NFSv4 Minor Version 1
draft-ietf-nfsv4-minorversion1-10.txt

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on September 5, 2007.

Copyright Notice

Copyright (C) The IETF Trust (2007).

Abstract

This Internet-Draft describes NFSv4 minor version one, including features retained from the base protocol and protocol extensions made subsequently. The current draft includes description of the major extensions, Sessions, Directory Delegations, and parallel NFS (pNFS). This Internet-Draft is an active work item of the NFSv4 working group. Active and resolved issues may be found in the issue tracker at: http://www.nfsv4-editor.org/cgi-bin/roundup/nfsv4. New issues
related to this document should be raised with the NFSv4 Working Group nfsv4@ietf.org and logged in the issue tracker.

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].

Table of Contents

1. Introduction .................................................. 10
   1.1. The NFSv4.1 Protocol .................................. 10
   1.2. NFS Version 4 Goals .................................. 10
   1.3. Minor Version 1 Goals ................................. 11
   1.4. Overview of NFS version 4.1 Features ................. 11
       1.4.1. RPC and Security ................................ 12
       1.4.2. Protocol Structure ............................... 12
       1.4.3. File System Model ............................... 13
       1.4.4. Locking Facilities ............................... 14
   1.5. General Definitions ................................... 15
   1.6. Differences from NFSv4.0 ............................. 17
2. Core Infrastructure ........................................... 17
   2.1. Introduction ........................................... 18
   2.2. RPC and XDR ........................................... 18
       2.2.1. RPC-based Security ............................. 18
   2.3. COMPOUND and CB_COMPOUND ............................ 21
   2.4. Client Identifiers and Client Owners ................. 22
       2.4.1. Server Release of Client ID .................... 26
       2.4.2. Handling Client Owner Conflicts ............... 26
   2.5. Server Owners ......................................... 27
   2.6. Security Service Negotiation ........................ 27
       2.6.1. NFSv4 Security Tuples ......................... 28
       2.6.2. SECINFO and SECINFO_NO_NAME .................. 28
       2.6.3. Security Error ................................. 28
   2.7. Minor Versioning ...................................... 32
   2.8. Non-RPC-based Security Services ...................... 34
       2.8.1. Authorization .................................... 34
       2.8.2. Auditing ........................................ 34
       2.8.3. Intrusion Detection ............................. 35
   2.9. Transport Layers ...................................... 35
       2.9.1. Required and Recommended Properties of Transports 35
       2.9.2. Client and Server Transport Behavior .......... 35
       2.9.3. Ports ............................................ 37
   2.10. Session ............................................... 37
       2.10.1. Motivation and Overview ....................... 37
       2.10.2. NFSv4 Integration .............................. 38
6.2.3. mode Attribute .......................... 116
6.2.4. mode_set_masked Attribute ............. 116
6.3. Common Methods ............................ 117
6.3.1. Interpreting an ACL ................... 117
6.3.2. Computing a Mode Attribute from an ACL 118
6.4. Requirements .............................. 119
6.4.1. Setting the mode and/or ACL Attributes 120
6.4.2. Retrieving the mode and/or ACL Attributes 121
6.4.3. Creating New Objects ................... 122

7. Single-server Name Space ................. 125
7.1. Server Exports .......................... 126
7.2. Browsing Exports ......................... 126
7.3. Server Pseudo File System ............... 126
7.4. Multiple Roots ........................... 127
7.5. Filehandle Volatility .................... 127
7.6. Exported Root ............................ 127
7.7. Mount Point Crossing ..................... 128
7.8. Security Policy and Name Space Presentation 128

8. File Locking and Share Reservations .... 129
8.1. Locking .................................... 130
8.1.1. Client and Session ID ................. 130
8.1.2. State-owner Definition ............... 130
8.1.3. Stateid Definition ..................... 131
8.1.4. Use of the Stateid and Locking ....... 134
8.2. Lock Ranges .............................. 137
8.3. Upgrading and Downgrading Locks ......... 137
8.4. Blocking Locks ............................ 138
8.5. Lease Renewal ............................. 138
8.6. Crash Recovery ............................ 139
8.6.1. Client Failure and Recovery .......... 139
8.6.2. Server Failure and Recovery .......... 140
8.6.3. Network Partitions and Recovery ...... 143
8.7. Server Revocation of Locks ............... 147
8.8. Share Reservations ....................... 148
8.9. OPEN/CLOSE Operations ................... 149
8.10. Open Upgrade and Downgrade .............. 149
8.11. Short and Long Leases .................... 150
8.12. Clocks, Propagation Delay, and Calculating Lease Expiration ............... 151
8.13. Vestigial Locking Infrastructure From V4.0 151

9. Client-Side Caching ......................... 152
9.1. Performance Challenges for Client-Side Caching 153
9.2. Delegation and Callbacks ................. 153
9.2.1. Delegation Recovery .................... 155
9.3. Data Caching ............................. 157
9.3.1. Data Caching and OPENs ................ 157
9.3.2. Data Caching and File Locking .......... 158
9.3.3. Data Caching and Mandatory File Locking 160
11.4. Delegation Recall ........................................ 222
11.5. Directory Delegation Recovery .......................... 222
12. Parallel NFS (pNFS) ........................................... 222
  12.1. Introduction ........................................ 222
  12.2. PNFS Definitions .......................................... 224
    12.2.1. Metadata ........................................... 224
    12.2.2. Metadata Server .................................... 224
    12.2.3. Client .............................................. 225
    12.2.4. Storage Device ...................................... 225
    12.2.5. Data Server ......................................... 225
    12.2.6. Storage Protocol or Data Protocol ................. 225
    12.2.7. Control Protocol .................................... 225
    12.2.8. Layout ................................................ 226
    12.2.9. Layout Types ........................................ 226
    12.2.10. Layout Iomode ........................................ 226
    12.2.11. Layout Segment ...................................... 227
    12.2.12. Device IDs .......................................... 228
  12.3. PNFS Operations ........................................... 228
  12.4. PNFS Attributes ........................................... 229
  12.5. Layout Semantics ......................................... 229
    12.5.1. Guarantees Provided by Layouts ....................... 229
    12.5.2. Getting a Layout ..................................... 230
    12.5.3. Committing a Layout .................................. 231
    12.5.4. Recalling a Layout ................................... 234
    12.5.5. Metadata Server Write Propagation .................. 240
  12.6. PNFS Mechanics ............................................ 240
  12.7. Recovery ................................................ 241
    12.7.1. Client Recovery ....................................... 241
    12.7.2. Dealing with Lease Expiration on the Client ........ 242
    12.7.3. Dealing with Loss of Layout State on the Metadata Server ........................................ 243
    12.7.4. Recovery from Metadata Server Restart .............. 244
    12.7.5. Operations During Metadata Server Grace Period .... 246
    12.7.6. Storage Device Recovery ................................ 246
  12.8. Metadata and Storage Device Roles ....................... 247
  12.9. Security Considerations .................................. 248
  13. PNFS: NFSv4.1 File Layout Type ............................ 249
    13.1. Session Considerations .................................. 249
    13.2. File Layout Definitions .................................. 251
    13.3. File Layout Data Types .................................. 251
    13.4. Interpreting the File Layout .............................. 255
    13.5. Sparse and Dense Stripe Unit Packing .................... 257
    13.6. Data Server Multipathing .................................. 259
    13.7. Operations Issued to NFSv4.1 Data Servers ............... 259
    13.8. COMMIT Through Metadata Server ........................ 260
    13.9. Global Stateid Requirements ............................. 261
    13.10. The Layout Iomode ...................................... 261
    13.11. Data Server State Propagation .......................... 261
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.11.1</td>
<td>Lock State Propagation</td>
<td>262</td>
</tr>
<tr>
<td>13.11.2</td>
<td>Open-mode Validation</td>
<td>262</td>
</tr>
<tr>
<td>13.11.3</td>
<td>File Attributes</td>
<td>263</td>
</tr>
<tr>
<td>13.12</td>
<td>Data Server Component File Size</td>
<td>263</td>
</tr>
<tr>
<td>13.13</td>
<td>Recovery Considerations</td>
<td>264</td>
</tr>
<tr>
<td>13.14</td>
<td>Security Considerations for the File Layout Type</td>
<td>265</td>
</tr>
<tr>
<td>14.</td>
<td>Internationalization</td>
<td>265</td>
</tr>
<tr>
<td>14.1</td>
<td>Stringprep profile for the utf8str_cs type</td>
<td>266</td>
</tr>
<tr>
<td>14.2</td>
<td>Stringprep profile for the utf8str_cis type</td>
<td>268</td>
</tr>
<tr>
<td>14.3</td>
<td>Stringprep profile for the utf8str_mixed type</td>
<td>269</td>
</tr>
<tr>
<td>14.4</td>
<td>UTF-8 Related Errors</td>
<td>271</td>
</tr>
<tr>
<td>15.</td>
<td>Error Values</td>
<td>271</td>
</tr>
<tr>
<td>15.1</td>
<td>Error Definitions</td>
<td>271</td>
</tr>
<tr>
<td>15.2</td>
<td>Operations and their valid errors</td>
<td>285</td>
</tr>
<tr>
<td>15.3</td>
<td>Callback operations and their valid errors</td>
<td>299</td>
</tr>
<tr>
<td>15.4</td>
<td>Errors and the operations that use them</td>
<td>300</td>
</tr>
<tr>
<td>16.</td>
<td>NFS version 4.1 Procedures</td>
<td>307</td>
</tr>
<tr>
<td>16.1</td>
<td>Procedure 0: NULL - No Operation</td>
<td>307</td>
</tr>
<tr>
<td>16.2</td>
<td>Procedure 1: COMPOUND - Compound Operations</td>
<td>308</td>
</tr>
<tr>
<td>17.</td>
<td>NFS version 4.1 Operations</td>
<td>313</td>
</tr>
<tr>
<td>17.1</td>
<td>Operation 3: ACCESS - Check Access Rights</td>
<td>313</td>
</tr>
<tr>
<td>17.2</td>
<td>Operation 4: CLOSE - Close File</td>
<td>315</td>
</tr>
<tr>
<td>17.3</td>
<td>Operation 5: COMMIT - Commit Cached Data</td>
<td>317</td>
</tr>
<tr>
<td>17.4</td>
<td>Operation 6: CREATE - Create a Non-Regular File Object</td>
<td>319</td>
</tr>
<tr>
<td>17.5</td>
<td>Operation 7: DELEGPURGE - Purge Delegations Awaiting Recovery</td>
<td>322</td>
</tr>
<tr>
<td>17.6</td>
<td>Operation 8: DELEGRETURN - Return Delegation</td>
<td>323</td>
</tr>
<tr>
<td>17.7</td>
<td>Operation 9: GETATTR - Get Attributes</td>
<td>323</td>
</tr>
<tr>
<td>17.8</td>
<td>Operation 10: GETFH - Get Current Filehandle</td>
<td>325</td>
</tr>
<tr>
<td>17.9</td>
<td>Operation 11: LINK - Create Link to a File</td>
<td>326</td>
</tr>
<tr>
<td>17.10</td>
<td>Operation 12: LOCK - Create Lock</td>
<td>327</td>
</tr>
<tr>
<td>17.11</td>
<td>Operation 13: LOCKT - Test For Lock</td>
<td>331</td>
</tr>
<tr>
<td>17.12</td>
<td>Operation 14: LOCKU - Unlock File</td>
<td>332</td>
</tr>
<tr>
<td>17.13</td>
<td>Operation 15: LOOKUP - Lookup Filename</td>
<td>334</td>
</tr>
<tr>
<td>17.14</td>
<td>Operation 16: LOOKUPP - Lookup Parent Directory</td>
<td>335</td>
</tr>
<tr>
<td>17.15</td>
<td>Operation 17: NVERIFY - Verify Difference in Attributes</td>
<td>337</td>
</tr>
<tr>
<td>17.16</td>
<td>Operation 18: OPEN - Open a Regular File</td>
<td>338</td>
</tr>
<tr>
<td>17.17</td>
<td>Operation 19: OPENATTR - Open Named Attribute</td>
<td>352</td>
</tr>
<tr>
<td></td>
<td>Directory</td>
<td></td>
</tr>
<tr>
<td>17.18</td>
<td>Operation 21: OPEN_DOWNGRADE - Reduce Open File Access</td>
<td>354</td>
</tr>
<tr>
<td>17.19</td>
<td>Operation 22: PUTFH - Set Current Filehandle</td>
<td>355</td>
</tr>
<tr>
<td>17.20</td>
<td>Operation 23: PUTPUBFH - Set Public Filehandle</td>
<td>356</td>
</tr>
<tr>
<td>17.21</td>
<td>Operation 24: PUTROOTFH - Set Root Filehandle</td>
<td>357</td>
</tr>
<tr>
<td>17.22</td>
<td>Operation 25: READ - Read from File</td>
<td>358</td>
</tr>
<tr>
<td>17.23</td>
<td>Operation 26: READDIR - Read Directory</td>
<td>360</td>
</tr>
<tr>
<td>17.24</td>
<td>Operation 27: READLINK - Read Symbolic Link</td>
<td>364</td>
</tr>
<tr>
<td>17.25</td>
<td>Operation 28: REMOVE - Remove File System Object</td>
<td>365</td>
</tr>
</tbody>
</table>
Internet-Draft

