4resok  resok4;
///  default:
///         void;
/// };
///
/// struct GETFH4resok {
///         nfs_fh4         object;
/// };
///
/// union GETFH4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         GETFH4resok     resok4;
///  default:
///         void;
/// };
///
/// struct LINK4args {
///         /* SAVED_FH: source object */
///         /* CURRENT_FH: target directory */
///         component4      newname;
/// };
///
/// struct LINK4resok {
///         change_info4    cinfo;
/// };
///
/// union LINK4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         LINK4resok resok4;
///  default:
///         void;
```

```
/// };
///
/// /*
///  * For LOCK, transition from open_owner to new lock_owner
///  */
/// struct open_to_lock_owner4 {
///         seqid4          open_seqid;
///         stateid4        open_stateid;
///         seqid4          lock_seqid;
///         lock_owner4     lock_owner;
/// };
///
/// /*
///  * For LOCK, existing lock_owner continues to request file locks
///  */
/// struct exist_lock_owner4 {
///         stateid4        lock_stateid;
///         seqid4          lock_seqid;
/// };
///
/// union locker4 switch (bool new_lock_owner) {
///  case TRUE:
///         open_to_lock_owner4     open_owner;
///  case FALSE:
///         exist_lock_owner4       lock_owner;
/// };
///
/// /*
///  * LOCK/LOCKT/LOCKU: Record lock management
///  */
/// struct LOCK4args {
///         /* CURRENT_FH: file */
///         nfs_lock_type4  locktype;
///         bool            reclaim;
///         offset4         offset;
///         length4         length;
///         locker4         locker;
/// };
///
/// struct LOCK4denied {
///         offset4         offset;
///         length4         length;
///         nfs_lock_type4  locktype;
///         lock_owner4     owner;
/// };
///
/// struct LOCK4resok {
///         stateid4        lock_stateid;
```

```
    /// };
    ///
    /// union LOCK4res switch (nfsstat4 status) {
    ///  case NFS4_OK:
    ///          LOCK4resok      resok4;
    ///  case NFS4ERR_DENIED:
    ///          LOCK4denied     denied;
    ///  default:
    ///          void;
    /// };
    ///
    /// struct LOCKT4args {
    ///          /* CURRENT_FH: file */
    ///          nfs_lock_type4  locktype;
    ///          offset4         offset;
    ///          length4         length;
    ///          lock_owner4     owner;
    /// };
    ///
    /// union LOCKT4res switch (nfsstat4 status) {
    ///  case NFS4ERR_DENIED:
    ///          LOCK4denied     denied;
    ///  case NFS4_OK:
    ///          void;
    ///  default:
    ///          void;
    /// };
    ///
    /// struct LOCKU4args {
    ///          /* CURRENT_FH: file */
    ///          nfs_lock_type4  locktype;
    ///          seqid4          seqid;
    ///          stateid4        lock_stateid;
    ///          offset4         offset;
    ///          length4         length;
    /// };
    ///
    /// union LOCKU4res switch (nfsstat4 status) {
    ///  case   NFS4_OK:
    ///          stateid4        lock_stateid;
    ///  default:
    ///          void;
    /// };
    ///
    /// struct LOOKUP4args {
    ///          /* CURRENT_FH: directory */
    ///          component4      objname;
    /// };
```

```
///
/// struct LOOKUP4res {
///         /* CURRENT_FH: object */
///         nfsstat4        status;
/// };
///
/// struct LOOKUPP4res {
///         /* CURRENT_FH: directory */
///         nfsstat4        status;
/// };
///
/// struct NVERIFY4args {
///         /* CURRENT_FH: object */
///         fattr4          obj_attributes;
/// };
///
/// struct NVERIFY4res {
///         nfsstat4        status;
/// };
///
/// const OPEN4_SHARE_ACCESS_READ   = 0x00000001;
/// const OPEN4_SHARE_ACCESS_WRITE  = 0x00000002;
/// const OPEN4_SHARE_ACCESS_BOTH   = 0x00000003;
///
/// const OPEN4_SHARE_DENY_NONE     = 0x00000000;
/// const OPEN4_SHARE_DENY_READ     = 0x00000001;
/// const OPEN4_SHARE_DENY_WRITE    = 0x00000002;
/// const OPEN4_SHARE_DENY_BOTH     = 0x00000003;
/// /*
///  * Various definitions for OPEN
///  */
/// enum createmode4 {
///         UNCHECKED4      = 0,
///         GUARDED4        = 1,
///         EXCLUSIVE4      = 2
/// };
///
/// union createhow4 switch (createmode4 mode) {
///  case UNCHECKED4:
///  case GUARDED4:
///         fattr4          createattrs;
///  case EXCLUSIVE4:
///         verifier4       createverf;
/// };
///
/// enum opentype4 {
///         OPEN4_NOCREATE  = 0,
///         OPEN4_CREATE    = 1
```

```
///     };
///
///     union openflag4 switch (opentype4 opentype) {
///      case OPEN4_CREATE:
///             createhow4      how;
///      default:
///             void;
///     };
///
///     /* Next definitions used for OPEN delegation */
///     enum limit_by4 {
///             NFS_LIMIT_SIZE          = 1,
///             NFS_LIMIT_BLOCKS        = 2
///             /* others as needed */
///     };
///
///     struct nfs_modified_limit4 {
///             uint32_t        num_blocks;
///             uint32_t        bytes_per_block;
///     };
///
///     union nfs_space_limit4 switch (limit_by4 limitby) {
///      /* limit specified as file size */
///      case NFS_LIMIT_SIZE:
///             uint64_t                filesize;
///      /* limit specified by number of blocks */
///      case NFS_LIMIT_BLOCKS:
///             nfs_modified_limit4     mod_blocks;
///     } ;
///
///     enum open_delegation_type4 {
///             OPEN_DELEGATE_NONE      = 0,
///             OPEN_DELEGATE_READ      = 1,
///             OPEN_DELEGATE_WRITE     = 2
///     };
///
///     enum open_claim_type4 {
///             CLAIM_NULL              = 0,
///             CLAIM_PREVIOUS          = 1,
///             CLAIM_DELEGATE_CUR      = 2,
///             CLAIM_DELEGATE_PREV     = 3
///     };
///
///     struct open_claim_delegate_cur4 {
///             stateid4        delegate_stateid;
///             component4      file;
///     };
///
```

```
/// union open_claim4 switch (open_claim_type4 claim) {
///   /*
///    * No special rights to file.
///    * Ordinary OPEN of the specified file.
///    */
///   case CLAIM_NULL:
///           /* CURRENT_FH: directory */
///           component4      file;
///   /*
///    * Right to the file established by an
///    * open previous to server reboot. File
///    * identified by filehandle obtained at
///    * that time rather than by name.
///    */
///   case CLAIM_PREVIOUS:
///           /* CURRENT_FH: file being reclaimed */
///           open_delegation_type4   delegate_type;
///
///   /*
///    * Right to file based on a delegation
///    * granted by the server. File is
///    * specified by name.
///    */
///   case CLAIM_DELEGATE_CUR:
///           /* CURRENT_FH: directory */
///           open_claim_delegate_cur4        delegate_cur_info;
///
///   /*
///    * Right to file based on a delegation
///    * granted to a previous boot instance
///    * of the client.  File is specified by name.
///    */
///   case CLAIM_DELEGATE_PREV:
///            /* CURRENT_FH: directory */
///           component4      file_delegate_prev;
/// };
///
/// /*
///  * OPEN: Open a file, potentially receiving an open delegation
///  */
/// struct OPEN4args {
///         seqid4          seqid;
///         uint32_t        share_access;
///         uint32_t        share_deny;
///         open_owner4     owner;
///         openflag4       openhow;
///         open_claim4     claim;
/// };
```

```
/// 
/// struct open_read_delegation4 {
/// stateid4 stateid;     /* Stateid for delegation*/
/// bool     recall;      /* Pre-recalled flag for
///                          delegations obtained
///                          by reclaim (CLAIM_PREVIOUS) */
/// 
/// nfsace4 permissions; /* Defines users who don't
///                          need an ACCESS call to
///                          open for read */
/// };
/// 
/// struct open_write_delegation4 {
/// stateid4 stateid;        /* Stateid for delegation */
/// bool     recall;         /* Pre-recalled flag for
///                             delegations obtained
///                             by reclaim
///                             (CLAIM_PREVIOUS) */
/// 
/// nfs_space_limit4
///          space_limit; /* Defines condition that
///                          the client must check to
///                          determine whether the
///                          file needs to be flushed
///                          to the server on close.  */
/// 
/// nfsace4  permissions; /* Defines users who don't
///                          need an ACCESS call as
///                          part of a delegated
///                          open. */
/// };
/// 
/// union open_delegation4
/// switch (open_delegation_type4 delegation_type) {
///         case OPEN_DELEGATE_NONE:
///                 void;
///         case OPEN_DELEGATE_READ:
///                 open_read_delegation4 read;
///         case OPEN_DELEGATE_WRITE:
///                 open_write_delegation4 write;
/// };
/// 
/// /*
///  * Result flags
///  */
/// 
/// /* Client must confirm open */
/// const OPEN4_RESULT_CONFIRM     = 0x00000002;
```

```
/// /* Type of file locking behavior at the server */
/// const OPEN4_RESULT_LOCKTYPE_POSIX = 0x00000004;
///
/// struct OPEN4resok {
///  stateid4        stateid;       /* Stateid for open */
///  change_info4   cinfo;         /* Directory Change Info */
///  uint32_t       rflags;        /* Result flags */
///  bitmap4        attrset;       /* attribute set for create*/
///  open_delegation4 delegation; /* Info on any open
///                                  delegation */
/// };
///
/// union OPEN4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         /* CURRENT_FH: opened file */
///         OPEN4resok      resok4;
///  default:
///         void;
/// };
///
/// struct OPENATTR4args {
///         /* CURRENT_FH: object */
///         bool    createdir;
/// };
///
/// struct OPENATTR4res {
///         /* CURRENT_FH: named attr directory */
///         nfsstat4        status;
/// };
///
/// struct OPEN_CONFIRM4args {
///         /* CURRENT_FH: opened file */
///         stateid4        open_stateid;
///         seqid4          seqid;
/// };
///
/// struct OPEN_CONFIRM4resok {
///         stateid4        open_stateid;
/// };
///
/// union OPEN_CONFIRM4res switch (nfsstat4 status) {
///  case NFS4_OK:
///          OPEN_CONFIRM4resok      resok4;
///  default:
///          void;
/// };
///
/// struct OPEN_DOWNGRADE4args {
```

```
///            /* CURRENT_FH: opened file */
///            stateid4        open_stateid;
///            seqid4          seqid;
///            uint32_t        share_access;
///            uint32_t        share_deny;
/// };
///
/// struct OPEN_DOWNGRADE4resok {
///            stateid4        open_stateid;
/// };
///
/// union OPEN_DOWNGRADE4res switch(nfsstat4 status) {
///  case NFS4_OK:
///            OPEN_DOWNGRADE4resok    resok4;
///  default:
///            void;
/// };
///
/// struct PUTFH4args {
///            nfs_fh4         object;
/// };
///
/// struct PUTFH4res {
///            /* CURRENT_FH: */
///            nfsstat4        status;
/// };
///
/// struct PUTPUBFH4res {
///            /* CURRENT_FH: public fh */
///            nfsstat4        status;
/// };
///
/// struct PUTROOTFH4res {
///            /* CURRENT_FH: root fh */
///            nfsstat4        status;
/// };
///
/// struct READ4args {
///            /* CURRENT_FH: file */
///            stateid4        stateid;
///            offset4         offset;
///            count4          count;
/// };
///
/// struct READ4resok {
///            bool            eof;
///            opaque          data<>;
/// };
```

```
    ///
    /// union READ4res switch (nfsstat4 status) {
    ///  case NFS4_OK:
    ///          READ4resok     resok4;
    ///  default:
    ///          void;
    /// };
    ///
    /// struct READDIR4args {
    ///          /* CURRENT_FH: directory */
    ///          nfs_cookie4    cookie;
    ///          verifier4      cookieverf;
    ///          count4         dircount;
    ///          count4         maxcount;
    ///          bitmap4        attr_request;
    /// };
    ///
    /// struct entry4 {
    ///          nfs_cookie4    cookie;
    ///          component4     name;
    ///          fattr4         attrs;
    ///          entry4         *nextentry;
    /// };
    ///
    /// struct dirlist4 {
    ///          entry4         *entries;
    ///          bool           eof;
    /// };
    ///
    /// struct READDIR4resok {
    ///          verifier4      cookieverf;
    ///          dirlist4       reply;
    /// };
    ///
    ///
    /// union READDIR4res switch (nfsstat4 status) {
    ///  case NFS4_OK:
    ///          READDIR4resok  resok4;
    ///  default:
    ///          void;
    /// };
    ///
    ///
    /// struct READLINK4resok {
    ///          linktext4      link;
    /// };
    ///
    /// union READLINK4res switch (nfsstat4 status) {
```

```
///  case NFS4_OK:
///          READLINK4resok resok4;
///  default:
///          void;
/// };
///
/// struct REMOVE4args {
///          /* CURRENT_FH: directory */
///          component4      target;
/// };
///
/// struct REMOVE4resok {
///          change_info4    cinfo;
/// };
///
/// union REMOVE4res switch (nfsstat4 status) {
///  case NFS4_OK:
///          REMOVE4resok    resok4;
///  default:
///          void;
/// };
///
/// struct RENAME4args {
///          /* SAVED_FH: source directory */
///          component4      oldname;
///          /* CURRENT_FH: target directory */
///          component4      newname;
/// };
///
/// struct RENAME4resok {
///          change_info4    source_cinfo;
///          change_info4    target_cinfo;
/// };
///
/// union RENAME4res switch (nfsstat4 status) {
///  case NFS4_OK:
///          RENAME4resok    resok4;
///  default:
///          void;
/// };
///
/// struct RENEW4args {
///          clientid4       clientid;
/// };
///
/// struct RENEW4res {
///          nfsstat4        status;
/// };
```

```
///
/// struct RESTOREFH4res {
///           /* CURRENT_FH: value of saved fh */
///           nfsstat4        status;
/// };
///
/// struct SAVEFH4res {
///           /* SAVED_FH: value of current fh */
///           nfsstat4        status;
/// };
///
/// struct SECINFO4args {
///           /* CURRENT_FH: directory */
///           component4      name;
/// };
///
/// /*
///  * From RFC 2203
///  */
/// enum rpc_gss_svc_t {
///           RPC_GSS_SVC_NONE        = 1,
///           RPC_GSS_SVC_INTEGRITY   = 2,
///           RPC_GSS_SVC_PRIVACY     = 3
/// };
///
/// struct rpcsec_gss_info {
///           sec_oid4        oid;
///           qop4            qop;
///           rpc_gss_svc_t   service;
/// };
///
/// /* RPCSEC_GSS has a value of '6' - See RFC 2203 */
/// union secinfo4 switch (uint32_t flavor) {
///  case RPCSEC_GSS:
///           rpcsec_gss_info       flavor_info;
///  default:
///           void;
/// };
///
/// typedef secinfo4 SECINFO4resok<>;
///
/// union SECINFO4res switch (nfsstat4 status) {
///  case NFS4_OK:
///           SECINFO4resok resok4;
///  default:
///           void;
/// };
///
```