NFSv4 Minor Version 1

17.26.
17.27.
17.28.
17.29.
17.30.
17.31.
17.32.
17.33.
17.34.
17.35.
17.36.

March 2007

Operation 29: RENAME - Rename Directory Entry . . . .
Operation 31: RESTOREFH - Restore Saved Filehandle . .
Operation 32: SAVEFH - Save Current Filehandle . . . .
Operation 34: SETATTR - Set Attributes . . . . . . . .
Operation 37: VERIFY - Verify Same Attributes . . . .
Operation 38: WRITE - Write to File . . . . . . . . .
Operation 40: BACKCHANNEL_CTL - Backchannel control .
Operation 41: BIND_CONN_TO_SESSION . . . . . . . . . .
Operation 42: EXCHANGE_ID - Instantiate Client ID . .
Operation 43: CREATE_SESSION - Create New Session and
Confirm Client ID . . . . . . . . . . . . . . . . . .
17.37. Operation 44: DESTROY_SESSION - Destroy existing
session . . . . . . . . . . . . . . . . . . . . . . .
17.38. Operation 45: FREE_STATEID - Free stateid with no
locks . . . . . . . . . . . . . . . . . . . . . . . .
17.39. Operation 46: GET_DIR_DELEGATION - Get a directory
delegation . . . . . . . . . . . . . . . . . . . . . .
17.41. Operation 48: GETDEVICELIST . . . . . . . . . . . . .
17.42. Operation 49: LAYOUTCOMMIT - Commit writes made using
a layout . . . . . . . . . . . . . . . . . . . . . . .
17.43. Operation 50: LAYOUTGET - Get Layout Information . . .
17.44. Operation 51: LAYOUTRETURN - Release Layout
Information . . . . . . . . . . . . . . . . . . . . .
17.45. Operation 52: SECINFO_NO_NAME - Get Security on
Unnamed Object . . . . . . . . . . . . . . . . . . . .
17.46. Operation 53: SEQUENCE - Supply per-procedure
sequencing and control . . . . . . . . . . . . . . . .
17.47. Operation 54: SET_SSV . . . . . . . . . . . . . . . .
17.48. Operation 55: TEST_STATEID - Test stateids for
validity . . . . . . . . . . . . . . . . . . . . . . .
17.49. Operation 56: WANT_DELEGATION . . . . . . . . . . . .
17.50. Operation 57: DESTROY_CLIENTID - Destroy existing
client ID . . . . . . . . . . . . . . . . . . . . . .
17.51. Operation 10044: ILLEGAL - Illegal operation . . . . .
18. NFS version 4.1 Callback Procedures . . . . . . . . . . . .
18.1. Procedure 0: CB_NULL - No Operation . . . . . . . . .
19. NFS version 4.1 Callback Operations . . . . . . . . . . . .
19.3. Operation 5: CB_LAYOUTRECALL . . . . . . . . . . . . .
19.4. Operation 6: CB_NOTIFY - Notify directory changes . .
19.5. Operation 7: CB_PUSH_DELEG . . . . . . . . . . . . . .
19.7. Operation 9: CB_RECALLABLE_OBJ_AVAIL . . . . . . . . .
19.8. Operation 10: CB_RECALL_SLOT - change flow control

Shepler, et al.

Expires September 5, 2007

.
.
.
.
.
.
.
.
.
.

367
369
370
370
374
376
377
382
383
387

. 395
. 405
. 406
. 407
. 412
. 413
. 414
. 417
. 420
. 423
. 424
. 429
. 431
. 432
.
.
.
.
.
.
.
.
.
.
.
.
.

435
436
437
437
437
439
439
441
442
444
447
448
451

[Page 8]


1. Introduction

1.1. The NFSv4.1 Protocol

The NFSv4.1 protocol is a minor version of the NFSv4 protocol described in [2]. It generally follows the guidelines for minor versioning model laid in Section 10 of RFC 3530. However, it diverges from guidelines 11 ("a client and server that supports minor version X must support minor versions 0 through X-1"), and 12 ("no features may be introduced as mandatory in a minor version"). These divergences are due to the introduction of the sessions model for managing non-idempotent operations and the RECLAIM_COMPLETE operation. These two new features are infrastructural in nature and simplify implementation of existing and other new features. Making them optional would add undue complexity to protocol definition and implementation. NFSv4.1 accordingly updates the Minor Versioning guidelines (Section 2.7).

NFSv4.1, as a minor version, is consistent with the overall goals for NFS Version 4, but extends the protocol so as to better meet those goals, based on experiences with NFSv4.0. In addition, NFSv4.1 has adopted some additional goals, which motivate some of the major extensions in minor version 1.

1.2. NFS Version 4 Goals

The NFS version 4 protocol is a further revision of the NFS protocol defined already by versions 2 [17] and 3 [18]. 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 NFS version 4 revision has the following goals:

- Improved access and good performance on the Internet.
  
The protocol is designed to transit firewalls easily, perform well where latency is high and bandwidth is low, and scale to very large numbers of clients per server.

- Strong security with negotiation built into the protocol.
  
The protocol builds on the work of the ONCRPC working group in supporting the RPCSEC_GSS protocol. Additionally, the NFS version 4 protocol provides a mechanism to allow clients and servers the ability to negotiate security and require clients and servers to support a minimal set of security schemes.
Good cross-platform interoperability.

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

Designed for protocol extensions.

The protocol is designed to accept standard extensions within a framework that enable and encourages backward compatibility.

1.3. Minor Version 1 Goals

Minor version one has the following goals, within the framework established by the overall version 4 goals.

- To correct significant structural weaknesses and oversights discovered in the base protocol.
- To add clarity and specificity to areas left unaddressed or not addressed in sufficient detail in the base protocol.
- To add specific features based on experience with the existing protocol and recent industry developments.
- To provide protocol support to take advantage of clustered server deployments including the ability to provide scalable parallel access to files distributed among multiple servers.

1.4. Overview of NFS version 4.1 Features

To provide a reasonable context for the reader, the major features of NFS version 4.1 protocol will be reviewed in brief. This will be done to provide an appropriate context for both the reader who is familiar with the previous versions of the NFS protocol and the reader that is new to the NFS protocols. For the reader new to the NFS protocols, there is still a set of fundamental knowledge that is expected. The reader should be familiar with the XDR and RPC protocols as described in [3] and [4]. A basic knowledge of file systems and distributed file systems is expected as well.

This description of version 4.1 features will not distinguish those added in minor version one from those present in the base protocol but will treat minor version 1 as a unified whole. See Section 1.6 for a description of the differences between the two minor versions.
1.4.1. RPC and Security

As with previous versions of NFS, the External Data Representation (XDR) and Remote Procedure Call (RPC) mechanisms used for the NFS version 4.1 protocol are those defined in [3] and [4]. To meet end-to-end security requirements, the RPCSEC_GSS framework [5] 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 [6] to provide one security framework. The LIPKEY and SPKM-3 GSS-API mechanisms described in [7] will be used to provide for the use of user password and client/server public key certificates by the NFS version 4 protocol. With the use of RPCSEC_GSS, other mechanisms may also be specified and used for NFS version 4.1 security.

To enable in-band security negotiation, the NFS version 4.1 protocol has operations which provide the client a method of querying the server about its policies regarding which security mechanisms must be used for access to the server’s file system resources. With this, the client can securely match the security mechanism that meets the policies specified at both the client and server.

1.4.2. Protocol Structure

1.4.2.1. Core Protocol

Unlike NFS Versions 2 and 3, which used a series of ancillary protocols (e.g. NLM, NSM, MOUNT), within all minor versions of NFS version 4 only a single RPC protocol is used to make requests of the server. Facilities that had been separate protocols, such as locking, are now integrated within a single unified protocol.

1.4.2.2. Parallel Access

Minor version one supports high-performance data access to a clustered server implementation by enabling a separation of metadata access and data access, with the latter done to multiple servers in parallel.

Such parallel data access is controlled by recallable objects known as "layouts", which are integrated into the protocol locking model. Clients direct requests for data access to a set of data servers specified by the layout via a data storage protocol which may be NFSv4.1 or may be another protocol.
1.4.3. File System Model

The general file system model used for the NFS version 4.1 protocol is the same as previous versions. The server file system is hierarchical with the regular files contained within being treated as opaque octet streams. In a slight departure, file and directory names are encoded with UTF-8 to deal with the basics of internationalization.

The NFS version 4.1 protocol does not require a separate protocol to provide for the initial mapping between path name and filehandle. All file systems exported by a server are presented as a tree so that all file systems are reachable from a special per-server global root filehandle. This allows LOOKUP operations to be used to perform functions previously provided by the MOUNT protocol. The server provides any necessary pseudo file systems to bridge any gaps that arise due to unexported gaps between exported file systems.

1.4.3.1. Filehandles

As in previous versions of the NFS protocol, opaque filehandles are used to identify individual files and directories. Lookup-type and create operations are used to go from file and directory names to the filehandle which is then used to identify the object to subsequent operations.