```
/// struct SETATTR4args {
///         /* CURRENT_FH: target object */
///         stateid4        stateid;
///         fattr4          obj_attributes;
/// };
///
/// struct SETATTR4res {
///         nfsstat4        status;
///         bitmap4         attrsset;
/// };
///
/// struct SETCLIENTID4args {
///         nfs_client_id4  client;
///         cb_client4      callback;
///         uint32_t        callback_ident;
/// };
///
/// struct SETCLIENTID4resok {
///         clientid4       clientid;
///         verifier4       setclientid_confirm;
/// };
///
/// union SETCLIENTID4res switch (nfsstat4 status) {
///  case NFS4_OK:
///          SETCLIENTID4resok       resok4;
///  case NFS4ERR_CLID_INUSE:
///          clientaddr4     client_using;
///  default:
///          void;
/// };
///
/// struct SETCLIENTID_CONFIRM4args {
///         clientid4       clientid;
///         verifier4       setclientid_confirm;
/// };
///
/// struct SETCLIENTID_CONFIRM4res {
///         nfsstat4        status;
/// };
///
/// struct VERIFY4args {
///         /* CURRENT_FH: object */
///         fattr4          obj_attributes;
/// };
///
/// struct VERIFY4res {
///         nfsstat4        status;
/// };
```

```
///
/// enum stable_how4 {
///         UNSTABLE4       = 0,
///         DATA_SYNC4      = 1,
///         FILE_SYNC4      = 2
/// };
///
/// struct WRITE4args {
///         /* CURRENT_FH: file */
///         stateid4        stateid;
///         offset4         offset;
///         stable_how4     stable;
///         opaque          data<>;
/// };
///
/// struct WRITE4resok {
///         count4          count;
///         stable_how4     committed;
///         verifier4       writeverf;
/// };
///
/// union WRITE4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         WRITE4resok     resok4;
///  default:
///         void;
/// };
///
/// struct RELEASE_LOCKOWNER4args {
///         lock_owner4     lock_owner;
/// };
///
/// struct RELEASE_LOCKOWNER4res {
///         nfsstat4        status;
/// };
///
/// struct ILLEGAL4res {
///         nfsstat4        status;
/// };
///
/// /*
///  * Operation arrays
///  */
///
/// enum nfs_opnum4 {
///  OP_ACCESS              = 3,
///  OP_CLOSE               = 4,
///  OP_COMMIT              = 5,
```

```
///   OP_CREATE              = 6,
///   OP_DELEGPURGE          = 7,
///   OP_DELEGRETURN         = 8,
///   OP_GETATTR             = 9,
///   OP_GETFH               = 10,
///   OP_LINK                = 11,
///   OP_LOCK                = 12,
///   OP_LOCKT               = 13,
///   OP_LOCKU               = 14,
///   OP_LOOKUP              = 15,
///   OP_LOOKUPP             = 16,
///   OP_NVERIFY             = 17,
///   OP_OPEN                = 18,
///   OP_OPENATTR            = 19,
///   OP_OPEN_CONFIRM        = 20,
///   OP_OPEN_DOWNGRADE      = 21,
///   OP_PUTFH               = 22,
///   OP_PUTPUBFH            = 23,
///   OP_PUTROOTFH           = 24,
///   OP_READ                = 25,
///   OP_READDIR             = 26,
///   OP_READLINK            = 27,
///   OP_REMOVE              = 28,
///   OP_RENAME              = 29,
///   OP_RENEW               = 30,
///   OP_RESTOREFH           = 31,
///   OP_SAVEFH              = 32,
///   OP_SECINFO             = 33,
///   OP_SETATTR             = 34,
///   OP_SETCLIENTID         = 35,
///   OP_SETCLIENTID_CONFIRM = 36,
///   OP_VERIFY              = 37,
///   OP_WRITE               = 38,
///   OP_RELEASE_LOCKOWNER   = 39,
///   OP_ILLEGAL             = 10044
/// };
///
/// union nfs_argop4 switch (nfs_opnum4 argop) {
///   case OP_ACCESS:        ACCESS4args opaccess;
///   case OP_CLOSE:         CLOSE4args opclose;
///   case OP_COMMIT:        COMMIT4args opcommit;
///   case OP_CREATE:        CREATE4args opcreate;
///   case OP_DELEGPURGE:    DELEGPURGE4args opdelegpurge;
///   case OP_DELEGRETURN:   DELEGRETURN4args opdelegreturn;
///   case OP_GETATTR:       GETATTR4args opgetattr;
///   case OP_GETFH:         void;
///   case OP_LINK:          LINK4args oplink;
///   case OP_LOCK:          LOCK4args oplock;
```

```
///   case OP_LOCKT:        LOCKT4args oplockt;
///   case OP_LOCKU:        LOCKU4args oplocku;
///   case OP_LOOKUP:       LOOKUP4args oplookup;
///   case OP_LOOKUPP:      void;
///   case OP_NVERIFY:      NVERIFY4args opnverify;
///   case OP_OPEN:         OPEN4args opopen;
///   case OP_OPENATTR:     OPENATTR4args opopenattr;
///   case OP_OPEN_CONFIRM: OPEN_CONFIRM4args opopen_confirm;
///   case OP_OPEN_DOWNGRADE:
///                         OPEN_DOWNGRADE4args opopen_downgrade;
///   case OP_PUTFH:        PUTFH4args opputfh;
///   case OP_PUTPUBFH:     void;
///   case OP_PUTROOTFH:    void;
///   case OP_READ:         READ4args opread;
///   case OP_READDIR:      READDIR4args opreaddir;
///   case OP_READLINK:     void;
///   case OP_REMOVE:       REMOVE4args opremove;
///   case OP_RENAME:       RENAME4args oprename;
///   case OP_RENEW:        RENEW4args oprenew;
///   case OP_RESTOREFH:    void;
///   case OP_SAVEFH:       void;
///   case OP_SECINFO:      SECINFO4args opsecinfo;
///   case OP_SETATTR:      SETATTR4args opsetattr;
///   case OP_SETCLIENTID:  SETCLIENTID4args opsetclientid;
///   case OP_SETCLIENTID_CONFIRM: SETCLIENTID_CONFIRM4args
///                         opsetclientid_confirm;
///   case OP_VERIFY:       VERIFY4args opverify;
///   case OP_WRITE:        WRITE4args opwrite;
///   case OP_RELEASE_LOCKOWNER:
///                         RELEASE_LOCKOWNER4args
///                         oprelease_lockowner;
///   case OP_ILLEGAL:      void;
/// };
///
/// union nfs_resop4 switch (nfs_opnum4 resop) {
///   case OP_ACCESS:       ACCESS4res opaccess;
///   case OP_CLOSE:        CLOSE4res opclose;
///   case OP_COMMIT:       COMMIT4res opcommit;
///   case OP_CREATE:       CREATE4res opcreate;
///   case OP_DELEGPURGE:   DELEGPURGE4res opdelegpurge;
///   case OP_DELEGRETURN:  DELEGRETURN4res opdelegreturn;
///   case OP_GETATTR:      GETATTR4res opgetattr;
///   case OP_GETFH:        GETFH4res opgetfh;
///   case OP_LINK:         LINK4res oplink;
///   case OP_LOCK:         LOCK4res oplock;
///   case OP_LOCKT:        LOCKT4res oplockt;
///   case OP_LOCKU:        LOCKU4res oplocku;
///   case OP_LOOKUP:       LOOKUP4res oplookup;
```

```
///   case OP_LOOKUPP:        LOOKUPP4res oplookupp;
///   case OP_NVERIFY:        NVERIFY4res opnverify;
///   case OP_OPEN:           OPEN4res opopen;
///   case OP_OPENATTR:       OPENATTR4res opopenattr;
///   case OP_OPEN_CONFIRM:   OPEN_CONFIRM4res opopen_confirm;
///   case OP_OPEN_DOWNGRADE:
///                           OPEN_DOWNGRADE4res
///                                     opopen_downgrade;
///   case OP_PUTFH:          PUTFH4res opputfh;
///   case OP_PUTPUBFH:       PUTPUBFH4res opputpubfh;
///   case OP_PUTROOTFH:      PUTROOTFH4res opputrootfh;
///   case OP_READ:           READ4res opread;
///   case OP_READDIR:        READDIR4res opreaddir;
///   case OP_READLINK:       READLINK4res opreadlink;
///   case OP_REMOVE:         REMOVE4res opremove;
///   case OP_RENAME:         RENAME4res oprename;
///   case OP_RENEW:          RENEW4res oprenew;
///   case OP_RESTOREFH:      RESTOREFH4res oprestorefh;
///   case OP_SAVEFH:         SAVEFH4res opsavefh;
///   case OP_SECINFO:        SECINFO4res opsecinfo;
///   case OP_SETATTR:        SETATTR4res opsetattr;
///   case OP_SETCLIENTID:    SETCLIENTID4res opsetclientid;
///   case OP_SETCLIENTID_CONFIRM:
///                           SETCLIENTID_CONFIRM4res
///                                     opsetclientid_confirm;
///   case OP_VERIFY:         VERIFY4res opverify;
///   case OP_WRITE:          WRITE4res opwrite;
///   case OP_RELEASE_LOCKOWNER:
///                           RELEASE_LOCKOWNER4res
///                                     oprelease_lockowner;
///   case OP_ILLEGAL:        ILLEGAL4res opillegal;
/// };
///
/// struct COMPOUND4args {
///         utf8str_cs      tag;
///         uint32_t        minorversion;
///         nfs_argop4      argarray<>;
/// };
///
/// struct COMPOUND4res {
///         nfsstat4        status;
///         utf8str_cs      tag;
///         nfs_resop4      resarray<>;
/// };
///
///
/// /*
///  * Remote file service routines
```

```
///   */
/// program NFS4_PROGRAM {
///         version NFS_V4 {
///                 void
///                         NFSPROC4_NULL(void) = 0;
///
///                 COMPOUND4res
///                         NFSPROC4_COMPOUND(COMPOUND4args) = 1;
///
///         } = 4;
/// } = 100003;
///
/// /*
///  * NFS4 Callback Procedure Definitions and Program
///  */
/// struct CB_GETATTR4args {
///         nfs_fh4 fh;
///         bitmap4 attr_request;
/// };
///
/// struct CB_GETATTR4resok {
///         fattr4  obj_attributes;
/// };
///
/// union CB_GETATTR4res switch (nfsstat4 status) {
///  case NFS4_OK:
///         CB_GETATTR4resok        resok4;
///  default:
///         void;
/// };
///
/// struct CB_RECALL4args {
///         stateid4        stateid;
///         bool            truncate;
///         nfs_fh4         fh;
/// };
///
/// struct CB_RECALL4res {
///         nfsstat4        status;
/// };
///
/// /*
///  * CB_ILLEGAL: Response for illegal operation numbers
///  */
/// struct CB_ILLEGAL4res {
///         nfsstat4        status;
/// };
///
```

```
/// /*
///  * Various definitions for CB_COMPOUND
///  */
/// %
/// enum nfs_cb_opnum4 {
///         OP_CB_GETATTR        = 3,
///         OP_CB_RECALL         = 4,
///         OP_CB_ILLEGAL        = 10044
/// };
///
/// union nfs_cb_argop4 switch (unsigned argop) {
///  case OP_CB_GETATTR:
///       CB_GETATTR4args         opcbgetattr;
///  case OP_CB_RECALL:
///       CB_RECALL4args          opcbrecall;
///  case OP_CB_ILLEGAL:          void;
/// };
///
/// union nfs_cb_resop4 switch (unsigned resop) {
///  case OP_CB_GETATTR:   CB_GETATTR4res   opcbgetattr;
///  case OP_CB_RECALL:    CB_RECALL4res    opcbrecall;
///  case OP_CB_ILLEGAL:   CB_ILLEGAL4res   opcbillegal;
/// };
///
///
/// struct CB_COMPOUND4args {
///         utf8str_cs      tag;
///         uint32_t        minorversion;
///         uint32_t        callback_ident;
///         nfs_cb_argop4   argarray<>;
/// };
///
/// struct CB_COMPOUND4res {
///         nfsstat4        status;
///         utf8str_cs      tag;
///         nfs_cb_resop4   resarray<>;
/// };
///
///
///
/// /*
///  * Program number is in the transient range since the client
///  * will assign the exact transient program number and provide
///  * that to the server via the SETCLIENTID operation.
///  */
/// program NFS4_CALLBACK {
///         version NFS_CB {
///                 void
```

```
///                          CB_NULL(void) = 0;
///                 CB_COMPOUND4res
///                          CB_COMPOUND(CB_COMPOUND4args) = 1;
///          } = 1;
/// } = 0x40000000;
```

2. Security Considerations

    See the Security Considerations section of [RFCNFSv4].

3. IANA Considerations

    This document does not have any IANA considerations.

4. Normative References

    [RFC4506] Eisler, M., "XDR: External Data Representation Standard",
               STD 67, RFC 4506, May 2006.

    [RFCNFSv4]
               Haynes, T. and D. Noveck, "NFS Version 4 Protocol", draft-
               ietf-nfsv4-rfc3530bis-35 (work in progress), Dec 2014.

Appendix A. Acknowledgments

    Tom Haynes would like to thank NetApp, Inc. for its funding of his
    time on this project.

    David Quigley tested the extraction of the .x file from this document
    and corrected the two resulting errors.

Appendix B. RFC Editor Notes

    [RFC Editor: please remove this section prior to publishing this
    document as an RFC]

    [RFC3530bis should be replaced by the RFC number of draft-ietf-
    nfsv4-rfc3530bis in this draft.]

    [RFC Editor: Please note that there is also a reference entry that
    needs to be modified for the companion document.]

Authors' Addresses

Thomas Haynes (editor)
Primary Data, Inc.
4300 El Camino Real Ste 100
Los Altos, CA 94022
USA

Phone: +1 408 215 1519
Email: thomas.haynes@primarydata.com

David Noveck (editor)
Dell
300 Innovative Way
Nashua, NH 03062
US

Phone: +1 781 572 8038
Email: dave_noveck@dell.com

NFSv4   M. Eisler
Internet-Draft   D. Kenchammana
Intended status: Standards Track   J. Lentini
Expires: August 27, 2011   M. Shankararao
                                                                NetApp
                                                               R. Iyer
                                                     February 23, 2011

                     NFS space reservation operations
                draft-iyer-nfsv4-space-reservation-ops-02.txt

   Abstract

      This document describes a set of NFS attributes and operations that
      are useful for applications like hypervisors to manage storage in a
      better manner.

described in the Simplified BSD License.

Table of Contents

1. Requirements notation

   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 [RFC2119].

2. Introduction

   This document describes a set of operations that allow applications
   such as hypervisors to reserve space for a file, report the amount of
   actual disk space a file occupies and freeup the backing space of a
   file when it is not required.

   In virtualized environments, virtual disk files are often stored on
   NFS mounted volumes. Since virtual disk files represent the hard
   disks of virtual machines, hypervisors often have to guarantee
   certain properties for the file.

   One such example is space reservation. When a hypervisor creates a
   virtual disk file, it often tries to preallocate the space for the
   file so that there are no future allocation related errors during the
   operation of the virtual machine. Such errors prevent a virtual
   machine from continuing execution and result in downtime.

   Another useful feature would be the ability to report the number of
   blocks that would be freed when a file is deleted. Currently, NFS
   reports two size attributes:

   o size - The logical file size of the file.

   o space_used - The size in bytes that the file occupies on disk

   While these attributes are sufficient for space accounting in
   traditional filesystems, they prove to be inadequate in modern
   filesystems that support block sharing. Having a way to tell the
   number of blocks that would be freed if the file was deleted would be
   useful to applications that wish to migrate files when a volume is
   low on space.

   Since virtual disks represent a hard drive in a virtual machine, a
   virtual disk can be viewed as a filesystem within a file. Since not
   all blocks within a filesystem are in use, there is an opportunity to
   reclaim blocks that are no longer in use. A call to deallocate
   blocks could result in better space efficiency. Lesser space MAY be
   consumed for backups after block deallocation.

   We propose the following operations and attributes for the

    aforementioned use cases:

    space_reserve: This attribute specifies whether the blocks backing
       the file have been preallocated.

    space_freed: This attribute specifies the space freed when a file is
       deleted, taking block sharing into consideration.

    HOLE_PUNCH: This operation zeroes and/or deallocates the blocks
       backing a region of the file.

    max_hole_punch: This attribute specifies the maximum sized hole that
       can be punched on the filesystem.

3. Use Cases

3.1. Space Reservation

    Some applications require that once a file of a certain size is
    created, writes to that file never fail with an out of space
    condition. One such example is that of a hypervisor writing to a
    virtual disk. An out of space condition while writing to virtual
    disks would mean that the virtual machine would need to be frozen.

    Currently, in order to achieve such a guarantee, applications zero
    the entire file. The initial zeroing allocates the backing blocks
    and all subsequent writes are overwrites of already allocated blocks.
    This approach is not only inefficient in terms of the amount of I/O
    done, it is also not guaranteed to work on filesystems that are log
    structured or deduplicated. An efficient way of guaranteeing space
    reservation would be beneficial to such applications.

    If the space_reserved attribute is set on a file, it is guaranteed
    that writes that do not grow the file will not fail with
    NFSERR_NOSPC.

3.2. Space freed on deletes

    Currently, files in NFS have two size attributes:

    o size - The logical file size of the file.

    o space_used - The size in bytes that the file occupies on disk.

    While these attributes are sufficient for space accounting in
    traditional filesystems, they prove to be inadequate in modern
    filesystems that support block sharing. In such filesystems,

multiple inodes can point to a single block with a block reference
count to guard against premature freeing.

If space_used of a file is interpreted to mean the size in bytes of
all disk blocks pointed to by the inode of the file, then shared
blocks get double counted, over-reporting the space utilization.
This also has the adverse effect that the deletion of a file with
shared blocks frees up less than space_used bytes.

On the other hand, if space_used is interpreted to mean the size in
bytes of those disk blocks unique to the inode of the file, then
shared blocks are not counted in any file, resulting in under-
reporting of the space utilization.

For example, two files A and B have 10 blocks each. Let 6 of these
blocks be shared between them. Thus, the combined space utilized by
the two files is 14 * BLOCK_SIZE bytes. In the former case, the
combined space utilization of the two files would be reported as 20 *
BLOCK_SIZE. However, deleting either would only result in 4 *
BLOCK_SIZE being freed. Conversely, the latter interpretation would
report that the space utilization is only 8 * BLOCK_SIZE.

Adding another size attribute, space_freed, is helpful in solving
this problem. space_freed is the number of blocks that are allocated
to the given file that would be freed on its deletion. In the
example, both A and B would report space_freed as 4 * BLOCK_SIZE and
space_used as 10 * BLOCK_SIZE. If A is deleted, B will report
space_freed as 10 * BLOCK_SIZE as the deletion of B would result in
the deallocation of all 10 blocks.

The addition of this problem doesn't solve the problem of space being
over-reported. However, over-reporting is better than under-
reporting.

4. Operations and attributes

   In the sections that follow, one operation and three attributes are
   defined that together provide the space management facilities
   outlined earlier in the document. The operation is intended to be
   OPTIONAL and the attributes RECOMMENDED as defined in section 17 of
   [RFC5661].

4.1. Attribute X: space_reserve

   The space_reserve attribute is a read/write attribute of type
   boolean. It is a per file attribute. When the space_reserved
   attribute is set via SETATTR, the server must ensure that there is

disk space to accommodate every byte in the file before it can return success. If the server cannot guarantee this, it must return NFS4ERR_NOSPC.

If the client tries to grow a file which has the space_reserved attribute set, the server must guarantee that there is disk space to accommodate every byte in the file with the new size before it can return success. If the server cannot guarantee this, it must return NFS4ERR_NOSPC.

It is not required that the server allocate the space to the file before returning success. The allocation can be deferred, however, it must be guaranteed that it will not fail for lack of space.

The value of space_reserved can be obtained at any time through GETATTR.

In order to avoid ambiguity, the space_reserve bit cannot be set along with the size bit in SETATTR. Increasing the size of a file with space_reserve set will fail if space reservation cannot be guaranteed for the new size. If the file size is decreased, space reservation is only guaranteed for the new size and the extra blocks backing the file can be released.

4.2. Attribute Y: space_freed

space_freed gives the number of bytes freed if the file is deleted. This attribute is read only and is of type length4. It is a per file attribute.

4.3. Attribute Z: max_hole_punch

max_hole_punch specifies the maximum size of a hole that the HOLE_PUNCH operation can handle. This attribute is read only and of type length4. It is a per filesystem attribute. This attribute MUST be implemented if HOLE_PUNCH is implemented.

4.4. Operation A: HOLE_PUNCH - Zero and deallocate blocks backing the file in the specified range.

ARGUMENTS

```
struct HOLE_PUNCH4args {
    /* CURRENT_FH: file */
    offset4         hpa_offset;
    length4         hpa_count;
};
```

RESULTS

```
struct HOLEPUNCH4res {
    nfsstat4        hpr_status;
};
```

DESCRIPTION

Whenever a client wishes to deallocate the blocks backing a particular region in the file, it calls the HOLE_PUNCH operation with the current filehandle set to the filehandle of the file in question, start offset and length in bytes of the region set in hpa_offset and hpa_count respectively. All further reads to this region MUST return zeros until overwritten. The filehandle specified must be that of a regular file.

Situations may arise where hpa_offset and/or hpa_offset + hpa_count will not be aligned to a boundary that the server does allocations/ deallocations in. For most filesystems, this is the block size of the file system. In such a case, the server can deallocate as many bytes as it can in the region. The blocks that cannot be deallocated MUST be zeroed. Except for the block deallocation and maximum hole punching capability, a HOLE_PUNCH operation is to be treated similar to a write of zeroes.

The server is not required to complete deallocating the blocks specified in the operation before returning. It is acceptable to have the deallocation be deferred. In fact, HOLE_PUNCH is merely a hint; it is valid for a server to return success without ever doing anything towards deallocating the blocks backing the region specified. However, any future reads to the region MUST return zeroes.

HOLE_PUNCH will result in the space_used attribute being decreased by the number of bytes that were deallocated. The space_freed attribute may or may not decrease, depending on the support and whether the blocks backing the specified range were shared or not. The size attribute will remain unchanged.

The HOLE_PUNCH operation MUST NOT change the space reservation guarantee of the file. While the server can deallocate the blocks specified by hpa_offset and hpa_count, future writes to this region MUST NOT fail with NFSERR_NOSPC.

The HOLE_PUNCH operation may fail for the following reasons (this is a partial list):

NFS4ERR_NOTSUPP: The Hole punch operations is not supported by the
   NFS server receiving this request.

NFS4ERR_DIR: The current filehandle is of type NF4DIR.

NFS4ERR_SYMLINK: The current filehandle is of type NF4LNK.

NFS4ERR_WRONG_TYPE: The current filehandle does not designate an
   ordinary file.

5. Security Considerations

   There are no security considerations.

6. IANA Considerations

   This document has no actions for IANA.

7. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC5661] Shepler, S., Eisler, M., and D. Noveck, "Network File
              System (NFS) Version 4 Minor Version 1 Protocol",
              RFC 5661, January 2010.

Authors' Addresses

   Mike Eisler
   NetApp
   5765 Chase Point Circle
   Colorado Springs, CO 80919
   USA

   Phone: +1 719 599 9026
   Email: mike@eisler.com
   URI: http://www.eisler.com

Deepak Kenchammana
NetApp
475 East Java Drive
Sunnyvale, CA 94089
USA

Phone: +1 408 822 4765
Email: kencham@netapp.com

James Lentini
NetApp
1601 Trapelo Rd, Suite 16
Waltham, MA 02451
USA

Phone: +1 781 768 5359
Email: jlentini@netapp.com

Manjunath Shankararao
NetApp
3rd Floor, Fair Winds Block, EGL Software Park
Bangalore, Karnataka 560085
INDIA

Phone: +91 80 4184 3397
Email: rudra@netapp.com

Rahul Iyer
655 S Fair Oaks Ave Apt #I-314
Sunnyvale, CA 94086
USA

Email: rahulair@yahoo.com

NFSv4   J. Lentini
Internet-Draft   M. Eisler
Intended status: Standards Track   D. Kenchammana
Expires: April 24, 2011   NetApp
                                                               A. Madan
                                             Carnegie Mellon University
                                                                R. Iyer
                                                       October 21, 2010

NFS Server-side Copy
draft-lentini-nfsv4-server-side-copy-06.txt

Abstract

   This document describes a set of NFS operations for offloading a file
   copy to a file server or between two file servers.

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 24, 2011.

described in the Simplified BSD License.

Table of Contents

1. Requirements notation

   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 [RFC2119].

2. Introduction

   This document describes a server-side copy feature for the NFS
   protocol.

   The server-side copy feature provides a mechanism for the NFS client
   to perform a file copy on the server without the data being
   transmitted back and forth over the network.

   Without this feature, an NFS client copies data from one location to
   another by reading the data from the server over the network, and
   then writing the data back over the network to the server. Using
   this server-side copy operation, the client is able to instruct the
   server to copy the data locally without the data being sent back and
   forth over the network unnecessarily.

   In general, this feature is useful whenever data is copied from one
   location to another on the server. It is particularly useful when
   copying the contents of a file from a backup. Backup-versions of a
   file are copied for a number of reasons, including restoring and
   cloning data.

   If the source object and destination object are on different file
   servers, the file servers will communicate with one another to
   perform the copy operation. The server-to-server protocol by which
   this is accomplished is not defined in this document.

3. Protocol Overview

   The server-side copy offload operations support both intra-server and
   inter-server file copies. An intra-server copy is a copy in which
   the source file and destination file reside on the same server. In
   an inter-server copy, the source file and destination file are on
   different servers. In both cases, the copy may be performed
   synchronously or asynchronously.

   Throughout the rest of this document, we refer to the NFS server
   containing the source file as the "source server" and the NFS server
   to which the file is transferred as the "destination server". In the
   case of an intra-server copy, the source server and destination

server are the same server. Therefore in the context of an intra-
server copy, the terms source server and destination server refer to
the single server performing the copy.

The operations described below are designed to copy files. Other
file system objects can be copied by building on these operations or
using other techniques. For example if the user wishes to copy a
directory, the client can synthesize a directory copy by first
creating the destination directory and then copying the source
directory's files to the new destination directory. If the user
wishes to copy a namespace junction [FEDFS-NSDB] [FEDFS-ADMIN], the
client can use the ONC RPC Federated Filesystem protocol
[FEDFS-ADMIN] to perform the copy. Specifically the client can
determine the source junction's attributes using the FEDFS_LOOKUP_FSN
procedure and create a duplicate junction using the
FEDFS_CREATE_JUNCTION procedure.

For the inter-server copy protocol, the operations are defined to be
compatible with a server-to-server copy protocol in which the
destination server reads the file data from the source server. This
model in which the file data is pulled from the source by the
destination has a number of advantages over a model in which the
source pushes the file data to the destination. The advantages of
the pull model include:

o The pull model only requires a remote server (i.e. the destination
   server) to be granted read access. A push model requires a remote
   server (i.e. the source server) to be granted write access, which
   is more privileged.

o The pull model allows the destination server to stop reading if it
   has run out of space. In a push model, the destination server
   must flow control the source server in this situation.

o The pull model allows the destination server to easily flow
   control the data stream by adjusting the size of its read
   operations. In a push model, the destination server does not have
   this ability. The source server in a push model is capable of
   writing chunks larger than the destination server has requested in
   attributes and session parameters. In theory, the destination
   server could perform a "short" write in this situation, but this
   approach is known to behave poorly in practice.

The following operations are provided to support server-side copy:

COPY_NOTIFY: For inter-server copies, the client sends this
   operation to the source server to notify it of a future file copy
   from a given destination server for the given user.

COPY_REVOKE: Also for inter-server copies, the client sends this
   operation to the source server to revoke permission to copy a file
   for the given user.

COPY: Used by the client to request a file copy.

COPY_ABORT: Used by the client to abort an asynchronous file copy.

COPY_STATUS: Used by the client to poll the status of an
   asynchronous file copy.

CB_COPY: Used by the destination server to report the results of an
   asynchronous file copy to the client.

These operations are described in detail in Section 4. This section
provides an overview of how these operations are used to perform
server-side copies.

3.1. Intra-Server Copy

To copy a file on a single server, the client uses a COPY operation.
The server may respond to the copy operation with the final results
of the copy or it may perform the copy asynchronously and deliver the
results using a CB_COPY operation callback. If the copy is performed
asynchronously, the client may poll the status of the copy using
COPY_STATUS or cancel the copy using COPY_ABORT.

A synchronous intra-server copy is shown in Figure 1. In this
example, the NFS server chooses to perform the copy synchronously.
The copy operation is completed, either successfully or
unsuccessfully, before the server replies to the client's request.
The server's reply contains the final result of the operation.

```
   Client                                      Server
     +                                           +
     |                                           |
     |--- COPY ----------------------------->| Client requests
     |<--------------------------------/| a file copy
     |                                           |
     |                                           |
```

                 Figure 1: A synchronous intra-server copy.

An asynchronous intra-server copy is shown in Figure 2. In this

example, the NFS server performs the copy asynchronously. The
server's reply to the copy request indicates that the copy operation
was initiated and the final result will be delivered at a later time.
The server's reply also contains a copy stateid. The client may use
this copy stateid to poll for status information (as shown) or to
cancel the copy using a COPY_ABORT. When the server completes the
copy, the server performs a callback to the client and reports the
results.