The NFS version 4.1 protocol provides support for persistent filehandles, guaranteed to be valid for the lifetime of the file system object designated. In addition it provides support to servers to provide filehandles with more limited validity guarantees, called volatile filehandles.

1.4.3.2. File Attributes

The NFS version 4.1 protocol has a rich and extensible attribute structure. Only a small set of the defined attributes are mandatory and must be provided by all server implementations. The other attributes are known as "recommended" attributes.

One significant recommended file attribute is the Access Control List (ACL) attribute. This attribute provides for directory and file access control beyond the model used in NFS Versions 2 and 3. 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.

One other type of attribute is the named attribute. A named
attribute is an opaque octet 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.

1.4.3.3. Multi-server Namespace

NFS Version 4.1 contains a number of features to allow implementation of namespaces that cross server boundaries and that allow and facilitate a non-disruptive transfer of support for individual file systems between servers. They are all based upon attributes that allow one file system to specify alternate or new locations for that file system.

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

- 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.

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

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

1.4.4. Locking Facilities

As mentioned previously, NFS v4.1, is a single protocol which includes locking facilities. These locking facilities include support for many types of locks including a number of sorts of recallable locks. Recallable locks such as delegations allow the client to be assured that certain events will not occur so long as that lock is held. When circumstances change, the lock is recalled via a callback request. The assurances provided by delegations allow more extensive caching to be done safely when circumstances allow it.

- Share reservations as established by OPEN operations.

- Byte-range locks.
File delegations which are recallable locks that assure the holder that inconsistent opens and file changes cannot occur so long as the delegation is held.

Directory delegations which are recallable delegations that assure the holder that inconsistent directory modifications cannot occur so long as the delegation is held.

Layouts which are recallable objects that assure the holder that direct access to the file data may be performed directly by the client and that no change to the data’s location inconsistent with that access may be made so long as the layout is held.

All locks for a given client are tied together under a single client-wide lease. All requests made on sessions associated with the client renew that lease. When leases are not promptly renewed lock are subject to revocation. In the event of server reinitialization, clients have the opportunity to safely reclaim their locks within a special grace period.

1.5. General Definitions

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

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

A client is uniquely identified by a Client Owner.

In the case of file locking the client is the entity that maintains a set of locks on behalf of one or more applications. This client is responsible for crash or failure recovery for those locks it manages.

Note that multiple clients may share the same transport and 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 client owner. The server is responsible for supplying the client ID.
Client Owner  The client owner is a unique string, opaque to the server, which identifies a client. Multiple network connections and source network addresses originating those connections may share a client owner. The server is expected to treat requests from connections with the same client owner has coming from the same client.

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

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

Lock  The term "lock" is used to refer to any of record (octet-range) locks, share reservations, delegations or layouts unless specifically stated otherwise.

Server  The "Server" is the entity responsible for coordinating client access to a set of file systems. A server can span multiple network addresses. In NFSv4.1, a server is a two tiered entity allows for servers consisting of multiple components the flexibility to tightly or loosely couple their components without requiring tight synchronization among the components. Every server has a "Server Owner" which reflects the two tiers of a server entity.

Server Owner  The "Server Owner" identifies the server to the client. The server owner consists of a major and minor identifier. When the client has two connections each to a peer with the same major and minor identifier, the client assumes both peers are the same server (the server namespace is the same via each connection), and further assumes session and lock state is sharable across both connections. When each peer has the same major identifier but different minor identifier, the client assumes both peers can serve the same namespace, but session and lock state is not sharable across both connections.

Stable Storage  NFS version 4 servers must be able to recover without data loss from multiple power failures (including cascading power failures, that is, several power failures in quick succession), operating system failures, and hardware failure of components other than the storage medium itself (for example, disk, nonvolatile RAM).
Some examples of stable storage that are allowable for an NFS server include:

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

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

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

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

Stateid A 128-bit quantity returned by a server that uniquely defines the open and locking state provided by the server for a specific open or lock owner 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.

1.6. Differences from NFSv4.0

The following summarizes the differences between minor version one and the base protocol:

- Implementation of the sessions model.

- Support for parallel access to data.

- Addition of the RECLAIM_COMPLETE operation to better structure the lock reclamation process.

- Support for delegations on directories and other file types in addition to regular files.

- Operations to re-obtain a delegation.

- Support for client and server implementation id’s.

2. Core Infrastructure
2.1.  Introduction

NFS version 4.1 (NFSv4.1) relies on core infrastructure common to nearly every operation. This core infrastructure is described in the remainder of this section.

2.2.  RPC and XDR

The NFS version 4.1 (NFSv4.1) protocol is a Remote Procedure Call (RPC) application that uses RPC version 2 and the corresponding eXternal Data Representation (XDR) as defined in RFC1831 [4] and RFC4506 [3].

2.2.1.  RPC-based Security

Previous NFS versions have been thought of as having a host-based authentication model, where the NFS server authenticates the NFS client, and trust the client to authenticate all users. Actually, NFS has always depended on RPC for authentication. The first form of RPC authentication which required a host-based authentication approach. NFSv4 also depends on RPC for basic security services, and mandates RPC support for a user-based authentication model. The user-based authentication model has user principals authenticated by a server, and in turn the server authenticated by user principals. RPC provides some basic security services which are used by NFSv4.

2.2.1.1.  RPC Security Flavors

As described in section 7.2 "Authentication" of [4], RPC security is encapsulated in the RPC header, via a security or authentication flavor, and information specific to the specification of the security flavor. Every RPC header conveys information used to identify and authenticate a client and server. As discussed in Section 2.2.1.1.1, some security flavors provide additional security services.

NFSv4 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.

2.2.1.1.1.  RPCSEC_GSS and Security Services

RPCSEC_GSS ([5]) uses the functionality of GSS-API RFC2743 [8]. This allows for the use of various security mechanisms by the RPC layer without the additional implementation overhead of adding RPC security flavors.
2.2.1.1.1.1. Identification, Authentication, Integrity, Privacy

Via the GSS-API, RPCSEC_GSS can be used to identify and authenticate users on clients to servers, and servers to users. It can also perform integrity checking on the entire RPC message, including the RPC header, and the arguments or results. Finally, privacy, usually via encryption, is a service available with RPCSEC_GSS. Privacy is performed on the arguments and results. Note that if privacy is selected, integrity, authentication, and identification are enabled. If privacy is not selected, but integrity is selected, authentication and identification are enabled. If integrity and privacy are not selected, but authentication is enabled, identification is enabled. RPCSEC_GSS does not provide identification as a separate service.

Although GSS-API has an authentication service distinct from its privacy and integrity services, GSS-API’s authentication service is not used for RPCSEC_GSS’s authentication service. Instead, each RPC request and response header is integrity protected with the GSS-API integrity service, and this allows RPCSEC_GSS to offer per-RPC authentication and identity. See [5] for more information.

NFSv4 client and servers MUST support RPCSEC_GSS’s integrity and authentication service. NFSv4.1 servers MUST support RPCSEC_GSS’s privacy service.

2.2.1.1.1.2. Security mechanisms for NFS version 4

RPCSEC_GSS, via GSS-API, normalizes access to mechanisms that provide security services. Therefore NFSv4 clients and servers MUST support three security mechanisms: Kerberos V5, SPKM-3, and LIPKEY.

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 (0) is used, leaving it up to the mechanism or the mechanism’s configuration to use an appropriate level of protection that QOP zero maps to. Each mandated mechanism specifies minimum set of cryptographic algorithms for implementing integrity and privacy. NFSv4 clients and servers MUST be implemented on operating environments that comply with the mandatory cryptographic algorithms of each mandated mechanism.

2.2.1.1.1.2.1. Kerberos V5

The Kerberos V5 GSS-API mechanism as described in RFC1964 [6] ({{Comment.1: need new Kerberos RFC}}) MUST be implemented with the RPCSEC_GSS services as specified in the following table:

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

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>390003</td>
<td>krb5</td>
<td>1.2.840.113554.1.2.2 rpc_gss_svc_none</td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>390004</td>
<td>krb5i</td>
<td>1.2.840.113554.1.2.2 rpc_gss_svc_integrity</td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>390005</td>
<td>krb5p</td>
<td>1.2.840.113554.1.2.2 rpc_gss_svc_privacy</td>
<td>no yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the number and name of the pseudo flavor is presented here as a mapping aid to the implementor. Because the NFSv4 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 the NFS version 3 since the security negotiation is done via the MOUNT protocol as described in [19].

2.2.1.1.1.2.2. LIPKEY

The LIPKEY V5 GSS-API mechanism as described in [7] MUST be implemented with the RPCSEC_GSS services as specified in the following table:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>390006</td>
<td>lipkey</td>
<td>1.3.6.1.5.5.5.9 rpc_gss_svc_none</td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>390007</td>
<td>lipkey-i</td>
<td>1.3.6.1.5.5.5.9 rpc_gss_svc_integrity</td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>390008</td>
<td>lipkey-p</td>
<td>1.3.6.1.5.5.5.9 rpc_gss_svc_privacy</td>
<td>no yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2.1.1.1.2.3. SPKM-3 as a security triple

The SPKM-3 GSS-API mechanism as described in [7] MUST be implemented with the RPCSEC_GSS services as specified in the following table:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>390009</td>
<td>spkm3</td>
<td>1.3.6.1.5.5.5.1.3 rpc_gss_svc_none</td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>390010</td>
<td>spkm3i</td>
<td>1.3.6.1.5.5.5.1.3 rpc_gss_svc_integrity</td>
<td>yes yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>390011</td>
<td>spkm3p</td>
<td>1.3.6.1.5.5.5.1.3 rpc_gss_svc_privacy</td>
<td>no yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2.1.1.1.3. GSS Server Principal

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, LIPKEY, and SPKM-3, the following form is RECOMMENDED:

```
    nfs/hostname
```

2.3. COMPOUND and CB_COMPOUND

A significant departure from the versions of the NFS protocol before version 4 is the introduction of the COMPOUND procedure. For the NFSv4 protocol, in all minor versions, there are exactly two RPC procedures, NULL and COMPOUND. The COMPOUND procedure is defined as a series of individual operations and these operations perform the sorts of functions performed by traditional NFS procedures.

The operations combined within a COMPOUND request are evaluated in order by the server, without any atomicity guarantees. A limited set of facilities exist to pass results from one operation to another. Once an operation returns a failing result, the evaluation ends and the results of all evaluated operations are returned to the client.

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, multi-component lookup requests can be constructed by combining multiple LOOKUP operations. Those can be further combined with operations such as GETATTR, READDIR, or OPEN plus READ to do more complicated sets of operation without incurring additional latency.

NFSv4 also contains a considerable set of callback operations in which the server makes an RPC directed at the client. Callback RPC’s have a similar structure to that of the normal server requests. For the NFS version 4 protocol callbacks in all minor versions, there are two RPC procedures, NULL and CB_COMPOUND. The CB_COMPOUND procedure
is defined in an analogous fashion to that of COMPOUND with its own set of callback operations.

Addition of new server and callback operation within the COMPOUND and CB_COMPOUND request framework provide means of extending the protocol in subsequent minor versions.