```
     Client                                        Server
       +                                             +
       |                                             |
       |--- COPY ---------------------------->| Client requests
       |<------------------------------------/| a file copy
       |                                             |
       |                                             |
       |--- COPY_STATUS -------------------->| Client may poll
       |<------------------------------------/| for status
       |                                             |
       |                 .                           | Multiple COPY_STATUS
       |                 .                           | operations may be sent.
       |                 .                           |
       |                                             |
       |<-- CB_COPY ------------------------| Server reports results
       |\----------------------------------->|
       |                                             |
```
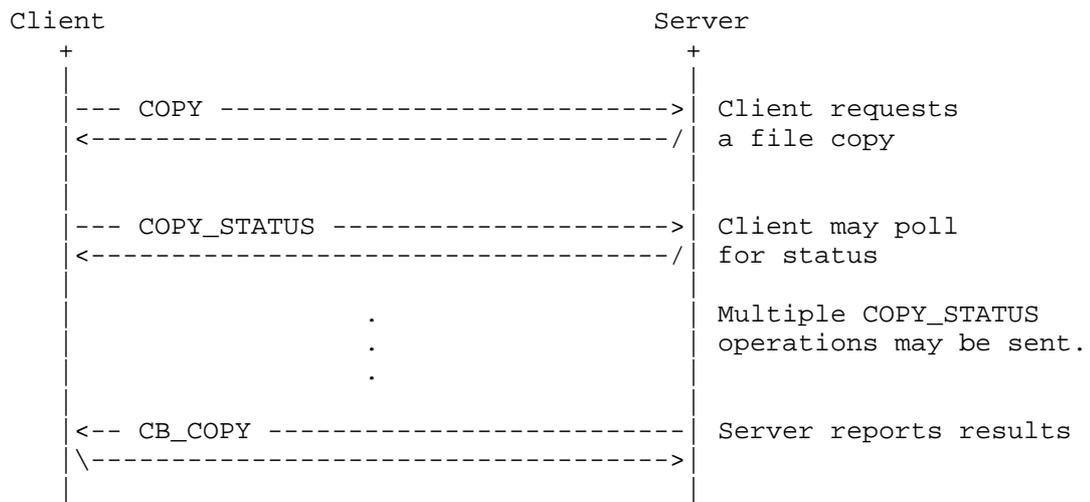
                Figure 2: An asynchronous intra-server copy.

3.2. Inter-Server Copy

   A copy may also be performed between two servers. The copy protocol
   is designed to accommodate a variety of network topologies. As shown
   in Figure 3, the client and servers may be connected by multiple
   networks. In particular, the servers may be connected by a
   specialized, high speed network (network 192.168.33.0/24 in the
   diagram) that does not include the client. The protocol allows the
   client to setup the copy between the servers (over network
   10.11.78.0/24 in the diagram) and for the servers to communicate on
   the high speed network if they choose to do so.

```
                         192.168.33.0/24
            +----------------------------------+
            |                                  |
            |                                  |
            | 192.168.33.18                    | 192.168.33.56
       +-------+------+                  +------+------+
       |    Source    |                  | Destination |
       +-------+------+                  +------+------+
            | 10.11.78.18                      | 10.11.78.56
            |                                  |
            |                                  |
            |          10.11.78.0/24           |
            +----------------+-----------------+
                             |
                             |
                             | 10.11.78.243
                       +-----+-----+
                       |  Client   |
                       +-----------+
```
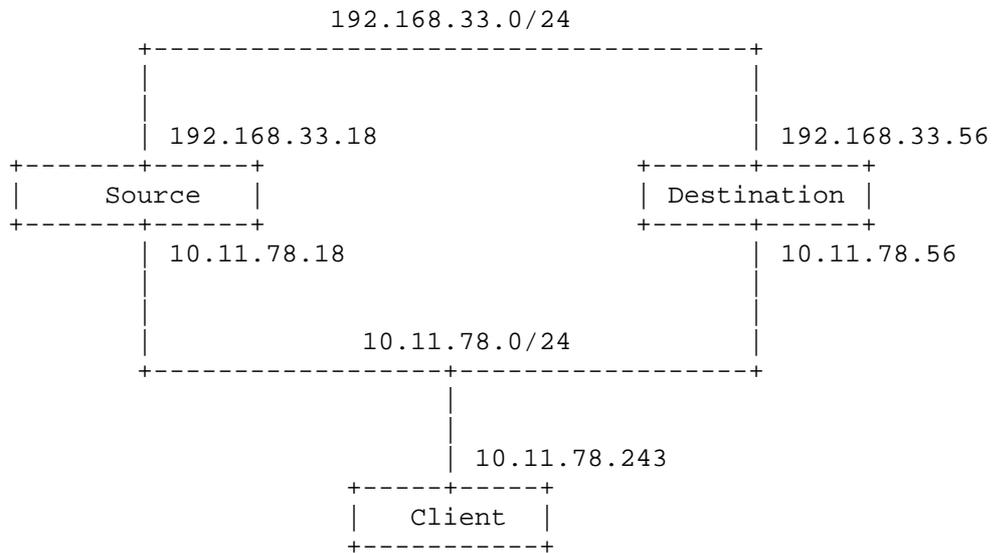
Figure 3: An example inter-server network topology.

For an inter-server copy, the client notifies the source server that
a file will be copied by the destination server using a COPY_NOTIFY
operation. The client then initiates the copy by sending the COPY
operation to the destination server. The destination server may
perform the copy synchronously or asynchronously.

A synchronous inter-server copy is shown in Figure 4. In this case,
the destination server chooses to perform the copy before responding
to the client's COPY request.

An asynchronous copy is shown in Figure 5. In this case, the
destination server chooses to respond to the client's COPY request
immediately and then perform the copy asynchronously.

```
      Client                Source          Destination
        +                     +                  +
        |                     |                  |
        |--- COPY_NOTIFY --->|                  |
        |<----------------/|                  |
        |                     |                  |
        |                     |                  |
        |--- COPY -------------------------->|
        |                     |                  |
        |                     |                  |
        |                     |<----- read -----|
        |                     |\------------->|
        |                     |                  |
        |                     |        .         | Multiple reads may
        |                     |        .         | be necessary
        |                     |        .         |
        |                     |                  |
        |                     |                  |
        |<--------------------------------/| Destination replies
        |                     |                  | to COPY
        |                     |                  |
```
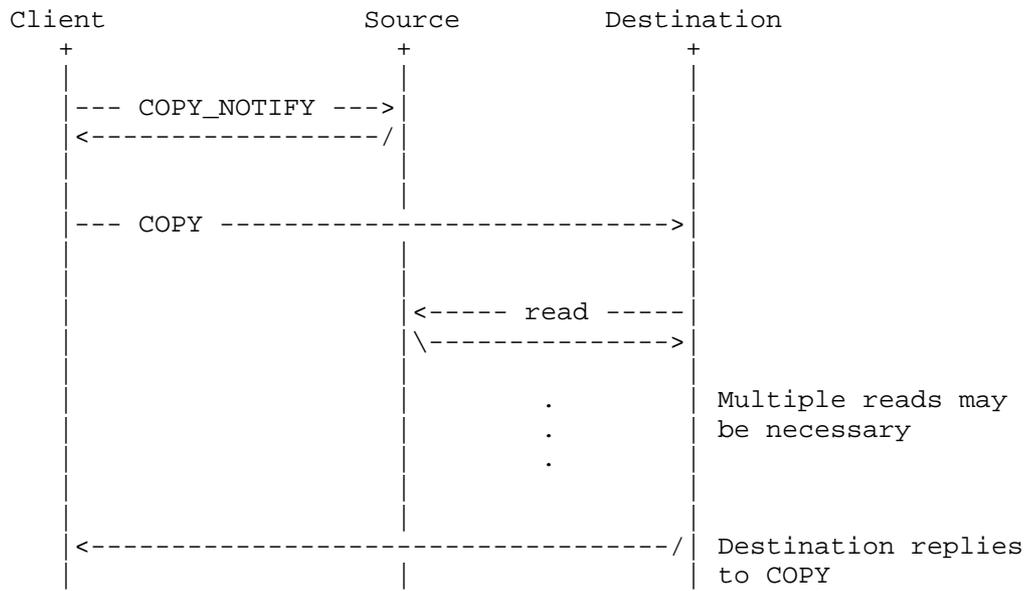
                Figure 4: A synchronous inter-server copy.

```
        Client              Source         Destination
         +                   +                 +
         |                   |                 |
         |--- COPY_NOTIFY --->|                 |
         |<------------------/|                 |
         |                   |                 |
         |                   |                 |
         |--- COPY --------------------------->|
         |<----------------------------------/|
         |                   |                 |
         |                   |                 |
         |                   |<----- read -----|
         |                   |\-------------->|
         |                   |                 |
         |                   |        .        | Multiple reads may
         |                   |        .        | be necessary
         |                   |        .        |
         |                   |                 |
         |                   |                 |
         |--- COPY_STATUS -------------------->| Client may poll
         |<----------------------------------/| for status
         |                   |                 |
         |                   |        .        | Multiple COPY_STATUS
         |                   |        .        | operations may be sent
         |                   |        .        |
         |                   |                 |
         |                   |                 |
         |<-- CB_COPY ------------------------| Destination reports
         |\---------------------------------->| results
         |                   |                 |
```
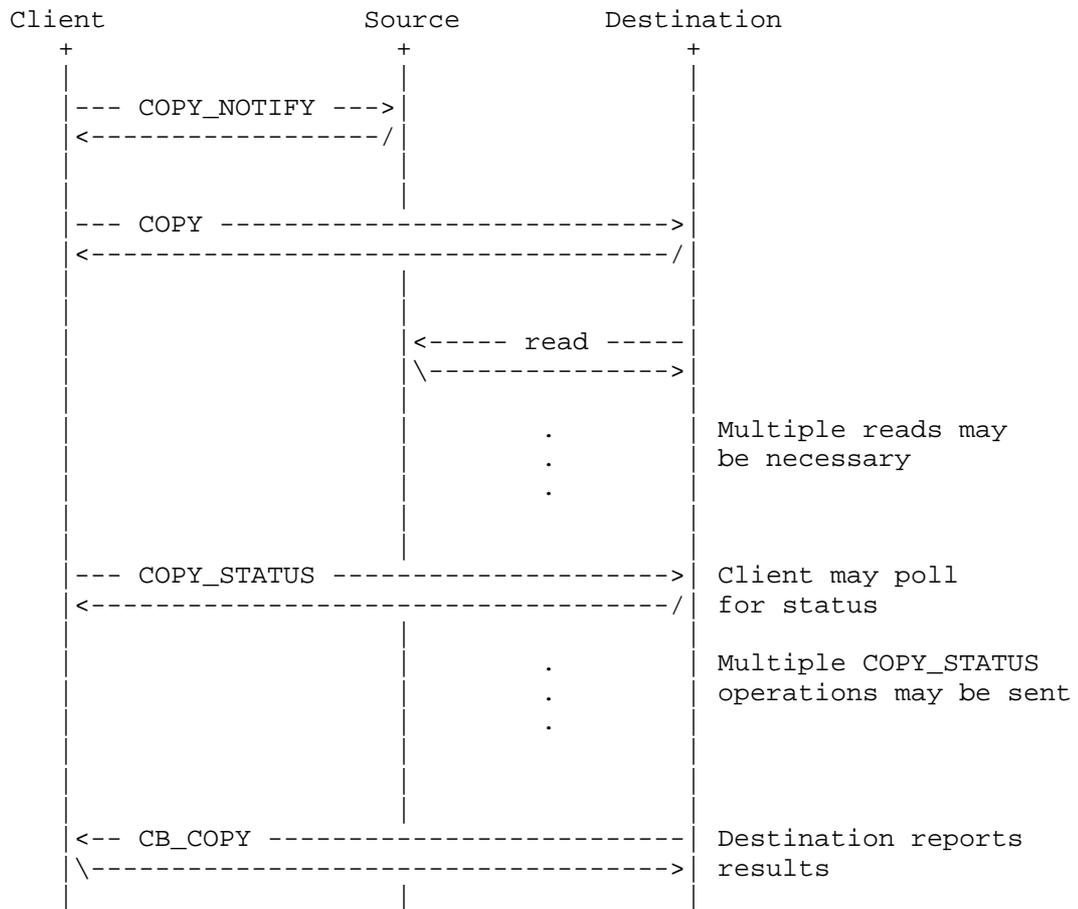
               Figure 5: An asynchronous inter-server copy.

3.3. Server-to-Server Copy Protocol

   During an inter-server copy, the destination server reads the file
   data from the source server. The source server and destination
   server are not required to use a specific protocol to transfer the
   file data. The choice of what protocol to use is ultimately the
   destination server's decision.

3.3.1. Using NFSv4.x as a Server-to-Server Copy Protocol

   The destination server MAY use standard NFSv4.x (where x >= 1) to
   read the data from the source server. If NFSv4.x is used for the
   server-to-server copy protocol, the destination server can use the
   filehandle contained in the COPY request with standard NFSv4.x

operations to read data from the source server. Specifically, the
destination server may use the NFSv4.x OPEN operation's CLAIM_FH
facility to open the file being copied and obtain an open stateid.
Using the stateid, the destination server may then use NFSv4.x READ
operations to read the file.

3.3.2. Using an alternative Server-to-Server Copy Protocol

In a homogeneous environment, the source and destination servers
might be able to perform the file copy extremely efficiently using
specialized protocols. For example the source and destination
servers might be two nodes sharing a common file system format for
the source and destination file systems. Thus the source and
destination are in an ideal position to efficiently render the image
of the source file to the destination file by replicating the file
system formats at the block level. Another possibility is that the
source and destination might be two nodes sharing a common storage
area network, and thus there is no need to copy any data at all, and
instead ownership of the file and its contents might simply be re-
assigned to the destination. To allow for these possibilities, the
destination server is allowed to use a server-to-server copy protocol
of its choice.

In a heterogeneous environment, using a protocol other than NFSv4.x
(e.g. HTTP [RFC2616] or FTP [RFC0959]) presents some challenges. In
particular, the destination server is presented with the challenge of
accessing the source file given only an NFSv4.x filehandle.

One option for protocols that identify source files with path names
is to use an ASCII hexadecimal representation of the source
filehandle as the file name.

Another option for the source server is to use URLs to direct the
destination server to a specialized service. For example, the
response to COPY_NOTIFY could include the URL
ftp://s1.example.com:9999/_FH/0x12345, where 0x12345 is the ASCII
hexadecimal representation of the source filehandle. When the
destination server receives the source server's URL, it would use
"_FH/0x12345" as the file name to pass to the FTP server listening on
port 9999 of s1.example.com. On port 9999 there would be a special
instance of the FTP service that understands how to convert NFS
filehandles to an open file descriptor (in many operating systems,
this would require a new system call, one which is the inverse of the
makefh() function that the pre-NFSv4 MOUNT service needs).

Authenticating and identifying the destination server to the source
server is also a challenge. Recommendations for how to accomplish
this are given in Section 5.1.2.4 and Section 5.1.4.

4. Operations

   In the sections that follow, several operations are defined that
   together provide the server-side copy feature. These operations are
   intended to be OPTIONAL operations as defined in section 17 of
   [RFC5661]. The COPY_NOTIFY, COPY_REVOKE, COPY, COPY_ABORT, and
   COPY_STATUS operations are designed to be sent within an NFSv4
   COMPOUND procedure. The CB_COPY operation is designed to be sent
   within an NFSv4 CB_COMPOUND procedure.

   Each operation is performed in the context of the user identified by
   the ONC RPC credential of its containing COMPOUND or CB_COMPOUND
   request. For example, a COPY_ABORT operation issued by a given user
   indicates that a specified COPY operation initiated by the same user
   be canceled. Therefore a COPY_ABORT MUST NOT interfere with a copy
   of the same file initiated by another user.

   An NFS server MAY allow an administrative user to monitor or cancel
   copy operations using an implementation specific interface.

4.1. netloc4 - Network Locations

   The server-side copy operations specify network locations using the
   netloc4 data type shown below:

```
                enum netloc_type4 {
                        NL4_NAME      = 0,
                        NL4_URL       = 1,
                        NL4_NETADDR   = 2
                };

                union netloc4 switch (netloc_type4 nl_type) {
                        case NL4_NAME:    utf8str_cis nl_name;
                        case NL4_URL:     utf8str_cis nl_url;
                        case NL4_NETADDR: netaddr4    nl_addr;
                };
```

   If the netloc4 is of type NL4_NAME, the nl_name field MUST be
   specified as a UTF-8 string. The nl_name is expected to be resolved
   to a network address via DNS, LDAP, NIS, /etc/hosts, or some other
   means. If the netloc4 is of type NL4_URL, a server URL [RFC3986]
   appropriate for the server-to-server copy operation is specified as a
   UTF-8 string. If the netloc4 is of type NL4_NETADDR, the nl_addr
   field MUST contain a valid netaddr4 as defined in Section 3.3.9 of
   [RFC5661].

   When netloc4 values are used for an inter-server copy as shown in
   Figure 3, their values may be evaluated on the source server,

destination server, and client. The network environment in which
these systems operate should be configured so that the netloc4 values
are interpreted as intended on each system.

4.2. Operation U: COPY_NOTIFY - Notify a source server of a future copy

   ARGUMENTS

```
                  struct COPY_NOTIFY4args {
                          /* CURRENT_FH: source file */
                          netloc4          cna_destination_server;
                  };
```

   RESULTS

```
                  union COPY_NOTIFY4res switch (nfsstat4 cnr_status) {
                  case NFS4_OK:
                          nfstime4         cnr_lease_time;
                          netloc4          cnr_source_server<>;
                  default:
                          void;
                  };
```

   DESCRIPTION

   This operation is used for an inter-server copy. A client sends this
   operation in a COMPOUND request to the source server to authorize a
   destination server identified by cna_destination_server to read the
   file specified by CURRENT_FH on behalf of the given user.

   The cna_destination_server MUST be specified using the netloc4
   network location format. The server is not required to resolve the
   cna_destination_server address before completing this operation.

   If this operation succeeds, the source server will allow the
   cna_destination_server to copy the specified file on behalf of the
   given user. If COPY_NOTIFY succeeds, the destination server is
   granted permission to read the file as long as both of the following
   conditions are met:

   o The destination server begins reading the source file before the
      cnr_lease_time expires. If the cnr_lease_time expires while the
      destination server is still reading the source file, the
      destination server is allowed to finish reading the file.

   o The client has not issued a COPY_REVOKE for the same combination
      of user, filehandle, and destination server.

The cnr_lease_time is chosen by the source server. A cnr_lease_time
of 0 (zero) indicates an infinite lease. To renew the copy lease
time the client should resend the same copy notification request to
the source server.

To avoid the need for synchronized clocks, copy lease times are
granted by the server as a time delta. However, there is a
requirement that the client and server clocks do not drift
excessively over the duration of the lease. There is also the issue
of propagation delay across the network which could easily be several
hundred milliseconds as well as the possibility that requests will be
lost and need to be retransmitted.

To take propagation delay into account, the client should subtract it
from copy lease times (e.g. if the client estimates the one-way
propagation delay as 200 milliseconds, then it can assume that the
lease is already 200 milliseconds old when it gets it). In addition,
it will take another 200 milliseconds to get a response back to the
server. So the client must send a lease renewal or send the copy
offload request to the cna_destination_server at least 400
milliseconds before the copy lease would expire. If the propagation
delay varies over the life of the lease (e.g. the client is on a
mobile host), the client will need to continuously subtract the
increase in propagation delay from the copy lease times.

The server's copy lease period configuration should take into account
the network distance of the clients that will be accessing the
server's resources. It is expected that the lease period will take
into account the network propagation delays and other network delay
factors for the client population. Since the protocol does not allow
for an automatic method to determine an appropriate copy lease
period, the server's administrator may have to tune the copy lease
period.

A successful response will also contain a list of names, addresses,
and URLs called cnr_source_server, on which the source is willing to
accept connections from the destination. These might not be
reachable from the client and might be located on networks to which
the client has no connection.

If the client wishes to perform an inter-server copy, the client MUST
send a COPY_NOTIFY to the source server. Therefore, the source
server MUST support COPY_NOTIFY.

For a copy only involving one server (the source and destination are
on the same server), this operation is unnecessary.

The COPY_NOTIFY operation may fail for the following reasons (this is

a partial list):

NFS4ERR_MOVED: The file system which contains the source file is not
   present on the source server. The client can determine the
   correct location and reissue the operation with the correct
   location.

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the
   NFS server receiving this request.

NFS4ERR_WRONGSEC: The security mechanism being used by the client
   does not match the server's security policy.

4.3. Operation V: COPY_REVOKE - Revoke a destination server's copy
      privileges

   ARGUMENTS

                 struct COPY_REVOKE4args {
                         /* CURRENT_FH: source file */
                         netloc4   cra_destination_server;
                 };

   RESULTS

                 struct COPY_REVOKE4res {
                         nfsstat4   crr_status;
                 };

   DESCRIPTION

   This operation is used for an inter-server copy. A client sends this
   operation in a COMPOUND request to the source server to revoke the
   authorization of a destination server identified by
   cra_destination_server from reading the file specified by CURRENT_FH
   on behalf of given user. If the cra_destination_server has already
   begun copying the file, a successful return from this operation
   indicates that further access will be prevented.

   The cra_destination_server MUST be specified using the netloc4
   network location format. The server is not required to resolve the
   cra_destination_server address before completing this operation.

   The COPY_REVOKE operation is useful in situations in which the source
   server granted a very long or infinite lease on the destination
   server's ability to read the source file and all copy operations on
   the source file have been completed.

For a copy only involving one server (the source and destination are on the same server), this operation is unnecessary.

If the server supports COPY_NOTIFY, the server is REQUIRED to support the COPY_REVOKE operation.

The COPY_REVOKE operation may fail for the following reasons (this is a partial list):

NFS4ERR_MOVED: The file system which contains the source file is not present on the source server. The client can determine the correct location and reissue the operation with the correct location.

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the NFS server receiving this request.

4.4. Operation W: COPY - Initiate a server-side copy

   ARGUMENTS

```
                #define COPY4_GUARDED       = 0x00000001;
                #define COPY4_METADATA      = 0x00000002;

                struct COPY4args {
                        /* SAVED_FH: source file */
                        /* CURRENT_FH: destination file or */
                        /*             directory          */
                        offset4         ca_src_offset;
                        offset4         ca_dst_offset;
                        length4         ca_count;
                        uint32_t        ca_flags;
                        component4      ca_destination;
                        netloc4         ca_source_server<>;
                };
```

   RESULTS

```
                union COPY4res switch (nfsstat4 cr_status) {
                        /* CURRENT_FH: destination file */
                case NFS4_OK:
                        stateid4        cr_callback_id<1>;
                default:
                        length4         cr_bytes_copied;
                };
```

   DESCRIPTION

The COPY operation is used for both intra- and inter-server copies.
In both cases, the COPY is always sent from the client to the
destination server of the file copy. The COPY operation requests
that a file be copied from the location specified by the SAVED_FH
value to the location specified by the combination of CURRENT_FH and
ca_destination.

The SAVED_FH must be a regular file. If SAVED_FH is not a regular
file, the operation MUST fail and return NFS4ERR_WRONG_TYPE.

In order to set SAVED_FH to the source file handle, the compound
procedure requesting the COPY will include a sub-sequence of
operations such as

                        PUTFH source-fh
                        SAVEFH

If the request is for a server-to-server copy, the source-fh is a
filehandle from the source server and the compound procedure is being
executed on the destination server. In this case, the source-fh is a
foreign filehandle on the server receiving the COPY request. If
either PUTFH or SAVEFH checked the validity of the filehandle, the
operation would likely fail and return NFS4ERR_STALE.

In order to avoid this problem, the minor version incorporating the
COPY operations will need to make a few small changes in the handling
of existing operations. If a server supports the server-to-server
COPY feature, a PUTFH followed by a SAVEFH MUST NOT return
NFS4ERR_STALE for either operation. These restrictions do not pose
substantial difficulties for servers. The CURRENT_FH and SAVED_FH
may be validated in the context of the operation referencing them and
an NFS4ERR_STALE error returned for an invalid file handle at that
point.

The CURRENT_FH and ca_destination together specify the destination of
the copy operation. If ca_destination is of 0 (zero) length, then
CURRENT_FH specifies the target file. In this case, CURRENT_FH MUST
be a regular file and not a directory. If ca_destination is not of 0
(zero) length, the ca_destination argument specifies the file name to
which the data will be copied within the directory identified by
CURRENT_FH. In this case, CURRENT_FH MUST be a directory and not a
regular file.

If the file named by ca_destination does not exist and the operation
completes successfully, the file will be visible in the file system
namespace. If the file does not exist and the operation fails, the
file MAY be visible in the file system namespace depending on when
the failure occurs and on the implementation of the NFS server

receiving the COPY operation. If the ca_destination name cannot be
created in the destination file system (due to file name
restrictions, such as case or length), the operation MUST fail.

The ca_src_offset is the offset within the source file from which the
data will be read, the ca_dst_offset is the offset within the
destination file to which the data will be written, and the ca_count
is the number of bytes that will be copied. An offset of 0 (zero)
specifies the start of the file. A count of 0 (zero) requests that
all bytes from ca_src_offset through EOF be copied to the
destination. If concurrent modifications to the source file overlap
with the source file region being copied, the data copied may include
all, some, or none of the modifications. The client can use standard
NFS operations (e.g. OPEN with OPEN4_SHARE_DENY_WRITE or mandatory
byte range locks) to protect against concurrent modifications if the
client is concerned about this. If the source file's end of file is
being modified in parallel with a copy that specifies a count of 0
(zero) bytes, the amount of data copied is implementation dependent
(clients may guard against this case by specifying a non-zero count
value or preventing modification of the source file as mentioned
above).

If the source offset or the source offset plus count is greater than
or equal to the size of the source file, the operation will fail with
NFS4ERR_INVAL. The destination offset or destination offset plus
count may be greater than the size of the destination file. This
allows for the client to issue parallel copies to implement
operations such as "cat file1 file2 file3 file4 > dest".

If the destination file is created as a result of this command, the
destination file's size will be equal to the number of bytes
successfully copied. If the destination file already existed, the
destination file's size may increase as a result of this operation
(e.g. if ca_dst_offset plus ca_count is greater than the
destination's initial size).

If the ca_source_server list is specified, then this is an inter-
server copy operation and the source file is on a remote server. The
client is expected to have previously issued a successful COPY_NOTIFY
request to the remote source server. The ca_source_server list
SHOULD be the same as the COPY_NOTIFY response's cnr_source_server
list. If the client includes the entries from the COPY_NOTIFY
response's cnr_source_server list in the ca_source_server list, the
source server can indicate a specific copy protocol for the
destination server to use by returning a URL, which specifies both a
protocol service and server name. Server-to-server copy protocol
considerations are described in Section 3.3 and Section 5.1.

The ca_flags argument allows the copy operation to be customized in the following ways using the guarded flag (COPY4_GUARDED) and the metadata flag (COPY4_METADATA).

[NOTE: Earlier versions of this document defined a COPY4_SPACE_RESERVED flag for controlling space reservations on the destination file. This flag has been removed with the expectation that the space_reserve attribute defined in [SPACE-RESERVE] will be adopted.]

If the guarded flag is set and the destination exists on the server, this operation will fail with NFS4ERR_EXIST.

If the guarded flag is not set and the destination exists on the server, the behavior is implementation dependent.

If the metadata flag is set and the client is requesting a whole file copy (i.e. ca_count is 0 (zero)), a subset of the destination file's attributes MUST be the same as the source file's corresponding attributes and a subset of the destination file's attributes SHOULD be the same as the source file's corresponding attributes. The attributes in the MUST and SHOULD copy subsets will be defined for each NFS version.

For NFSv4.1, Table 1 and Table 2 list the REQUIRED and RECOMMENDED attributes respectively. A "MUST" in the "Copy to destination file?" column indicates that the attribute is part of the MUST copy set. A "SHOULD" in the "Copy to destination file?" column indicates that the attribute is part of the SHOULD copy set.