Except for a small number of operations needed for session creation, server requests and callback requests are performed within the context of a session. Sessions provide a client context for every request and support robust replay protection for non-idempotent requests.

2.4. Client Identifiers and Client Owners

For each operation that obtains or depends on locking state, the specific client must be determinable by the server. In NFSv4, each distinct client instance is represented by a client ID, which is a 64-bit identifier that identifies a specific client at a given time and which is changed whenever the client or the server re-initializes. Client IDs are used to support lock identification and crash recovery.

In NFSv4.1, during steady state operation, the client ID associated with each operation is derived from the session (see Section 2.10) on which the operation is issued. Each session is associated with a specific client ID at session creation and that client ID then becomes the client ID associated with all requests issued using it. Therefore, unlike NFSv4.0, the only NFSv4.1 operations possible before a client ID is established, are those directly connected with establishing the client ID.

A sequence of an EXCHANGE_ID operation followed by a CREATE_SESSION operation using that client ID (eir_clientid as returned from EXCHANGE_ID) is required to establish the identification on the server. Establishment of identification by a new incarnation of the client also has the effect of immediately releasing any locking state that a previous incarnation of that same client might have had on the server. Such released state would include 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 9.2.1.

Releasing such state requires that the server be able to determine that one client instance is the successor of another. Where this cannot be done, for any of a number of reasons, the locking state will remain for a time subject to lease expiration (see Section 8.5)
and the new client will need to wait for such state to be removed, if it makes conflicting lock requests.

Client identification is encapsulated in the following Client Owner structure:

```c
struct client_owner4 {
    verifier4 co_verifier;
    opaque co_ownerid<NFS4_OPAQUE_LIMIT>;
};
```

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

The second field, `co_ownerid` is a variable length string that uniquely defines the client so that subsequent instances of the same client bear the same `co_ownerid` with a different verifier.

There are several considerations for how the client generates the `co_ownerid` string:

- 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.

- The string should be selected so the subsequent incarnations (e.g. reboots) of the same client cause the client to present the same string. The implementor is cautioned from 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 NFS version 4 server.

- The string should be the same for each server network address that the client accesses, rather than common to all server network addresses (note: the precise opposite was advised in RFC3530). This way, if a server has multiple interfaces, the client can trunk traffic over multiple network paths as described in Section 2.10.3.4.1.

- The algorithm for generating the string should not assume that the client’s network address will not change, unless the client
implementation knows it is using statically assigned network addresses. This includes changes between client incarnations and even changes while the client is still running in its current incarnation. This means that if the client includes just the client's network address in the co_ownerid string, there is a real risk, with dynamic address assignment, that after the client gives up the network address, another client, using a similar algorithm for generating the co_ownerid string, would generate a conflicting co_ownerid string.

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

- If applicable, the client’s statically assigned network address.
- 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 (again, a one way function should be performed).
  - The timestamp of when the NFS version 4 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 NFS version 4).
  - 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.
- For a user level NFS version 4 client, it should contain additional information to distinguish the client from other user level clients running on the same host, such as a process identifier or other unique sequence.

As a security measure, the server MUST NOT cancel a client’s leased state if the principal established the state for a given co_ownerid string is not the same as the principal issuing the EXCHANGE_ID.

A server may compare an client_owner4 in a EXCHANGE_ID with an nfs_client_id4 established using SETCLIENTID using NFSv4 minor version 0, so that an NFSv4.1 client is not forced to delay until lease expiration for locking state established by the earlier client using minor version 0. This requires the client_owner4 be
constructed the same way as the nfs_client_id4. If the latter’s contents included the server’s network address, and the NFSv4.1 client does not wish to use a client ID that prevents trunking, it should issue two EXCHANGE_ID operations. The first EXCHANGE_ID will have a client_owner4 equal to the nfs_client_id4. This will clear the state created by the NFSv4.0 client. The second EXCHANGE_ID will not have the server’s network address. The state created for the second EXCHANGE_ID will not have to wait for lease expiration, because there will be no state to expire.

Once an EXCHANGE_ID has been done, and the resulting client ID established as associated with a session, all requests made on that session implicitly identify that client ID, which in turn designates the client specified using the long-form client_owner4 structure. The shorthand client identifier (a client ID) is assigned by the server (the eir_clientid result from EXCHANGE_ID) 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.

In the event of a server restart, a client may find out that its current client ID is no longer valid when receives a NFS4ERR_STALE_CLIENTID error. The precise circumstances depend on the characteristics of the sessions involved, specifically whether the session is persistent (see Section 2.10.4.5).

When a session is not persistent, the client will need to create a new session. When the existing client ID is presented to a server as part of creating a session 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 EXCHANGE_ID operation and then use that client ID as the basis of the basis of a new session and then proceed to any other necessary recovery for the server reboot case (See Section 8.6.2).

In the case of the session being persistent, the client will re-establish communication using the existing session after the reboot. This session will be associated with a client ID that has had state revoked (but the persistent session is never associated with a stale client ID, because if the session is persistent, the client ID MUST persist), and the client will receive an indication of that fact in the sr_status_flags field returned by the SEQUENCE operation (see Section 17.46.4). The client can then use the existing session to do whatever operations are necessary to determine the status of requests outstanding at the time of reboot, while avoiding issuing new requests, particularly any involving locking on that session. Such requests would fail with an NFS4ERR_STALE_STATEID error, if
See the detailed descriptions of EXCHANGE_ID (Section 17.35 and CREATE_SESSION (Section 17.36) for a complete specification of these operations.

2.4.1. Server Release of Client ID

NFSv4.1 introduces a new operation called DESTROY_CLIENTID (Section 17.50) which the client SHOULD use to destroy a client ID it no longer needs. This permits graceful, bilateral release of a client ID.

If the server determines that the client holds no associated state for its client ID (including sessions, opens, locks, delegations, layouts, and wants), the server may choose to unilaterally 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 EXCHANGE_ID/CREATE_SESSION 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. As long as there are sessions, opens, locks, delegations, layouts, or wants, the server MUST not release the client ID. See Section 2.10.8.1.4 for discussion on releasing inactive sessions.

2.4.2. Handling Client Owner Conflicts

If the co_ownerid string in a EXCHANGE_ID request is properly constructed, and if the client takes care to use the same principal for each successive use of EXCHANGE_ID, then, barring an active denial of service attack, conflicts are not possible.

However, client bugs, server bugs, or perhaps a deliberate change of the principal owner of the co_ownerid 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 a conflict.

When the server gets a EXCHANGE_ID for a client owner that currently has no state, or if it has state, but the lease has expired, server MUST allow the EXCHANGE_ID, and confirm the new client ID if followed by the appropriate CREATE_SESSION.
When the server gets a EXCHANGE_ID for a client owner that currently has state, or an unexpired lease, and the principal that issues the
EXCHANGE_ID is different than principal the previously established
client owner, the server MUST not destroy the any state that currently exists for client owner. Regardless, the server has two choices. First, it can return NFS4ERR_CLID_INUSE. Second, it can allow the EXCHANGE_ID, and simply treat the client owner as consisting of both the co_ownerid and the principal that issued the
EXCHANGE_ID.

2.5. Server Owners

The Server Owner is somewhat similar to a Client Owner (Section 2.4), but unlike the Client Owner, there is no shorthand serverid. The Server Owner is defined in the following structure:

```
struct server_owner4 {
    uint64_t        so_minor_id;
    opaque          so_major_id<NFS4_OPAQUE_LIMIT>;
};
```

The Server Owner is returned in the results of EXCHANGE_ID. When the so_major_id fields are the same in two EXCHANGE_ID results, the connections each EXCHANGE_ID are sent over can be assumed to address the same Server (as defined in Section 1.5). If the so_minor_id fields are also the same, then not only do both connections connect to the same server, but the session and other state can be shared across both connections. The reader is cautioned that multiple servers may deliberately or accidentally claim to have the same so_major_id or so_major_id/so_minor_id; the reader should examine Section 2.10.3.4.1 and Section 17.35.

The considerations for generating an so_major_id are similar to that for generating a co_ownerid string (see Section 2.4). The consequences of two servers generating conflict so_major_id values are less dire than they are for co_ownerid conflicts because the client can use RPCSEC_GSS to compare the authenticity of each server (see Section 2.10.3.4.1).

2.6. Security Service Negotiation

With the NFS version 4 server potentially offering multiple security mechanisms, the client needs a method to determine or negotiate which mechanism is to be used for its communication with the server. The NFS server may have multiple points within its file system namespace that are available for use by NFS clients. These points can be considered security policy boundaries, and in some NFS
implementations are tied to NFS export points. In turn the NFS server may be configured such that each of these security policy boundaries may have different or multiple security mechanisms in use.

The security negotiation between client and server must be done with a secure channel to eliminate the possibility of a third party intercepting the negotiation sequence and forcing the client and server to choose a lower level of security than required or desired. See Section 20 for further discussion.

2.6.1. NFSv4 Security Tuples

An NFS server can assign one or more "security tuples" to each security policy boundary in its namespace. Each security tuple consists of a security flavor (see Section 2.2.1.1), and if the flavor is RPCSEC_GSS, a GSS-API mechanism OID, a GSS-API quality of protection, and an RPCSEC_GSS service.

2.6.2. SECINFO and SECINFO_NO_NAME

The SECINFO and SECINFO_NO_NAME operations allow the client to determine, on a per filehandle basis, what security tuple is to be used for server access. In general, the client will not have to use either operation except during initial communication with the server or when the client crosses security policy boundaries at the server. It is possible that the server’s policies change during the client’s interaction therefore forcing the client to negotiate a new security tuple.

Where the use of different security tuples would affect the type of access that would be allowed if a request was issued over the same connection used for the SECINFO or SECINFO_NO_NAME operation (e.g. read-only vs. read-write) access, security tuples that allow greater access should be presented first. Where the general level of access is the same and different security flavors limit the range of principals whose privileges are recognized (e.g. allowing or disallowing root access), flavors supporting the greatest range of principals should be listed first.

2.6.3. Security Error

Based on the assumption that each NFS version 4 client and server must support a minimum set of security (i.e., LIPKEY, SPKM-3, and Kerberos-V5 all under RPCSEC_GSS), the NFS client will initiate file access to the server with one of the minimal security tuples. During communication with the server, the client may receive an NFS error of NFS4ERR_WRONGSEC. This error allows the server to notify the client that the security tuple currently being used is contravenes the
server’s security policy. The client is then responsible for
determining (see Section 2.6.3.1) what security tuples are available
at the server and choosing one which is appropriate for the client.

2.6.3.1. Using NFS4ERR_WRONGSEC, SECINFO, and SECINFO_NO_NAME

This section explains of the mechanics of NFSv4.1 security
negotiation. The term "put filehandle operation" refers to
PUTROOTFH, PUTPUBFH, PUTFH, and RESTOREFH.

2.6.3.1.1. Put Filehandle Operation + SAVEFH

The client is saving a filehandle for a future RESTOREFH. The server
MUST NOT return NFS4ERR_WRONG to either the put filehandle operation
or SAVEFH.

2.6.3.1.2. Two or More Put Filehandle Operations

For a series of N put filehandle operations, the server MUST NOT
return NFS4ERR_WRONGSEC to the first N-1 put filehandle operations.
The Nth put filehandle operation is handled as if it is the first in
a series of operations, and the second in the series of operations is
not a put filehandle operation. For example if the server received
PUTFH, PUTROOTFH, LOOKUP, then the PUTFH is ignored for
NFS4ERR_WRONGSEC purposes, and the PUTROOTFH, LOOKUP subseries is
processed as according to Section 2.6.3.1.3.

2.6.3.1.3. Put Filehandle Operation + LOOKUP (or OPEN by Name)

This situation also applies to a put filehandle operation followed by
a LOOKUP or an OPEN operation that specifies a component name.

In this situation, the client is potentially crossing a security
policy boundary, and the set of security tuples the parent directory
supports differ from those of the child. The server implementation
may decide whether to impose any restrictions on security policy
administration. There are at least three approaches
(sec_policy_child is the tuple set of the child export,
sec_policy_parent is that of the parent).

a) sec_policy_child <= sec_policy_parent (<= for subset). This
means that the set of security tuples specified on the security
policy of a child directory is always a subset of that of its
parent directory.
b)  \text{sec\_policy\_child} \cap \text{sec\_policy\_parent} \neq \{\} (\cap \text{for intersection,}
\{\} \text{for the empty set}).  \text{This means that the security tuples}
specified on the security policy of a child directory always has a
non empty intersection with that of the parent.

c)  \text{sec\_policy\_child} \cap \text{sec\_policy\_parent} = \{\}. \text{This means that}
the set of tuples specified on the security policy of a child
directory may not intersect with that of the parent.  \text{In other}
words, there are no restrictions on how the system administrator
may set up these tuples.

For a server to support approach (b) (when client chooses a flavor
that is not a member of sec\_policy\_parent) and (c), the put
filehandle operation must NOT return NFS4ERR_WRONGSEC in case of
security mismatch.  Instead, it should be returned from the LOOKUP
(or OPEN by component name) that follows.

Since the above guideline does not contradict approach (a), it should
be followed in general.  Even if approach (a) is implemented, it is
possible for the security tuple used to be acceptable for the target
of LOOKUP but not for the filehandles used in the put filehandle
operation.  The put filehandle operation could be a PUTROOTFH or
PUTPUBFH, where the client cannot know the security tuples for the
root or public filehandle.  Or the security policy for the filehandle
used by the put filehandle operation could have changed since the
time the filehandle was obtained.

Therefore, an NFSv4.1 server MUST NOT return NFS4ERR_WRONGSEC in
response to the put filehandle operation if the operation is
immediately followed by a LOOKUP or an OPEN by component name.

2.6.3.1.4.  Put Filehandle Operation + LOOKUPP

Since SECINFO only works its way down, there is no way LOOKUPP can
return NFS4ERR_WRONGSEC without SECINFO\_NO\_NAME.  SECINFO\_NO\_NAME
solves this issue because via style SECINFO\_STYLE4\_PARENT, it works
in the opposite direction as SECINFO.  As with Section 2.6.3.1.3, the
put filehandle operation must not return NFS4ERR_WRONGSEC whenever it
is followed by LOOKUPP.  If the server does not support
SECINFO\_NO\_NAME, the client’s only recourse is to issue the put
filehandle operation, LOOKUPP, GETFH sequence of operations with
every security tuple it supports.

Regardless whether SECINFO\_NO\_NAME is supported, an NFSv4.1 server
MUST NOT return NFS4ERR_WRONGSEC in response to a put filehandle
operation if the operation is immediately followed by a LOOKUPP.
2.6.3.1.5. Put Filehandle Operation + SECINFO/SECINFO_NO_NAME

A security sensitive client is allowed to choose a strong security tuple when querying a server to determine a file object’s permitted security tuples. The security tuple chosen by the client does not have to be included in the tuple list of the security policy of the either parent directory indicated in the put filehandle operation, or the child file object indicated in SECINFO (or any parent directory indicated in SECINFO_NO_NAME). Of course the server has to be configured for whatever security tuple the client selects, otherwise the request will fail at RPC layer with an appropriate authentication error.

In theory, there is no connection between the security flavor used by SECINFO or SECINFO_NO_NAME and those supported by the security policy. But in practice, the client may start looking for strong flavors from those supported by the security policy, followed by those in the mandatory set.

The NFSv4.1 server MUST NOT return NFS4ERR_WRONGSEC to a put filehandle operation whenever it is immediately followed by SECINFO or SECINFO_NO_NAME. The NFSv4.1 server MUST NOT return NFS4ERR_WRONGSEC from SECINFO or SECINFO_NO_NAME.

2.6.3.1.6. Put Filehandle Operation + Nothing

The NFSv4.1 server MUST NOT return NFS4ERR_WRONGSEC.

2.6.3.1.7. Put Filehandle Operation + Anything Else

"Anything Else" includes OPEN by filehandle.

The security policy enforcement applies to the filehandle specified in the put filehandle operation. Therefore PUTFH must return NFS4ERR_WRONGSEC in case of security tuple on the part of the mismatch. This avoids the complexity adding NFS4ERR_WRONGSEC as an allowable error to every other operation.

A COMPOUND containing the series put filehandle operation + SECINFO_NO_NAME (style SECINFO_STYLE4_CURRENT_FH) is an efficient way for the client to recover from NFS4ERR_WRONGSEC.

The NFSv4.1 server MUST not return NFS4ERR_WRONGSEC to any operation other than a put filehandle operation, LOOKUP, LOOKUPP, and OPEN (by component name).
2.7. Minor Versioning

To address the requirement of an NFS protocol that can evolve as the need arises, the NFS version 4 protocol contains the rules and framework to allow for future minor changes or versioning.

The base assumption with respect to minor versioning is that any future accepted minor version must follow the IETF process and be documented in a standards track RFC. Therefore, each minor version number will correspond to an RFC. Minor version zero of the NFS version 4 protocol is represented by [2], and minor version one is represented by this document. The COMPOUND and CB_COMPOUND procedures support the encoding of the minor version being requested by the client.

The following items represent the basic rules for the development of minor versions. Note that a future minor version may decide to modify or add to the following rules as part of the minor version definition.

1. Procedures are not added or deleted

To maintain the general RPC model, NFS version 4 minor versions will not add to or delete procedures from the NFS program.

2. Minor versions may add operations to the COMPOUND and CB_COMPOUND procedures.

The addition of operations to the COMPOUND and CB_COMPOUND procedures does not affect the RPC model.

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

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

* Minor version X must append any new attributes after the last documented attribute.

Since attribute results are specified as an opaque array of per-attribute XDR encoded results, the complexity of adding new attributes in the midst of the current definitions would be too burdensome.

3. Minor versions must not modify the structure of an existing operation’s arguments or results.
Again the complexity of handling multiple structure definitions for a single operation is too burdensome. New operations should be added instead of modifying existing structures for a minor version.

This rule does not preclude the following adaptations in a minor version.

* adding bits to flag fields such as new attributes to GETATTR’s bitmap4 data type
* adding bits to existing attributes like ACLs that have flag words
* extending enumerated types (including NFS4ERR_*) with new values

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

5. Minor versions may not delete operations.

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

6. Minor versions may not delete attributes.

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

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

   Specifying an operation as "mandatory to not implement" is equivalent to obsoleting an operation. For the client, it means that the operation should not be sent to the server. For the server, an NFS error can be returned as opposed to "dropping" the request as an XDR decode error. This approach allows for the obsolescence of an operation while maintaining its structure so that a future minor version can reintroduce the operation.

1. Minor versions may declare attributes mandatory to NOT implement.

2. Minor versions may declare flag bits or enumeration values as mandatory to NOT implement.
9. Minor versions may downgrade features from mandatory to recommended, or recommended to optional.

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

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

12. Except for infrastructural changes, no new features may be introduced as mandatory in a minor version.

   This rule allows for the introduction of new functionality and forces the use of implementation experience before designating a feature as mandatory. On the other hand, some classes of features are infrastructural and have broad effects. Allowing such features to not be mandatory complicates implementation of the minor version.

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

2.8. Non-RPC-based Security Services

As described in Section 2.2.1.1.1.1, NFSv4 relies on RPC for identification, authentication, integrity, and privacy. NFSv4 itself provides additional security services as described in the next several subsections.

2.8.1. Authorization

Authorization to access a file object via an NFSv4 operation is ultimately determined by the NFSv4 server. A client can predetermine its access to a file object via the OPEN (Section 17.16) and the ACCESS (Section 17.1) operations.

Principals with appropriate access rights can modify the authorization on a file object via the SETATTR (Section 17.30) operation. Four attributes that affect access rights are: mode, owner, owner_group, and acl. See Section 5.

2.8.2. Auditing

NFSv4 provides auditing on a per file object basis, via the ACL attribute as described in Section 6. It is outside the scope of this specification to specify audit log formats or management policies.
2.8.3. Intrusion Detection

NFSv4 provides alarm control on a per file object basis, via the ACL attribute as described in Section 6. Alarms may serve as the basis for intrusion detection. It is outside the scope of this specification to specify heuristics for detecting intrusion via alarms.

2.9. Transport Layers

2.9.1. Required and Recommended Properties of Transports

NFSv4 works over RDMA and non-RDMA-based transports with the following attributes:

- The transport supports reliable delivery of data, which NFSv4 requires but neither NFSv4 nor RPC has facilities for ensuring. [20]
- The transport delivers data in the order it was sent. Ordered delivery simplifies detection of transmit errors, and simplifies the sending of arbitrary sized requests and responses, via the record marking protocol [4].

Where an NFS version 4 implementation supports operation over the IP network protocol, any transport used between NFS and IP MUST be among the IETF-approved congestion control transport protocols. At the time this document was written, the only two transports that had the above attributes were TCP and SCTP. To enhance the possibilities for interoperability, an NFS version 4 implementation MUST support operation over the TCP transport protocol.

Even if NFS version 4 is used over a non-IP network protocol, it is RECOMMENDED that the transport support congestion control.

It is permissible for a connectionless transport to be used under NFSv4.1, however reliable and in-order delivery of data by the connectionless transport is still required. NFSv4.1 assumes that a client transport address and server transport address used to send data over a transport together constitute a connection, even if the underlying transport eschews the concept of a connection.

2.9.2. Client and Server Transport Behavior

If a connection-oriented transport (e.g. TCP) is used the client and server SHOULD use long lived connections for at least three reasons:
1. This will prevent the weakening of the transport’s congestion control mechanisms via short lived connections.

2. This will improve performance for the WAN environment by eliminating the need for connection setup handshakes.

3. The NFSv4.1 callback model differs from NFSv4.0, and requires the client and server to maintain a client-created channel (see Section 2.10.3.4 for the server to use.

In order to reduce congestion, if a connection-oriented transport is used, and the request is not the NULL procedure,

- A requester MUST NOT retry a request unless the connection the request was issued over was disconnected before the reply was received.

- A replier MUST NOT silently drop a request, even if the request is a retry. (The silent drop behavior of RPCSEC_GSS [5] does not apply because this behavior happens at the RPCSEC_GSS layer, a lower layer in the request processing). Instead, the replier SHOULD return an appropriate error (see Section 2.10.4.1) or it MAY disconnect the connection.