```
         +-------------------+----+--------------------------+
         | Name              | Id | Copy to destination file? |
         +-------------------+----+--------------------------+
         | supported_attrs   | 0  | no                       |
         | type              | 1  | MUST                     |
         | fh_expire_type    | 2  | no                       |
         | change            | 3  | SHOULD                   |
         | size              | 4  | MUST                     |
         | link_support      | 5  | no                       |
         | symlink_support   | 6  | no                       |
         | named_attr        | 7  | no                       |
         | fsid              | 8  | no                       |
         | unique_handles    | 9  | no                       |
         | lease_time        | 10 | no                       |
         | rdattr_error      | 11 | no                       |
         | filehandle        | 19 | no                       |
         | suppattr_exclcreat| 75 | no                       |
         +-------------------+----+--------------------------+
```

                          Table 1

```
         +-------------------+----+--------------------------+
         | Name              | Id | Copy to destination file? |
         +-------------------+----+--------------------------+
         | acl               | 12 | MUST                     |
         | aclsupport        | 13 | no                       |
         | archive           | 14 | no                       |
         | cansettime        | 15 | no                       |
         | case_insensitive  | 16 | no                       |
         | case_preserving   | 17 | no                       |
         | change_policy     | 60 | no                       |
         | chown_restricted  | 18 | MUST                     |
         | dacl              | 58 | MUST                     |
         | dir_notif_delay   | 56 | no                       |
         | dirent_notif_delay| 57 | no                       |
         | fileid            | 20 | no                       |
         | files_avail       | 21 | no                       |
         | files_free        | 22 | no                       |
         | files_total       | 23 | no                       |
         | fs_charset_cap    | 76 | no                       |
         | fs_layout_type    | 62 | no                       |
         | fs_locations      | 24 | no                       |
         | fs_locations_info | 67 | no                       |
         | fs_status         | 61 | no                       |
         | hidden            | 25 | MUST                     |
         | homogeneous       | 26 | no                       |
         | layout_alignment  | 66 | no                       |
         | layout_blksize    | 65 | no                       |
```

layout_hint	63	no
layout_type	64	no
maxfilesize	27	no
maxlink	28	no
maxname	29	no
maxread	30	no
maxwrite	31	no
mdsthreshold	68	no
mimetype	32	MUST
mode	33	MUST
mode_set_masked	74	no
mounted_on_fileid	55	no
no_trunc	34	no
numlinks	35	no
owner	36	MUST
owner_group	37	MUST
quota_avail_hard	38	no
quota_avail_soft	39	no
quota_used	40	no
rawdev	41	no
retentevt_get	71	MUST
retentevt_set	72	no
retention_get	69	MUST
retention_hold	73	MUST
retention_set	70	no
sacl	59	MUST
space_avail	42	no
space_free	43	no
space_total	44	no
space_used	45	no
system	46	MUST
time_access	47	MUST
time_access_set	48	no
time_backup	49	no
time_create	50	MUST
time_delta	51	no
time_metadata	52	SHOULD
time_modify	53	MUST
time_modify_set	54	no

                             Table 2

   [NOTE: The space_reserve attribute [SPACE-RESERVE] will be in the
   MUST set.]

   [NOTE: The source file's attribute values will take precedence over
   any attribute values inherited by the destination file.]

In the case of an inter-server copy or an intra-server copy between
file systems, the attributes supported for the source file and
destination file could be different. By definition,the REQUIRED
attributes will be supported in all cases. If the metadata flag is
set and the source file has a RECOMMENDED attribute that is not
supported for the destination file, the copy MUST fail with
NFS4ERR_ATTRNOTSUPP.

Any attribute supported by the destination server that is not set on
the source file SHOULD be left unset.

Metadata attributes not exposed via the NFS protocol SHOULD be copied
to the destination file where appropriate.

The destination file's named attributes are not duplicated from the
source file. After the copy process completes, the client MAY
attempt to duplicate named attributes using standard NFSv4
operations. However, the destination file's named attribute
capabilities MAY be different from the source file's named attribute
capabilities.

If the metadata flag is not set and the client is requesting a whole
file copy (i.e. ca_count is 0 (zero)), the destination file's
metadata is implementation dependent.

If the client is requesting a partial file copy (i.e. ca_count is not
0 (zero)), the client SHOULD NOT set the metadata flag and the server
MUST ignore the metadata flag.

If the operation does not result in an immediate failure, the server
will return NFS4_OK, and the CURRENT_FH will remain the destination's
filehandle.

If an immediate failure does occur, cr_bytes_copied will be set to
the number of bytes copied to the destination file before the error
occurred. The cr_bytes_copied value indicates the number of bytes
copied but not which specific bytes have been copied.

A return of NFS4_OK indicates that either the operation is complete
or the operation was initiated and a callback will be used to deliver
the final status of the operation.

If the cr_callback_id is returned, this indicates that the operation
was initiated and a CB_COPY callback will deliver the final results
of the operation. The cr_callback_id stateid is termed a copy
stateid in this context. The server is given the option of returning
the results in a callback because the data may require a relatively
long period of time to copy.

If no cr_callback_id is returned, the operation completed
synchronously and no callback will be issued by the server. The
completion status of the operation is indicated by cr_status.

If the copy completes successfully, either synchronously or
asynchronously, the data copied from the source file to the
destination file MUST appear identical to the NFS client. However,
the NFS server's on disk representation of the data in the source
file and destination file MAY differ. For example, the NFS server
might encrypt, compress, deduplicate, or otherwise represent the on
disk data in the source and destination file differently.

In the event of a failure the state of the destination file is
implementation dependent. The COPY operation may fail for the
following reasons (this is a partial list).

NFS4ERR_MOVED: The file system which contains the source file, or
    the destination file or directory is not present. The client can
    determine the correct location and reissue the operation with the
    correct location.

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the
    NFS server receiving this request.

NFS4ERR_PARTNER_NOTSUPP: The remote server does not support the
    server-to-server copy offload protocol.

NFS4ERR_PARTNER_NO_AUTH: The remote server does not authorize a
    server-to-server copy offload operation. This may be due to the
    client's failure to send the COPY_NOTIFY operation to the remote
    server, the remote server receiving a server-to-server copy
    offload request after the copy lease time expired, or for some
    other permission problem.

NFS4ERR_FBIG: The copy operation would have caused the file to grow
    beyond the server's limit.

NFS4ERR_NOTDIR: The CURRENT_FH is a file and ca_destination has non-
    zero length.

NFS4ERR_WRONG_TYPE: The SAVED_FH is not a regular file.

NFS4ERR_ISDIR: The CURRENT_FH is a directory and ca_destination has
    zero length.

NFS4ERR_INVAL: The source offset or offset plus count are greater
    than or equal to the size of the source file.

NFS4ERR_DELAY: The server does not have the resources to perform the
    copy operation at the current time. The client should retry the
    operation sometime in the future.

NFS4ERR_METADATA_NOTSUPP: The destination file cannot support the
    same metadata as the source file.

NFS4ERR_WRONGSEC: The security mechanism being used by the client
    does not match the server's security policy.

4.5. Operation X: COPY_ABORT - Cancel a server-side copy

   ARGUMENTS

```
                struct COPY_ABORT4args {
                        /* CURRENT_FH: destination file */
                        stateid4        caa_stateid;
                };
```

   RESULTS

```
                struct COPY_ABORT4res {
                        nfsstat4        car_status;
                };
```

   DESCRIPTION

   COPY_ABORT is used for both intra- and inter-server asynchronous
   copies. The COPY_ABORT operation allows the client to cancel a
   server-side copy operation that it initiated. This operation is sent
   in a COMPOUND request from the client to the destination server.
   This operation may be used to cancel a copy when the application that
   requested the copy exits before the operation is completed or for
   some other reason.

   The request contains the filehandle and copy stateid cookies that act
   as the context for the previously initiated copy operation.

   The result's car_status field indicates whether the cancel was
   successful or not. A value of NFS4_OK indicates that the copy
   operation was canceled and no callback will be issued by the server.
   A copy operation that is successfully canceled may result in none,
   some, or all of the data copied.

   If the server supports asynchronous copies, the server is REQUIRED to

support the COPY_ABORT operation.

The COPY_ABORT operation may fail for the following reasons (this is
a partial list):

NFS4ERR_NOTSUPP: The abort operation is not supported by the NFS
   server receiving this request.

NFS4ERR_RETRY: The abort failed, but a retry at some time in the
   future MAY succeed.

NFS4ERR_COMPLETE_ALREADY: The abort failed, and a callback will
   deliver the results of the copy operation.

NFS4ERR_SERVERFAULT: An error occurred on the server that does not
   map to a specific error code.

4.6. Operation Y: COPY_STATUS - Poll for status of a server-side copy

   ARGUMENTS

```
                  struct COPY_STATUS4args {
                          /* CURRENT_FH: destination file */
                          stateid4        csa_stateid;
                  };
```

   RESULTS

```
                  union COPY_STATUS4res switch (nfsstat4 csr_status) {
                  case NFS4_OK:
                          length4         csr_bytes_copied;
                          nfsstat4        csr_complete<1>;
                  default:
                          void;
                  };
```

   DESCRIPTION

   COPY_STATUS is used for both intra- and inter-server asynchronous
   copies. The COPY_STATUS operation allows the client to poll the
   server to determine the status of an asynchronous copy operation.
   This operation is sent by the client to the destination server.

   If this operation is successful, the number of bytes copied are
   returned to the client in the csr_bytes_copied field. The
   csr_bytes_copied value indicates the number of bytes copied but not
   which specific bytes have been copied.

If the optional csr_complete field is present, the copy has
completed. In this case the status value indicates the result of the
asynchronous copy operation. In all cases, the server will also
deliver the final results of the asynchronous copy in a CB_COPY
operation.

The failure of this operation does not indicate the result of the
asynchronous copy in any way.

If the server supports asynchronous copies, the server is REQUIRED to
support the COPY_STATUS operation.

The COPY_STATUS operation may fail for the following reasons (this is
a partial list):

NFS4ERR_NOTSUPP: The copy status operation is not supported by the
   NFS server receiving this request.

NFS4ERR_BAD_STATEID: The stateid is not valid (see Section 4.8
   below).

NFS4ERR_EXPIRED: The stateid has expired (see Copy Offload Stateid
   section below).

4.7. Operation Z: CB_COPY - Report results of a server-side copy

   ARGUMENTS

```
union copy_info4 switch (nfsstat4 cca_status) {
case NFS4_OK:
        void;
default:
        length4        cca_bytes_copied;
};

struct CB_COPY4args {
        nfs_fh4        cca_fh;
        stateid4       cca_stateid;
        copy_info4     cca_copy_info;
};
```

   RESULTS

```
struct CB_COPY4res {
        nfsstat4       ccr_status;
};
```

   DESCRIPTION

CB_COPY is used for both intra- and inter-server asynchronous copies.
The CB_COPY callback informs the client of the result of an
asynchronous server-side copy. This operation is sent by the
destination server to the client in a CB_COMPOUND request. The copy
is identified by the filehandle and stateid arguments. The result is
indicated by the status field. If the copy failed, cca_bytes_copied
contains the number of bytes copied before the failure occurred. The
cca_bytes_copied value indicates the number of bytes copied but not
which specific bytes have been copied.

In the absence of an established backchannel, the server cannot
signal the completion of the COPY via a CB_COPY callback. The loss
of a callback channel would be indicated by the server setting the
SEQ4_STATUS_CB_PATH_DOWN flag in the sr_status_flags field of the
SEQUENCE operation. The client must re-establish the callback
channel to receive the status of the COPY operation. Prolonged loss
of the callback channel could result in the server dropping the COPY
operation state and invalidating the copy stateid.

If the client supports the COPY operation, the client is REQUIRED to
support the CB_COPY operation.

The CB_COPY operation may fail for the following reasons (this is a
partial list):

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the
   NFS client receiving this request.

4.8. Copy Offload Stateids

A server may perform a copy offload operation asynchronously. An
asynchronous copy is tracked using a copy offload stateid. Copy
offload stateids are included in the COPY, COPY_ABORT, COPY_STATUS,
and CB_COPY operations.

Section 8.2.4 of [RFC5661] specifies that stateids are valid until
either (A) the client or server restart or (B) the client returns the
resource.

A copy offload stateid will be valid until either (A) the client or
server restart or (B) the client returns the resource by issuing a
COPY_ABORT operation or the client replies to a CB_COPY operation.

A copy offload stateid's seqid MUST NOT be 0 (zero). In the context
of a copy offload operation, it is ambiguous to indicate the most
recent copy offload operation using a stateid with seqid of 0 (zero).
Therefore a copy offload stateid with seqid of 0 (zero) MUST be
considered invalid.

5. Security Considerations

   The security considerations pertaining to NFSv4 [RFC3530] apply to
   this document.

   The standard security mechanisms provide by NFSv4 [RFC3530] may be
   used to secure the protocol described in this document.

   NFSv4 clients and servers supporting the the inter-server copy
   operations described in this document are REQUIRED to implement
   [RPCSEC_GSSv3], including the RPCSEC_GSSv3 privileges copy_from_auth
   and copy_to_auth. If the server-to-server copy protocol is ONC RPC
   based, the servers are also REQUIRED to implement the RPCSEC_GSSv3
   privilege copy_confirm_auth. These requirements to implement are not
   requirements to use. NFSv4 clients and servers are RECOMMENDED to
   use [RPCSEC_GSSv3] to secure server-side copy operations.

5.1. Inter-Server Copy Security

5.1.1. Requirements for Secure Inter-Server Copy

   Inter-server copy is driven by several requirements:

   o The specification MUST NOT mandate an inter-server copy protocol.
      There are many ways to copy data. Some will be more optimal than
      others depending on the identities of the source server and
      destination server. For example the source and destination
      servers might be two nodes sharing a common file system format for
      the source and destination file systems. Thus the source and
      destination are in an ideal position to efficiently render the
      image of the source file to the destination file by replicating
      the file system formats at the block level. In other cases, the
      source and destination might be two nodes sharing a common storage
      area network, and thus there is no need to copy any data at all,
      and instead ownership of the file and its contents simply gets re-
      assigned to the destination.

   o The specification MUST provide guidance for using NFSv4.x as a
      copy protocol. For those source and destination servers willing
      to use NFSv4.x there are specific security considerations that
      this specification can and does address.

   o The specification MUST NOT mandate pre-configuration between the
      source and destination server. Requiring that the source and
      destination first have a "copying relationship" increases the
      administrative burden. However the specification MUST NOT
      preclude implementations that require pre-configuration.

   o The specification MUST NOT mandate a trust relationship between
      the source and destination server. The NFSv4 security model
      requires mutual authentication between a principal on an NFS
      client and a principal on an NFS server. This model MUST continue
      with the introduction of COPY.

5.1.2. Inter-Server Copy with RPCSEC_GSSv3

   When the client sends a COPY_NOTIFY to the source server to expect
   the destination to attempt to copy data from the source server, it is
   expected that this copy is being done on behalf of the principal
   (called the "user principal") that sent the RPC request that encloses
   the COMPOUND procedure that contains the COPY_NOTIFY operation. The
   user principal is identified by the RPC credentials. A mechanism
   that allows the user principal to authorize the destination server to
   perform the copy in a manner that lets the source server properly
   authenticate the destination's copy, and without allowing the
   destination to exceed its authorization is necessary.

   An approach that sends delegated credentials of the client's user
   principal to the destination server is not used for the following
   reasons. If the client's user delegated its credentials, the
   destination would authenticate as the user principal. If the
   destination were using the NFSv4 protocol to perform the copy, then
   the source server would authenticate the destination server as the
   user principal, and the file copy would securely proceed. However,
   this approach would allow the destination server to copy other files.
   The user principal would have to trust the destination server to not
   do so. This is counter to the requirements, and therefore is not
   considered. Instead an approach using RPCSEC_GSSv3 [RPCSEC_GSSv3]
   privileges is proposed.

   One of the stated applications of the proposed RPCSEC_GSSv3 protocol
   is compound client host and user authentication [+ privilege
   assertion]. For inter-server file copy, we require compound NFS
   server host and user authentication [+ privilege assertion]. The
   distinction between the two is one without meaning.

   RPCSEC_GSSv3 introduces the notion of privileges. We define three
   privileges:

   copy_from_auth: A user principal is authorizing a source principal
      ("nfs@<source>") to allow a destination principal ("nfs@
      <destination>") to copy a file from the source to the destination.
      This privilege is established on the source server before the user
      principal sends a COPY_NOTIFY operation to the source server.

```
        typedef string secret4<>;

        struct copy_from_auth_priv {
            secret4             cfap_shared_secret;
            netloc4             cfap_destination;
            /* the NFSv4 user name that the user principal maps to */
            utf8str_mixed       cfap_username;
            /* equal to seq_num of rpc_gss_cred_vers_3_t */
            unsigned int        cfap_seq_num;
        };
```

cap_shared_secret is a secret value the user principal generates.

copy_to_auth: A user principal is authorizing a destination
   principal ("nfs@<destination>") to allow it to copy a file from
   the source to the destination. This privilege is established on
   the destination server before the user principal sends a COPY
   operation to the destination server.

```
        struct copy_to_auth_priv {
            /* equal to cfap_shared_secret */
            secret4             ctap_shared_secret;
            netloc4             ctap_source;
            /* the NFSv4 user name that the user principal maps to */
            utf8str_mixed       ctap_username;
            /* equal to seq_num of rpc_gss_cred_vers_3_t */
            unsigned int        ctap_seq_num;
        };
```

ctap_shared_secret is a secret value the user principal generated
and was used to establish the copy_from_auth privilege with the
source principal.

copy_confirm_auth: A destination principal is confirming with the
   source principal that it is authorized to copy data from the
   source on behalf of the user principal. When the inter-server
   copy protocol is NFSv4, or for that matter, any protocol capable
   of being secured via RPCSEC_GSSv3 (i.e. any ONC RPC protocol),
   this privilege is established before the file is copied from the
   source to the destination.

```
        struct copy_confirm_auth_priv {
            /* equal to GSS_GetMIC() of cfap_shared_secret */
            opaque              ccap_shared_secret_mic<>;
            /* the NFSv4 user name that the user principal maps to */
            utf8str_mixed       ccap_username;
            /* equal to seq_num of rpc_gss_cred_vers_3_t */
            unsigned int        ccap_seq_num;
        };
```

5.1.2.1. Establishing a Security Context

   When the user principal wants to COPY a file between two servers, if
   it has not established copy_from_auth and copy_to_auth privileges on
   the servers, it establishes them:

   o The user principal generates a secret it will share with the two
      servers. This shared secret will be placed in the
      cfap_shared_secret and ctap_shared_secret fields of the
      appropriate privilege data types, copy_from_auth_priv and
      copy_to_auth_priv.

   o An instance of copy_from_auth_priv is filled in with the shared
      secret, the destination server, and the NFSv4 user id of the user
      principal. It will be sent with an RPCSEC_GSS3_CREATE procedure,
      and so cfap_seq_num is set to the seq_num of the credential of the
      RPCSEC_GSS3_CREATE procedure. Because cfap_shared_secret is a
      secret, after XDR encoding copy_from_auth_priv, GSS_Wrap() (with
      privacy) is invoked on copy_from_auth_priv. The
      RPCSEC_GSS3_CREATE procedure's arguments are:


```
        struct {
            rpc_gss3_gss_binding    *compound_binding;
            rpc_gss3_chan_binding   *chan_binding_mic;
            rpc_gss3_assertion      assertions<>;
            rpc_gss3_extension      extensions<>;
        } rpc_gss3_create_args;
```


   The string "copy_from_auth" is placed in assertions[0].privs. The
   output of GSS_Wrap() is placed in extensions[0].data. The field
   extensions[0].critical is set to TRUE. The source server calls
   GSS_Unwrap() on the privilege, and verifies that the seq_num
   matches the credential. It then verifies that the NFSv4 user id
   being asserted matches the source server's mapping of the user
   principal. If it does, the privilege is established on the source
   server as: <"copy_from_auth", user id, destination>. The
   successful reply to RPCSEC_GSS3_CREATE has:

```
        struct {
            opaque                  handle<>;
            rpc_gss3_chan_binding   *chan_binding_mic;
            rpc_gss3_assertion      granted_assertions<>;
            rpc_gss3_assertion      server_assertions<>;
            rpc_gss3_extension      extensions<>;
        } rpc_gss3_create_res;
```

The field "handle" is the RPCSEC_GSSv3 handle that the client will use on COPY_NOTIFY requests involving the source and destination server. granted_assertions[0].privs will be equal to "copy_from_auth". The server will return a GSS_Wrap() of copy_to_auth_priv.

o An instance of copy_to_auth_priv is filled in with the shared secret, the source server, and the NFSv4 user id. It will be sent with an RPCSEC_GSS3_CREATE procedure, and so ctap_seq_num is set to the seq_num of the credential of the RPCSEC_GSS3_CREATE procedure. Because ctap_shared_secret is a secret, after XDR encoding copy_to_auth_priv, GSS_Wrap() is invoked on copy_to_auth_priv. The RPCSEC_GSS3_CREATE procedure's arguments are:

```
        struct {
            rpc_gss3_gss_binding    *compound_binding;
            rpc_gss3_chan_binding   *chan_binding_mic;
            rpc_gss3_assertion      assertions<>;
            rpc_gss3_extension      extensions<>;
        } rpc_gss3_create_args;
```

The string "copy_to_auth" is placed in assertions[0].privs. The output of GSS_Wrap() is placed in extensions[0].data. The field extensions[0].critical is set to TRUE. After unwrapping, verifying the seq_num, and the user principal to NFSv4 user ID mapping, the destination establishes a privilege of <"copy_to_auth", user id, source>. The successful reply to RPCSEC_GSS3_CREATE has:

```
        struct {
            opaque                  handle<>;
            rpc_gss3_chan_binding   *chan_binding_mic;
            rpc_gss3_assertion      granted_assertions<>;
            rpc_gss3_assertion      server_assertions<>;
            rpc_gss3_extension      extensions<>;
```

```
} rpc_gss3_create_res;
```

The field "handle" is the RPCSEC_GSSv3 handle that the client will
use on COPY requests involving the source and destination server.
The field granted_assertions[0].privs will be equal to
"copy_to_auth". The server will return a GSS_Wrap() of
copy_to_auth_priv.

5.1.2.2. Starting a Secure Inter-Server Copy

When the client sends a COPY_NOTIFY request to the source server, it
uses the privileged "copy_from_auth" RPCSEC_GSSv3 handle.
cna_destination_server in COPY_NOTIFY MUST be the same as the name of
the destination server specified in copy_from_auth_priv. Otherwise,
COPY_NOTIFY will fail with NFS4ERR_ACCESS. The source server
verifies that the privilege <"copy_from_auth", user id, destination>
exists, and annotates it with the source filehandle, if the user
principal has read access to the source file, and if administrative
policies give the user principal and the NFS client read access to
the source file (i.e. if the ACCESS operation would grant read
access). Otherwise, COPY_NOTIFY will fail with NFS4ERR_ACCESS.

When the client sends a COPY request to the destination server, it
uses the privileged "copy_to_auth" RPCSEC_GSSv3 handle.
ca_source_server in COPY MUST be the same as the name of the source
server specified in copy_to_auth_priv. Otherwise, COPY will fail
with NFS4ERR_ACCESS. The destination server verifies that the
privilege <"copy_to_auth", user id, source> exists, and annotates it
with the source and destination filehandles. If the client has
failed to establish the "copy_to_auth" policy it will reject the
request with NFS4ERR_PARTNER_NO_AUTH.

If the client sends a COPY_REVOKE to the source server to rescind the
destination server's copy privilege, it uses the privileged
"copy_from_auth" RPCSEC_GSSv3 handle and the cra_destination_server
in COPY_REVOKE MUST be the same as the name of the destination server
specified in copy_from_auth_priv. The source server will then delete
the <"copy_from_auth", user id, destination> privilege and fail any
subsequent copy requests sent under the auspices of this privilege
from the destination server.

5.1.2.3. Securing ONC RPC Server-to-Server Copy Protocols

After a destination server has a "copy_to_auth" privilege established
on it, and it receives a COPY request, if it knows it will use an ONC
RPC protocol to copy data, it will establish a "copy_confirm_auth"
privilege on the source server, using nfs@<destination> as the

initiator principal, and nfs@<source> as the target principal.

The value of the field ccap_shared_secret_mic is a GSS_VerifyMIC() of the shared secret passed in the copy_to_auth privilege. The field ccap_username is the mapping of the user principal to an NFSv4 user name ("user"@"domain" form), and MUST be the same as ctap_username and cfap_username. The field ccap_seq_num is the seq_num of the RPCSEC_GSSv3 credential used for the RPCSEC_GSS3_CREATE procedure the destination will send to the source server to establish the privilege.

The source server verifies the privilege, and establishes a <"copy_confirm_auth", user id, destination> privilege. If the source server fails to verify the privilege, the COPY operation will be rejected with NFS4ERR_PARTNER_NO_AUTH. All subsequent ONC RPC requests sent from the destination to copy data from the source to the destination will use the RPCSEC_GSSv3 handle returned by the source's RPCSEC_GSS3_CREATE response.

Note that the use of the "copy_confirm_auth" privilege accomplishes the following:

o if a protocol like NFS is being used, with export policies, export policies can be overridden in case the destination server as-an-NFS-client is not authorized

o manual configuration to allow a copy relationship between the source and destination is not needed.

If the attempt to establish a "copy_confirm_auth" privilege fails, then when the user principal sends a COPY request to destination, the destination server will reject it with NFS4ERR_PARTNER_NO_AUTH.

5.1.2.4. Securing Non ONC RPC Server-to-Server Copy Protocols

If the destination won't be using ONC RPC to copy the data, then the source and destination are using an unspecified copy protocol. The destination could use the shared secret and the NFSv4 user id to prove to the source server that the user principal has authorized the copy.

For protocols that authenticate user names with passwords (e.g. HTTP [RFC2616] and FTP [RFC0959]), the nfsv4 user id could be used as the user name, and an ASCII hexadecimal representation of the RPCSEC_GSSv3 shared secret could be used as the user password or as input into non-password authentication methods like CHAP [RFC1994].

5.1.3. Inter-Server Copy via ONC RPC but without RPCSEC_GSSv3

   ONC RPC security flavors other than RPCSEC_GSSv3 MAY be used with the
   server-side copy offload operations described in this document. In
   particular, host-based ONC RPC security flavors such as AUTH_NONE and
   AUTH_SYS MAY be used. If a host-based security flavor is used, a
   minimal level of protection for the server-to-server copy protocol is
   possible.

   In the absence of strong security mechanisms such as RPCSEC_GSSv3,
   the challenge is how the source server and destination server
   identify themselves to each other, especially in the presence of
   multi-homed source and destination servers. In a multi-homed
   environment, the destination server might not contact the source
   server from the same network address specified by the client in the
   COPY_NOTIFY. This can be overcome using the procedure described
   below.

   When the client sends the source server the COPY_NOTIFY operation,
   the source server may reply to the client with a list of target
   addresses, names, and/or URLs and assign them to the unique triple:
   <source fh, user ID, destination address Y>. If the destination uses
   one of these target netlocs to contact the source server, the source
   server will be able to uniquely identify the destination server, even
   if the destination server does not connect from the address specified
   by the client in COPY_NOTIFY.

   For example, suppose the network topology is as shown in Figure 3.
   If the source filehandle is 0x12345, the source server may respond to
   a COPY_NOTIFY for destination 10.11.78.56 with the URLs:

      nfs://10.11.78.18//_COPY/10.11.78.56/_FH/0x12345

      nfs://192.168.33.18//_COPY/10.11.78.56/_FH/0x12345

   The client will then send these URLs to the destination server in the
   COPY operation. Suppose that the 192.168.33.0/24 network is a high
   speed network and the destination server decides to transfer the file
   over this network. If the destination contacts the source server
   from 192.168.33.56 over this network using NFSv4.1, it does the
   following:

   COMPOUND { PUTROOTFH, LOOKUP "_COPY" ; LOOKUP "10.11.78.56"; LOOKUP
      "_FH" ; OPEN "0x12345" ; GETFH }

   The source server will therefore know that these NFSv4.1 operations
   are being issued by the destination server identified in the
   COPY_NOTIFY.

5.1.4. Inter-Server Copy without ONC RPC and RPCSEC_GSSv3

   The same techniques as Section 5.1.3, using unique URLs for each
   destination server, can be used for other protocols (e.g. HTTP
   [RFC2616] and FTP [RFC0959]) as well.

6. IANA Considerations

   This document has no actions for IANA.

7. References

7.1. Normative References

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC3530]   Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
               Beame, C., Eisler, M., and D. Noveck, "Network File System
               (NFS) version 4 Protocol", RFC 3530, April 2003.

   [RFC3986]   Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
               Resource Identifier (URI): Generic Syntax", STD 66,
               RFC 3986, January 2005.

   [RFC5661]   Shepler, S., Eisler, M., and D. Noveck, "Network File
               System (NFS) Version 4 Minor Version 1 Protocol",
               RFC 5661, January 2010.

   [RPCSEC_GSSv3]
               Williams, N., "Remote Procedure Call (RPC) Security
               Version 3", draft-williams-rpcsecgssv3 (work in progress),
               2008.

7.2. Informational References

   [FEDFS-ADMIN]
               Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M.
               Naik, "Administration Protocol for Federated Filesystems",
               draft-ietf-nfsv4-federated-fs-admin (Work In Progress),
               2010.

   [FEDFS-NSDB]
               Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M.
               Naik, "NSDB Protocol for Federated Filesystems",
               draft-ietf-nfsv4-federated-fs-protocol (Work In Progress),

          2010.

   [RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol",
              STD 9, RFC 959, October 1985.

   [RFC1994] Simpson, W., "PPP Challenge Handshake Authentication
              Protocol (CHAP)", RFC 1994, August 1996.

   [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,
              Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext
              Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

   [SPACE-RESERVE]
              Eisler, M., Kenchammana, D., Lentini, J., Shankararao, M.,
              and R. Iyer, "NFS space reservation operations",
              draft-iyer-nfsv4-space-reservation-ops (work in progress),
              2010.

Appendix A. Acknowledgments

   Tom Talpey co-authored an unpublished version of this document. We
   thank Tom for his contributions, especially with regards to the
   asynchronous completion callback mechanism.

   This document was reviewed by a number of individuals. We would like
   to thank Pranoop Erasani, Tom Haynes, Arthur Lent, Trond Myklebust,
   Dave Noveck, Theresa Lingutla-Raj, Manjunath Shankararao, Satyam
   Vaghani, and Nico Williams for their input and advice.

Authors' Addresses

   James Lentini
   NetApp
   1601 Trapelo Rd, Suite 16
   Waltham, MA 02451
   USA

   Phone: +1 781-768-5359
   Email: jlentini@netapp.com

Mike Eisler
NetApp
5765 Chase Point Circle
Colorado Springs, CO 80919
USA

Phone: +1 719-599-9026
Email: mike@eisler.com
URI: http://www.eisler.com

Deepak Kenchammana
NetApp
475 East Java Drive
Sunnyvale, CA 94089
USA

Phone: +1 408-822-4765
Email: kencham@netapp.com

Anshul Madan
Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15213
USA

Email: anshulmadan@cmu.edu

Rahul Iyer
655 S Fair Oaks Ave
Apt #I-314
Sunnyvale, CA 94086
USA

Email: rahulair@yahoo.com

      NFS Version 4 Minor Version 2 unstable file creation and attribute
                         update improvements
           draft-myklebust-nfsv42-unstable-file-creation-00

Abstract

   This document describes an extension to the NFSv4 protocol to allow
   clients to create and write files with greater efficiency.

   The first proposal allows the server to defer creating the file on
   stable storage when replying to an OPEN call. The aim is to improve
   server efficiency and scalability by reducing the number of required
   disk accesses when writing a file from scratch.

   The second proposal allows the server to share information about the
   implementation of its change attribute with the client. The aim is
   to improve the client's ability to determine the order in which
   parallel updates to the same file were processed.

Keywords

   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 [RFC2119].

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 20, 2011.

Copyright Notice

Table of Contents

1. Introduction

   One of the remaining sources of performance and scalability issues in
   the NFSv4.1 protocol [RFC5661], for workloads that require the
   creation of large numbers of files, is that file creation is still
   required to be synchronous. This limitation means that the minimum
   number of disk accesses in a workload that involves creating a file,
   writing to it and then closing it is 2: one at OPEN time, and one at
   COMMIT. The following proposal allows the client to indicate to the
   server, by means of a new attribute, that it is prepared to take on
   the burden of re-creating the file from scratch if the server should
   reboot before the file has been fully written. The same attribute
   also allows the client to check on the state of the file on the
   server, and thus perhaps to optimise away unnecessary COMMIT
   requests.

   Another frequent source of inefficiencies is due to the lack of
   clarity in the protocol defining the change attribute. While the
   change attribute itself is a mandatory attribute, it is not
   sufficiently well defined to allow the client to conclude which value
   represents the current state of the file, after two COMPOUNDs, both
   containing WRITE and GETATTR requests for the same file, have been
   sent in parallel. In some cases, the only recourse available to the
   client may be to send a third COMPOUND containing a GETATTR after
   receiving the responses to the first two. The solution is to allow
   the server to share details about how the change attribute is
   expected to evolve in this kind of situation.

2. Definition of the 'stable_state' per-file attribute