When using RDMA transports there are other reasons for not tolerating retries over the same connection:

- RDMA transports use "credits" to enforce flow control, where a credit is a right to a peer to transmit a message. If one peer were to retransmit a request (or reply), it would consume an additional credit. If the replier retransmitted a reply, it would certainly result in an RDMA connection loss, since the requester would typically only post a single receive buffer for each request. If the requester retransmitted a request, the additional credit consumed on the server might lead to RDMA connection failure unless the client accounted for it and decreased its available credit, leading to wasted resources.

- RDMA credits present a new issue to the reply cache in NFSv4.1. The reply cache may be used when a connection within a session is lost, such as after the client reconnects. Credit information is a dynamic property of the RDMA connection, and stale values must not be replayed from the cache. This implies that the reply cache contents must not be blindly used when replies are issued from it, and credit information appropriate to the channel must be refreshed by the RPC layer.

In addition, the NFSv4.1 requester is not allowed to stop waiting for
a reply, as described in Section 2.10.4.2.

2.9.3. Ports

Historically, NFS version 2 and version 3 servers have resided on port 2049. The registered port 2049 for the NFS protocol should be the default configuration. NFSv4 clients SHOULD NOT use the RPC binding protocols as described in RFC1833 [22].

2.10. Session

2.10.1. Motivation and Overview

Previous versions and minor versions of NFS have suffered from the following:

- Lack of support for exactly once semantics (EOS). This includes lack of support for EOS through server failure and recovery.
- Limited callback support, including no support for sending callbacks through firewalls, and races between responses from normal requests, and callbacks.
- Limited trunking over multiple network paths.
- Requiring machine credentials for fully secure operation.

Through the introduction of a session, NFSv4.1 addresses the above shortfalls with practical solutions:

- EOS is enabled by a reply cache with a bounded size, making it feasible to keep on persistent storage and enable EOS through server failure and recovery. One reason that previous revisions of NFS did not support EOS was because some EOS approaches often limited parallelism. As will be explained in Section 2.10.4), NFSv4.1 supports both EOS and unlimited parallelism.
- The NFSv4.1 client provides creates transport connections and gives them to the server for sending callbacks, thus solving the firewall issue (Section 17.34). Races between responses from client requests, and callbacks caused by the requests are detected via the session’s sequencing properties which are a byproduct of EOS (Section 2.10.4.3).
- The NFSv4.1 client can add an arbitrary number of connections to the session, and thus provide trunking (Section 2.10.3.4.1).
The NFSv4.1 session produces a session key independent of client and server machine credentials which can be used to compute a digest for protecting key session management operations (Section 2.10.6.3).

The NFSv4.1 client can also create secure RPCSEC_GSS contexts for use by the session’s callback channel that do not require the server to authenticate to a client machine principal (Section 2.10.6.2).

A session is a dynamically created, long-lived server object created by a client, used over time from one or more transport connections. Its function is to maintain the server’s state relative to the connection(s) belonging to a client instance. This state is entirely independent of the connection itself, and indeed the state exists whether the connection exists or not (though locks, delegations, etc. and generally expire in the extended absence of an open connection). The session in effect becomes the object representing an active client on a set of zero or more connections.

### 2.10.2. NFSv4 Integration

Sessions are part of NFSv4.1 and not NFSv4.0. Normally, a major infrastructure change like sessions would require a new major version number to an RPC program like NFS. However, because NFSv4 encapsulates its functionality in a single procedure, COMPOUND, and because COMPOUND can support an arbitrary number of operations, sessions are almost trivially added. COMPOUND includes a minor version number field, and for NFSv4.1 this minor version is set to 1. When the NFSv4 server processes a COMPOUND with the minor version set to 1, it expects a different set of operations than it does for NFSv4.0. One operation it expects is the SEQUENCE operation, which is required for every COMPOUND that operates over an established session.

#### 2.10.2.1. SEQUENCE and CB_SEQUENCE

In NFSv4.1, when the SEQUENCE operation is present, it is always the first operation in the COMPOUND procedure. The primary purpose of SEQUENCE is to carry the session identifier. The session identifier associates all other operations in the COMPOUND procedure with a particular session. SEQUENCE also contains required information for maintaining EOS (see Section 2.10.4). Session-enabled NFSv4.1 COMPOUND requests thus have the form:
A CB_COMPOUND procedure request and reply has a similar form, but instead of a SEQUENCE operation, there is a CB_SEQUENCE operation, and there is an additional field called "callback_ident", which is superfluous in NFSv4.1. CB_SEQUENCE has the same information as SEQUENCE, but includes other information needed to solve callback races (Section 2.10.4.3).

2.10.2.2. Client ID and Session Association

Sessions are subordinate to the client ID (Section 2.4). Each client ID can have zero or more active sessions. A client ID, and a session bound to it are required to do anything useful in NFSv4.1. Each time a session is used, the state leased to its associated client ID is automatically renewed.

State such as share reservations, locks, delegations, and layouts (Section 1.4.4) is tied to the client ID, not the sessions of the client ID. Successive state changing operations from a given state owner can go over different sessions, as long each session is associated with the same client ID. Callbacks can arrive over a different session than the session that sent the operation that acquired the state that the callback is for. For example, if session A is used to acquire a delegation, a request to recall the delegation can arrive over session B.

2.10.3. Channels

Each session has one or two channels: the "operation" or "fore" channel used for ordinary requests from client to server, and the "back" channel, used for callback requests from server to client. The session allocates resources for each channel, including separate reply caches (see Section 2.10.4.1). These resources are for the most part specified at time the session is created.
2.10.3.1. Operation Channel

The operation channel carries COMPOUND requests and responses. A session always has an operation channel.

2.10.3.2. Backchannel

The backchannel carries CB_COMPOUND requests and responses. Whether there is a backchannel or not is a decision of the client; NFSv4.1 servers MUST support backchannels.

2.10.3.3. Session and Channel Association

Because there are at most two channels per session, and because each channel has a distinct purpose, channels are not assigned identifiers. The operation and backchannel are implicitly created and associated when the session is created.

2.10.3.4. Connection and Channel Association

Each channel is associated with zero or more transport connections. A connection can be bound to one channel or both channels of a session; the client and server negotiate whether a connection will carry traffic for one channel or both channels via the CREATE_SESSION (Section 17.36) and the BIND_CONN_TO_SESSION (Section 17.34) operations. When a session is created via CREATE_SESSION, it is automatically bound to the operation channel, and optionally the backchannel. If the client does not specify connecting binding enforcement when the session is created, then additional connections are automatically bound to the operation channel when the are used with a SEQUENCE operation that has the session’s sessionid.

A connection MAY be bound to the channels of other sessions. The client decides, and the NFSv4.1 server MUST allow it. A connection MAY be bound to the channels of other sessions of other clientids. Again, the client decides, and the server MUST allow it.

It is permissible for connections of multiple types to be bound to the same channel. For example a TCP and RDMA connection can be bound to the operation channel. In the event an RDMA and non-RDMA connection are bound to the same channel, the maximum number of slots must be at least one more than the total number of credits. This way if all RDMA credits are use, the non-RDMA connection can have at least one outstanding request.

It is permissible for a connection of one type to be bound to the operation channel, and another type bound to the backchannel.
2.10.3.4.1. Trunking

A client is allowed to issue EXCHANGE_ID multiple times to the same server. The client may be unaware that two different server network addresses refer to the same server. The use of EXCHANGE_ID allows a client to become aware that an additional network address refers to a server the client already has an established client ID and session for. The eir_server_owner and eir_server_scope results from EXCHANGE_ID give a client a hint that the server it is connected to may be the same as the server it is connected to via another connection. When EXCHANGE_ID is issued over two different connections, and each return the same eir_server_owner.so_major_id and eir_server_scope, the client treats the connections as connected to the same server (subject to verification, as described later in this section (Paragraph 2), even if the destination network addresses are different). As long two unrelated servers have not selected and returned a conflicting pair of eir_major_id and eir_server_scope, or unless the client has used different co_ownerid values in each EXCHANGE_ID request, or the server has lost client ID state (e.g. the server has rebooted) the server MUST return the same eir_clientid result. Otherwise, the client and server use the common eir_clientid to identify the client. The eir_server_owner.so_minor_id field allows the server to control binding of connections to sessions. When two connections have a matching eir_server_scope, so_major_id and so_minor_id, the client may bind both connections to a common session; this is session trunking. When two connections have a matching so_major_id and eir_server_scope, but different so_minor_id, the client will need to create a new session for the client ID in order to use the connection; this is client ID trunking. In either session or client ID trunking, the bandwidth capacity can scale with the number of connections.

When two servers over two connections claim matching or partially matching eir_server_owner, eir_server_scope, and eir_clientid values the client does not have to trust the servers’ claims. The client may verify these claims before trunking traffic in the following ways:

- For session trunking, clients and servers can reliably verify if connections between different network paths are in fact bound to the same NFSv4.1 server and usable on the same session. The SET_SSV (Section 17.47) operation allows a client and server to establish a unique, shared key value (the SSV). When a new connection is bound to the session (via the BIND_CONN_TO_SESSION operation, see Section 17.34), the client offers a digest that is based on the SSV. If the client mistakenly tries to bind a connection to a session of a wrong server, the server will either reject the attempt because it is not aware of the session.
identifier of the BIND_CONN_TO_SESSION arguments, or it will reject the attempt because the digest for the SSV does not match what the server expects. Even if the server mistakenly or maliciously accepts the connection bind attempt, the digest it computes in the response will not be verified by the client, the client will know it cannot use the connection for trunking the specified channel.

- In the case of client ID trunking, the client can use RPCSEC_GSS to verify that each connection is aimed at the same server. When the client invokes EXCHANGE_ID, it should use RPCSEC_GSS. If each RPCSEC_GSS context over each connection has the same server principal, then -- barring a compromise of the server’s GSS credentials -- the servers at the end of each connection are the same.

### 2.10.4. Exactly Once Semantics

Via the session, NFSv4.1 offers exactly once semantics (EOS) for requests sent over a channel. EOS is supported on both the operation and back channels.

Each COMPOUND or CB_COMPOUND request that is issued with a leading SEQUENCE or CB_SEQUENCE operation MUST be executed by the receiver exactly once. This requirement is regardless whether the request is issued with reply caching specified (see Section 2.10.4.1.2). The requirement holds even if the requester is issuing the request over a session created between a pNFS data client and pNFS data server. The rationale for this requirement is understood by categorizing requests into three classifications:

- Nonidempotent requests.
- Idempotent modifying requests.
- Idempotent non-modifying requests.

An example of a non-idempotent request is RENAME. If is obvious that if a replier executes the same RENAME request twice, and the first execution succeeds, the re-execution will fail. If the replier returns the result from the re-execution, this result is incorrect. Therefore, EOS is required for nonidempotent requests.

An example of an idempotent modifying request is a COMPOUND request containing a WRITE operation. Repeated execution of the same WRITE has the same effect as execution of that write once. Nevertheless, putting enforcing EOS for WRITEs and other idempotent modifying requests is necessary to avoid data corruption.
Suppose a client issues WRITEs A, B, C to a noncompliant server that
does not enforce EOS, and receives no response, perhaps due to a
network partition. The client reconnects to the server and re-issues
all three WRITEs. Now, the server has outstanding two instances of
each of A, B, and C. The server can be in a situation in which it
executes and replies to the retries of A, B, and C while the first A,
B, and C are still waiting around in the server’s I/O system for some
resource. Upon receiving the replies to the second attempts of
WRITEs A, B, and C, the client believes its writes are done so it is
free to do issue WRITE D which overlaps the range of one or more of
A, B, C. If any of A, B, or C are subsequently executed for the
second time, then what has been written by D can be overwritten and
thus corrupted.

Note that it is not required the server cache the reply to the
modifying operation to avoid data corruption (but if the client
specified the reply to be cached, the server must cache it).

An example of an idempotent non-modifying request is a COMPOUND
containing SEQUENCE, PUTFH, READLINK and nothing else. The re-
execution of a such a request will not cause data corruption, or
produce an incorrect result. Nonetheless, for simplicity, the
replier MUST enforce EOS for such requests.

2.10.4.1. Slot Identifiers and Reply Cache

The RPC layer provides a transaction ID (xid), which, while required
to be unique, is not especially convenient for tracking requests.
The xid is only meaningful to the requester it cannot be interpreted
at the replier except to test for equality with previously issued
requests. Because RPC operations may be completed by the replier in
any order, many transaction IDs may be outstanding at any time. The
requester may therefore perform a computationally expensive lookup
operation in the process of demultiplexing each reply.

In the NFSv4.1, there is a limit to the number of active requests.
This immediately enables a computationally efficient index for each
request which is designated as a Slot Identifier, or slotid.

When the requester issues a new request, it selects a slotid in the
range 0..N-1, where N is the replier’s current "outstanding requests"
limit granted to the requester on the session over which the request
is to be issued. The value of N outstanding requests starts out as
the value of ca_maxrequests (Section 17.36), but can be adjusted by
the response to SEQUENCE or CB_SEQUENCE as described later in this
section. The slotid must be unused by any of the requests which the
requester has already active on the session. "Unused" here means the
requester has no outstanding request for that slotid. Because the
slot id is always an integer in the range 0..N-1, requester implementations can use the slot id from a replier response to efficiently match responses with outstanding requests, such as, for example, by using the slot id to index into an outstanding request array. This can be used to avoid expensive hashing and lookup functions in the performance-critical receive path.

The sequenceid, which accompanies the slot id in each request, is for an important check at the server: it must be able to be determined efficiently whether a request using a certain slot id is a retransmit or a new, never-before-seen request. It is not feasible for the client to assert that it is retransmitting to implement this, because for any given request the client cannot know the server has seen it unless the server actually replies. Of course, if the client has seen the server’s reply, the client would not retransmit.

The sequenceid MUST increase monotonically for each new transmit of a given slot id, and MUST remain unchanged for any retransmission. The server must in turn compare each newly received request’s sequenceid with the last one previously received for that slot id, to see if the new request is:

- A new request, in which the sequenceid is one greater than that previously seen in the slot (accounting for sequence wraparound). The replier proceeds to execute the new request.

- A retransmitted request, in which the sequenceid is equal to that last seen in the slot. Note that this request may be either complete, or in progress. The replier performs replay processing in these cases.

- A misordered replay, in which the sequenceid is less than (accounting for sequence wraparound) than that previously seen in the slot. The replier MUST return NFS4ERR_SEQ_MISORDERED (as the result from SEQUENCE or CB_SEQUENCE).

- A misordered new request, in which the sequenceid is two or more than (accounting for sequence wraparound) than that previously seen in the slot. Note that because the sequenceid must wraparound one it reaches 0xFFFFFFFF, a misordered new request and a misordered replay cannot be distinguished. Thus, the replier MUST return NFS4ERR_SEQ_MISORDERED (as the result from SEQUENCE or CB_SEQUENCE).

Unlike the XID, the slot id is always within a specific range; this has two implications. The first implication is that for a given session, the replier need only cache the results of a limited number of COMPOUND requests. The second implication derives from the first,
which is unlike XID-indexed reply caches (also know as duplicate request caches - DRCs), the slotid-based reply cache cannot be overflowed. Through use of the sequenceid to identify retransmitted requests, the replier does not need to actually cache the request itself, reducing the storage requirements of the reply cache further. These new facilities makes it practical to maintain all the required entries for an effective reply cache.

The slotid and sequenceid therefore take over the traditional role of the XID and port number in the replier reply cache implementation, and the session replaces the IP address. This approach is considerably more portable and completely robust - it is not subject to the frequent reassignment of ports as clients reconnect over IP networks. In addition, the RPC XID is not used in the reply cache, enhancing robustness of the cache in the face of any rapid reuse of XIDs by the client. [[Comment.3: We need to discuss the requirements of the client for changing the XID.]]

The slotid information is included in each request, without violating the minor versioning rules of the NFSv4.0 specification, by encoding it in the SEQUENCE operation within each NFSv4.1 COMPOUND and CB_COMPOUND procedure. The operation easily piggybacks within existing messages. [[Comment.4: Need a better term than piggyback]]

The receipt of a new sequenced request arriving on any valid slot is an indication that the previous reply cache contents of that slot may be discarded.

The SEQUENCE (and CB_SEQUENCE) operation also carries a "highest_slotid" value which carries additional client slot usage information. The requester must always provide a slotid representing the outstanding request with the highest-numbered slot value. The requester should in all cases provide the most conservative value possible, although it can be increased somewhat above the actual instantaneous usage to maintain some minimum or optimal level. This provides a way for the requester to yield unused request slots back to the replier, which in turn can use the information to reallocate resources.

The replier responds with both a new target highest_slotid, and an enforced highest_slotid, described as follows:

o The target highest_slotid is an indication to the requester of the highest_slotid the replier wishes the requester to be using. This permits the replier to withdraw (or add) resources from a requester that has been found to not be using them, in order to more fairly share resources among a varying level of demand from other requesters. The requester must always comply with the
replier’s value updates, since they indicate newly established hard limits on the requester’s access to session resources. However, because of request pipelining, the requester may have active requests in flight reflecting prior values, therefore the replier must not immediately require the requester to comply.

- The enforced highest_slotid indicates the highest slotid the requester is permitted to use on a subsequent SEQUENCE or CB_SEQUENCE operation.

The requester is required to use the lowest available slot when issuing a new request. This way, the replier may be able to retire slot entries faster. However, where the replier is actively adjusting its granted maximum request count (i.e. the highest_slotid) to the requester, it will not be able to use just the receipt of the slotid and highest_slotid in the request. Neither the slotid nor the highest_slotid used in a request may reflect the replier’s current idea of the requester’s session limit, because the request may have been sent from the requester before the update was received. Therefore, in the downward adjustment case, the replier may have to retain a number of reply cache entries at least as large as the old value of maximum requests outstanding, until operation sequencing rules allow it to infer that the requester has seen its reply.

2.10.4.1.1. Errors from SEQUENCE and CB_SEQUENCE

Any time SEQUENCE or CB_SEQUENCE return an error, the sequenceid of the slot MUST NOT change. The replier MUST NOT modify the reply cache entry for the slot whenever an error is returned from SEQUENCE or CB_SEQUENCE.

2.10.4.1.2. Optional Reply Caching

On a per-request basis the requester can choose to direct the replier to cache the reply to all operations after the first operation (SEQUENCE or CB_SEQUENCE) via the sa_cachethis or csa_cachethis fields of the arguments to SEQUENCE or CB_SEQUENCE. The reason it would not direct the replier to cache the entire reply is that the request is composed of all idempotent operations [20]. Caching the reply may offer little benefit, and if the reply is too large (see Section 2.10.4.4), it may not be cacheable anyway.

Whether the requester requests the reply to be cached or not has no effect on the slot processing. If the results of SEQUENCE or CB_SEQUENCE are NFS4_OK, then the slot’s sequenceid MUST be incremented by one. If a requester does not direct the replier to cache, the reply, the replier MUST do one of following:
o The replier can cache the entire original reply. Even though sa_cachethis or csa_cachethis are FALSE, the replier is always free to cache. It may choose this approach in order to simplify implementation.

o The replier enters into its reply cache a reply consisting of the original results to the SEQUENCE or CB_SEQUENCE operation, followed by the error NFS4ERR_RETRY_UNCACHED_REP. Thus if the requester later retries the request, it will get NFS4ERR_RETRY_UNCACHED_REP.

2.10.4.1.3. Multiple Connections and Sharing the Reply Cache

Multiple connections can be bound to a session’s channel, hence the connections share the same table of slotids. For connections over non-RDMA transports like TCP, there are no particular considerations. Considerations for multiple RDMA connections sharing a slot table are discussed in Section 2.10.5.1. [[Comment.5: Also need to discuss when RDMA and non-RDMA share a slot table.]]

2.10.4.2. Retry and Replay

A client MUST NOT retry a request, unless the connection it used to send the request disconnects. The client can then reconnect and resend the request, or it can resend the request over a different connection. In the case of the server resending over the backchannel, it cannot reconnect, and either resends the request over another connection that the client has bound to the backchannel, or if there is no other backchannel connection, waits for the client to bind a connection to the backchannel.

A client MUST wait for a reply to a request before using the slot for another request. If it does not wait for a reply, then the client does not know what sequenceid to use for the slot on its next request. For example, suppose a client sends a request with sequenceid 1, and does not wait for the response. The next time it uses the slot, it sends the new request with sequenceid 2. If the server has not seen the request with sequenceid 1, then the server is expecting sequenceid 2, and rejects the client’s new request with NFS4ERR_SEQ_MISORDERED (as the result from SEQUENCE or CB_SEQUENCE).

RDMA fabrics do not guarantee that the memory handles (Steering Tags) within each RDMA three-tuple are valid on a scope [[Comment.6: What is a three-tuple?]] outside that of a single connection. Therefore, handles used by the direct operations become invalid after connection loss. The server must ensure that any RDMA operations which must be replayed from the reply cache use the newly provided handle(s) from the most recent request.
2.10.4.3. Resolving server callback races with sessions

It is possible for server callbacks to arrive at the client before the reply from related forward channel operations. For example, a client may have been granted a delegation to a file it has opened, but the reply to the OPEN (informing the client of the granting of the delegation) may be delayed in the network. If a conflicting operation arrives at the server, it will recall the delegation using the callback channel, which may be on a different transport connection, perhaps even a different network. In NFSv4.0, if the callback request arrives before the related reply, the client may reply to the server with an error.

The presence of a session between client and server alleviates this issue. When a session is in place, each client request is uniquely identified by its { slotid, sequenceid } pair. By the rules under which slot entries (reply cache entries) are retired, the server has knowledge whether the client has "seen" each of the server’s replies. The server can therefore provide sufficient information to the client to allow it to disambiguate between an erroneous or conflicting callback and a race condition.