```
const NFS4_UNSTABLE_METADATA = 0x00000001;
const NFS4_UNSTABLE_DATA     = 0x00000002;
const NFS4_UNSTABLE_PNFS     = 0x00000004;
```

```
+--------------+----+-----------+-----+
| Name         | Id | Data Type | Acc |
+--------------+----+-----------+-----+
| stable_state | XX | uint32_t  | R W |
+--------------+----+-----------+-----+
```

The attribute 'stable_state' is an optional per-file attribute that
can be used by the client to determine whether or not the server
believes that all metadata and data has been committed to persistent
storage. It is expected that clients may wish to poll it as part of
a post-op attribute request or an attribute refresh.
o If the server returns a zero value, then the client may assume
   that all metadata and data changes that were made since the server
   last rebooted have been committed to persistent storage.
o If the server sets the bits NFS4_UNSTABLE_METADATA and/or
   NFS4_UNSTABLE_DATA, then this means that there may be respectively
   metadata, or data that has not been synced to disk. The client
   should be prepared to send a COMMIT request in order to ensure
   persistence of metadata and data.
o If the server sets the bit NFS4_UNSTABLE_PNFS, then this indicates
   that there are outstanding layouts for write, and thus the state
   of the file may not be fully known to the server.

A naive server may choose to implement 'stable_state' in terms of a
simple flag: it sets NFS4_UNSTABLE_DATA when it receives an unstable
WRITE request, sets NFS4_UNSTABLE_METADATA when it receives an
unstable OPEN or SETATTR requests and clears both flags when it
receives a COMMIT. While such an implementation may not be as useful
for avoiding unnecessary COMMIT operations, it is sufficient to
support unstable OPEN and SETATTR.

2.1. Use of the 'stable_state' attribute for unstable OPEN requests

We propose a new mode of file creation named "unstable file
creation". By choosing this mode of creation, the client is
notifying the server that it may defer syncing to disk the new file's
directory entry as well as the new file metadata. In case of a
server reboot, the client is then responsible for replaying the file
creation if the reboot occurred before the file metadata was
committed to disk.

2.1.1. Client use of the 'stable_state' attribute

   In order to indicate that the client wishes to have the server use
   unstable file creation, it must set the NFS4_UNSTABLE_METADATA bit in
   the optional attribute 'stable_state'. Upon return of the OPEN call,
   the client then checks that 'stable_state' was indeed set by
   inspecting the 'attrset' bitmap in the usual way. It can assume that
   if the 'stable_state' was not set, then the file has been created in
   persistent storage.

   The client MUST NOT set the 'stable_state' to any value other than
   NFS4_UNSTABLE_METADATA. The server SHOULD return NFS4ERR_INVAL if it
   receives an invalid value.

   Once the client is done making changes to the file, it may use a
   COMMIT to force the server to flush all data and metadata changes to
   persistent storage.

2.1.2. Server response upon receiving an unstable OPEN request

   Upon receiving an OPEN request that includes a 'stable_state'
   attribute, the server MAY choose to ignore it, and simply apply the
   NFSv4.1 rule that all metadata must be committed to persistent
   storage. If so, it simply omits the 'stable_state' bit from the
   returned attribute bitmap.

   The server MUST NOT set the 'stable_state' flag if the file already
   exists.

   If the server does choose to honour the 'stable_state' attribute,
   then it MUST also return a write delegation to the client. This
   write delegation is needed in order to allow the client to detect the
   recovery edge condition in which a second client attempts to rename
   the file or delete it just prior to a server reboot.

   Once the file has been created in the server cache memory, the server
   is then free to process the remaining elements of the COMPOUND
   without syncing the new file metadata to disk.

2.1.3. Delegation return and unstable OPEN

   If the client returns the write delegation, then it MUST ensure that
   the file metadata is in a stable state. It does so by sending a
   COMMIT operation, unless polling has already established that the
   'stable_state' attribute no longer sets the NFS4_UNSTABLE_METADATA
   bit.

2.1.4. Client unstable OPEN recovery in case of a server reboot

```
enum open_claim_type4 {
        /*
         * Not a reclaim.
         */
        CLAIM_NULL              = 0,

        CLAIM_PREVIOUS          = 1,
        CLAIM_DELEGATE_CUR      = 2,
        CLAIM_DELEGATE_PREV     = 3,

        /*
         * Not a reclaim.
         *
         * Like CLAIM_NULL, but object identified
         * by the current filehandle.
         */
        CLAIM_FH                = 4, /* new to v4.1 */

        /*
         * Like CLAIM_DELEGATE_CUR, but object identified
         * by current filehandle.
         */
        CLAIM_DELEG_CUR_FH      = 5, /* new to v4.1 */

        /*
         * Like CLAIM_DELEGATE_PREV, but object identified
         * by current filehandle.
         */
        CLAIM_DELEG_PREV_FH     = 6, /* new to v4.1 */

        /*
         * Like CLAIM_PREVIOUS, but object identified
         * by directory filehandle + filename.
         */
        CLAIM_PREVIOUS_UNSTABLE = 7
};

union open_claim4 switch (open_claim_type4 claim) {
 /*
  * No special rights to file.
  * Ordinary OPEN of the specified file.
  */
 case CLAIM_NULL:
        /* CURRENT_FH: directory */
        component4      file;

 /*
  * Right to the file established by an
```

```
    * open previous to server reboot.  File
    * identified by filehandle obtained at
    * that time rather than by name.
    */
   case CLAIM_PREVIOUS:
          /* CURRENT_FH: file being reclaimed */
          open_delegation_type4   delegate_type;

   /*
    * Right to file based on a delegation
    * granted by the server.  File is
    * specified by name.
    */
   case CLAIM_DELEGATE_CUR:
          /* CURRENT_FH: directory */
          open_claim_delegate_cur4        delegate_cur_info;

   /*
    * Right to file based on a delegation
    * granted to a previous boot instance
    * of the client.  File is specified by name.
    */
   case CLAIM_DELEGATE_PREV:
          /* CURRENT_FH: directory */
          component4       file_delegate_prev;

   /*
    * Like CLAIM_NULL.  No special rights
    * to file.  Ordinary OPEN of the
    * specified file by current filehandle.
    */
   case CLAIM_FH: /* new to v4.1 */
          /* CURRENT_FH: regular file to open */
          void;

   /*
    * Like CLAIM_DELEGATE_PREV.  Right to file based on a
    * delegation granted to a previous boot
    * instance of the client.  File is identified by
    * by filehandle.
    */
   case CLAIM_DELEG_PREV_FH: /* new to v4.1 */
          /* CURRENT_FH: file being opened */
          void;

   /*
    * Like CLAIM_DELEGATE_CUR.  Right to file based on
    * a delegation granted by the server.
```

```
     * File is identified by filehandle.
     */
    case CLAIM_DELEG_CUR_FH: /* new to v4.1 */
            /* CURRENT_FH: file being opened */
            stateid4        oc_delegate_stateid;

   /*
    * Right to the file established by an
    * unstable open previous to server reboot.
    * File is specified by name.
    */
    case CLAIM_PREVIOUS_UNSTABLE: /* new to v4.2 */
           /* CURRENT_FH: directory */
           component4      file_previous_unstable;
  };
```

A server that supports unstable file creation SHOULD reject all
CREATE and ordinary file creation attempts during the grace period
using the error NFS4ERR_GRACE in order to allow clients to recover
any unstable files that may have been lost.

In order to recover the file, the client MUST replay the original
OPEN that was used to create the file, using an open claim type of
CLAIM_PREVIOUS_UNSTABLE.
o If the server discovers that the file already exists, it treats
   the OPEN as if it were a CLAIM_PREVIOUS request for a write
   delegation.
o If the file does not exist, then the server creates the file in
   the usual fashion and returns a valid write delegation.

2.1.5. Directory cache consistency and unstable files

While the client that created the file can easily recover in case of
a server reboot, it is not necessarily so easy for other clients to
do so. While the write delegation does indeed ensure that those
clients do not hold the file open (neither do they hold any cached
data), it does not guarantee that they are not caching LOOKUP or
READDIR data.

In order to avoid issues with directory cache consistency across
server reboots, it is therefore RECOMMENDED that servers ensure that
initial file metadata be committed to persistent storage prior to
replying to a another client's LOOKUP of the new file, or READDIR of
the directory in which the new file was created. This will also
prevent those clients from seeing filehandles and fileids that might
change upon server reboot.

2.2. Use of the 'stable_state' attribute in unstable SETATTR requests

   If it holds a write delegation, the client may also use the
   'stable_state' attribute in a SETATTR request to indicate to the
   server that it is ready to replay this SETATTR in the case of a
   server reboot.

   The procedure is the same as for OPEN. In order to indicate to the
   server that it wants the SETATTR request to be unstable, the client
   sets the 'stable_state' attribute to the value
   NFS4_UNSTABLE_METADATA.

   Again, the server MAY ignore the 'stable_state' attribute, in which
   case it MUST immediately commit the attributes to stable storage, and
   MUST clear the 'stable_state' bit in the returned attribute bitmap.

   If the client does not hold a valid write delegation, then the server
   MUST also ignore the 'stable_state' attribute.

2.2.1. Client unstable SETATTR recovery in case of a server reboot

   If the server reboots before the client has had a chance to issue a
   COMMIT, then after recovering the write delegation, the client SHOULD
   check the server attributes against its own cached values. If there
   is a mismatch, then it is responsible for correcting this by
   replaying the relevant SETATTR calls.

2.2.2. Delegation return and unstable SETATTR

   If the client returns the write delegation, then it MUST ensure that
   the file metadata is in a stable state. It does so by sending a
   COMMIT operation, unless polling has already established that the
   'stable_state' attribute no longer sets the NFS4_UNSTABLE_METADATA
   bit.

3. Definition of the 'change_attr_type' per-file system attribute

```
enum change_attr_typeinfo = {
        NFS4_CHANGE_TYPE_IS_MONOTONIC_INCR          = 0,
        NFS4_CHANGE_TYPE_IS_VERSION_COUNTER         = 1,
        NFS4_CHANGE_TYPE_IS_VERSION_COUNTER_NOPNFS  = 2,
        NFS4_CHANGE_TYPE_IS_TIME_METADATA           = 3,
        NFS4_CHANGE_TYPE_IS_UNDEFINED               = 4
};
```

```
+------------------+----+-------------------------+-----+
| Name             | Id | Data Type               | Acc |
+------------------+----+-------------------------+-----+
| change_attr_type | XX | enum change_attr_typeinfo | R   |
+------------------+----+-------------------------+-----+
```

Although the original NFSv4 protocol [RFC3530] does describe a
possible implementation of the change attribute in terms of the
time_metadata attribute, it does little to limit the implementation
other than to state that the value changes if the file data,
directory contents or attributes change.

While this allows for a wide range of implementations, it also leaves
the client with a conundrum: how does it determine which is the most
recent value for the change attribute in a case where several RPC
calls have been issued in parallel?

The proposed solution is to have the NFS server provide additional
information about how it expects the change attribute value to
evolve. To do so, we provide for a new optional attribute,
'change_attr_type', which may take values from enum
change_attr_typeinfo as follows:
NFS4_CHANGE_TYPE_IS_MONOTONIC_INCR: The change attribute MUST change
   in a monotonically increasing manner.
NFS4_CHANGE_TYPE_IS_VERSION_COUNTER: The change attribute MUST
   increment by the value "1" for every atomic change to the file
   data, attributes or directory contents. This property is
   preserved when writing to pNFS data servers.
NFS4_CHANGE_TYPE_IS_VERSION_COUNTER_NOPNFS: The change attribute
   MUST increment by the value "1" for every atomic change to the
   file data, attributes or directory contents. In the case where
   the client is writing to pNFS data servers, the number of
   increments is not guaranteed to exactly match the number of
   writes.

NFS4_CHANGE_TYPE_IS_TIME_METADATA: The change attribute is
    implemented as suggested in the NFSv4 spec [RFC3530] in terms of
    the time_metadata attribute.
NFS4_CHANGE_TYPE_IS_UNDEFINED: The change attribute does not take
    values that fit into any of these categories.

If either NFS4_CHANGE_TYPE_IS_MONOTONIC_INCR,
NFS4_CHANGE_TYPE_IS_VERSION_COUNTER, or
NFS4_CHANGE_TYPE_IS_TIME_METADATA are set, then the client knows at
the very least that the change attribute is monotonically increasing,
which is sufficient to resolve the question of which value is the
most recent.

If the client sees the value NFS4_CHANGE_TYPE_IS_TIME_METADATA, then
by inspecting the value of the 'time_delta' attribute it additionally
has the option of detecting rogue server implementations that use
time_metadata in violation of the spec.

Finally, if the client sees NFS4_CHANGE_TYPE_IS_VERSION_COUNTER, it
has the ability to predict what the resulting change attribute value
should be after a COMPOUND containing a SETATTR, WRITE, or CREATE.
This again allows it to detect changes made in parallel by another
client. The value NFS4_CHANGE_TYPE_IS_VERSION_COUNTER_NOPNFS permits
the same, but only if the client is not doing pNFS WRITEs.

4. References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", RFC 2119.

   [RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R.,
              Beame, C., Eisler, M., and D. Noveck, "Network File System
              (NFS) version 4 Protocol", RFC 3530.

   [RFC5661] Shepler, S., Eisler, M., and D. Noveck, "Network File
              System (NFS) Version 4 Minor Version 1 Protocol",
              RFC 5661.

Author's Address

   Trond Myklebust
   NetApp
   3215 Bellflower Ct
   Ann Arbor, MI   48103
   USA

   Phone: +1-734-662-6608
   Email: Trond.Myklebust@netapp.com