For each client operation which might result in some sort of server callback, the server should "remember" the { slotid, sequenceid } pair of the client request until the slotid retirement rules allow the server to determine that the client has, in fact, seen the server’s reply. Until the time the { slotid, sequenceid } request pair can be retired, any recalls of the associated object MUST carry an array of these referring identifiers (in the CB_SEQUENCE operation’s arguments), for the benefit of the client. After this time, it is not necessary for the server to provide this information in related callbacks, since it is certain that a race condition can no longer occur.

The CB_SEQUENCE operation which begins each server callback carries a list of "referring" { slotid, sequenceid } tuples. If the client finds the request corresponding to the referring slotid and sequenced id be currently outstanding (i.e. the server’s reply has not been seen by the client), it can determine that the callback has raced the reply, and act accordingly.

The client must not simply wait forever for the expected server reply to arrive on any of the session’s operations channels, because it is possible that they will be delayed indefinitely. However, it should wait for a period of time, and if the time expires it can provide a more meaningful error such as NFS4ERR_DELAY.

[[Comment.7: We need to consider the clients’ options here, and
describe them... NFS4ERR_DELAY has been discussed as a legal reply to CB_RECALL?

There are other scenarios under which callbacks may race replies, among them pNFS layout recalls, described in Section 12.5.4.2

2.10.4.4. COMPOUND and CB_COMPOUND Construction Issues

Very large requests and replies may pose both buffer management issues (especially with RDMA) and reply cache issues. When the session is created, (Section 17.36) the client and server negotiate the maximum sized request they will send or process (ca_maxrequestsize), the maximum sized reply they will return or process (ca_maxresponsesize), and the maximum sized reply they will store in the reply cache (ca_maxresponsesize_cached).

If a request exceeds ca_maxrequestsize, the reply will have the status NFS4ERR_REQ_TOO_BIG. A replier may return NFS4ERR_REQ_TOO_BIG as the status for first operation (SEQUENCE or CB_SEQUENCE) in the request, or it may chose to return it on a subsequent operation.

If a reply exceeds ca_maxresponsesize, the reply will have the status NFS4ERR_REP_TOO_BIG. A replier may return NFS4ERR_REP_TOO_BIG as the status for first operation (SEQUENCE or CB_SEQUENCE) in the request, or it may chose to return it on a subsequent operation.

If sa_cachethis or csa_cachethis are TRUE, then the replier MUST cache a reply except if an error is returned by the SEQUENCE or CB_SEQUENCE operation (see Section 2.10.4.1.1). If the reply exceeds ca_maxresponsesize_cached, (and sa_cachethis or csa_cachethis are TRUE) then the server MUST return NFS4ERR_REP_TOO_BIG_TO_CACHE. Even if NFS4ERR_REP_TOO_BIG_TO_CACHE (or any other error for that matter) is returned on a operation other than first operation (SEQUENCE or CB_SEQUENCE), then the reply MUST be cached if sa_cachethis or csa_cachethis are TRUE. For example, if a COMPOUND has eleven operations, including SEQUENCE, the fifth operation is a RENAME, and the tenth operation is a READ for one million bytes, server may return NFS4ERR_REP_TOO_BIG_TO_CACHE on the tenth operation. Since the server executed several operations, especially the non-idempotent RENAME, the client’s request to cache the reply needs to be honored in order for correct operation of exactly once semantics. If the client retries the request, the server will have cached a reply that contains results for ten of the eleven requested operations, with the tenth operation having a status of NFS4ERR_REP_TOO_BIG_TO_CACHE.

A client needs to take care that when sending operations that change the current filehandle (except for PUTFH, PUTPUBFH, and PUTROOTFH)
that it not exceed the maximum reply buffer before the GETFH operation. Otherwise the client will have to retry the operation that changed the current filehandle, in order obtain the desired filehandle. For the OPEN operation (see Section 17.16), retry is not always available as an option. The following guidelines for the handling of filehandle changing operations are advised:

- A client SHOULD issue GETFH immediately after a current filehandle changing operation. This is especially important after any current filehandle changing non-idempotent operation. It is critical to issue GETFH immediately after OPEN.
- A server MAY return NFS4ERR_REP_TOO_BIG or NFS4ERR_REP_TOO_BIG_TO_CACHE (if sa_cachethis is TRUE) on a filehandle changing operation if the reply would be too large on the next operation.
- A server SHOULD return NFS4ERR_REP_TOO_BIG or NFS4ERR_REP_TOO_BIG_TO_CACHE (if sa_cachethis is TRUE) on a filehandle changing non-idempotent operation if the reply would be too large on the next operation, especially if the operation is OPEN.
- A server MAY return NFS4ERR_UNSAFE_COMPOUND if it looks at the next operation after a non-idempotent current filehandle changing operation, and finds it is not GETFH. The server would do this if it is unable to determine in advance whether the total response size would exceed ca_maxresponse_size_cached or ca_maxresponse_size.

2.10.4.5. Persistence

Since the reply cache is bounded, it is practical for the server reply cache to persist across server reboots, and to be kept in stable storage (a client’s reply cache for callbacks need not persist across client reboots unless the client intends for its session and other state to persist across reboots).

- The slot table including the sequenceid and cached reply for each slot.
- The sessionid.
- The client ID.
- The SSV (see Section 2.10.6.3).

The CREATE_SESSION (see Section 17.36) operation determines the persistence of the reply cache.
2.10.5. RDMA Considerations

A complete discussion of the operation of RPC-based protocols atop RDMA transports is in [RPCRDMA]. A discussion of the operation of NFSv4, including NFSv4.1 over RDMA is in [NFSDDP]. Where RDMA is considered, this specification assumes the use of such a layering; it addresses only the upper layer issues relevant to making best use of RPC/RDMA.

2.10.5.1. RDMA Connection Resources

RDMA requires its consumers to register memory and post buffers of a specific size and number for receive operations.

Registration of memory can be a relatively high-overhead operation, since it requires pinning of buffers, assignment of attributes (e.g. readable/writable), and initialization of hardware translation. Preregistration is desirable to reduce overhead. These registrations are specific to hardware interfaces and even to RDMA connection endpoints, therefore negotiation of their limits is desirable to manage resources effectively.

Following the basic registration, these buffers must be posted by the RPC layer to handle receives. These buffers remain in use by the RPC/NFSv4 implementation; the size and number of them must be known to the remote peer in order to avoid RDMA errors which would cause a fatal error on the RDMA connection.

NFSv4.1 manages slots as resources on a per session basis (see Section 2.10), while RDMA connections manage credits on a per connection basis. This means that in order for a peer to send data over RDMA to a remote buffer, it has to have both an NFSv4.1 slot, and an RDMA credit.

2.10.5.2. Flow Control

NFSv4.0 and all previous versions do not provide for any form of flow control; instead they rely on the windowing provided by transports like TCP to throttle requests. This does not work with RDMA, which provides no operation flow control and will terminate a connection in error when limits are exceeded. Limits such as maximum number of requests outstanding are therefore negotiated when a session is created (see the ca_maxrequests field in Section 17.36). These limits then provide the maxima each session’s channels’ connections must operate within. RDMA connections are managed within these limits as described in section 3.3 of [RPCRDMA]; if there are multiple RDMA connections, then the maximum requests for a channel will be divided among the RDMA connections. The limits may also be
modified dynamically at the server’s choosing by manipulating certain parameters present in each NFSv4.1 request. In addition, the CB_RECALL_SLOT callback operation (see Section 19.8) can be issued by a server to a client to return RDMA credits to the server, thereby lowering the maximum number of requests a client can have outstanding to the server.

2.10.5.3. Padding

Header padding is requested by each peer at session initiation (see the csa_headerpadsize argument to CREATE_SESSION in Section 17.36), and subsequently used by the RPC RDMA layer, as described in [RPCRDMA]. Zero padding is permitted.

Padding leverages the useful property that RDMA receives preserve alignment of data, even when they are placed into anonymous (untagged) buffers. If requested, client inline writes will insert appropriate pad bytes within the request header to align the data payload on the specified boundary. The client is encouraged to add sufficient padding (up to the negotiated size) so that the "data" field of the NFSv4.1 WRITE operation is aligned. Most servers can make good use of such padding, which allows them to chain receive buffers in such a way that any data carried by client requests will be placed into appropriate buffers at the server, ready for file system processing. The receiver’s RPC layer encounters no overhead from skipping over pad bytes, and the RDMA layer’s high performance makes the insertion and transmission of padding on the sender a significant optimization. In this way, the need for servers to perform RDMA Read to satisfy all but the largest client writes is obviated. An added benefit is the reduction of message round trips on the network – a potentially good trade, where latency is present.

The value to choose for padding is subject to a number of criteria. A primary source of variable-length data in the RPC header is the authentication information, the form of which is client-determined, possibly in response to server specification. The contents of COMPOUNDS, sizes of strings such as those passed to RENAME, etc. all go into the determination of a maximal NFSv4 request size and therefore minimal buffer size. The client must select its offered value carefully, so as not to overburden the server, and vice-versa. The payoff of an appropriate padding value is higher performance.

Sender gather:

| RPC Request | Pad bytes | Length | -> | User data...

Receiver scatter:

Sender gather:

| RPC Request | Pad bytes | Length | -> | User data...

Receiver scatter:

In the above case, the server may recycle unused buffers to the next posted receive if unused by the actual received request, or may pass the now-complete buffers by reference for normal write processing. For a server which can make use of it, this removes any need for data copies of incoming data, without resorting to complicated end-to-end buffer advertisement and management. This includes most kernel-based and integrated server designs, among many others. The client may perform similar optimizations, if desired.

2.10.5.4. Dual RDMA and Non-RDMA Transports

Some RDMA transports (for example see [RDDP]), [[[Comment.9: need xref]]] require a "streaming" (non-RDMA) phase, where ordinary traffic might flow before "stepping" up to RDMA mode, commencing RDMA traffic. Some RDMA transports start connections always in RDMA mode. NFSv4.1 allows, but does not assume, a streaming phase before RDMA mode. When a connection is bound to a session, the client and server negotiate whether the connection is used in RDMA or non-RDMA mode (see Section 17.36 and Section 17.34).

2.10.6. Sessions Security

2.10.6.1. Session Callback Security

Via session connection binding, NFSv4.1 improves security over that provided by NFSv4.0 for the callback channel. The connection is client-initiated (see Section 17.34), and subject to the same firewall and routing checks as the operations channel. The connection cannot be hijacked by an attacker who connects to the client port prior to the intended server. At the client’s option (see Section 17.36 binding is fully authenticated before being activated (see Section 17.34). Traffic from the server over the callback channel is authenticated exactly as the client specifies (see Section 2.10.6.2).

2.10.6.2. Backchannel RPC Security

When the NFSv4.1 client establishes the backchannel, it informs the server what security flavors and principals it must use when sending requests over the backchannel. If the security flavor is RPCSEC_GSS, the client expresses the principal in the form of an established RPCSEC_GSS context. The server is free to use any flavor/principal combination the server offers, but MUST NOT use unoffered combinations.

This way, the client does not have to provide a target GSS principal
as it did with NFSv4.0, and the server does not have to implement an
RPCSEC_GSS initiator as it did with NFSv4.0.  ([Comment.10: xrefs])

The CREATE_SESSION (Section 17.36) and BACKCHANNEL_CTL
(Section 17.33) operations allow the client to specify flavor/
principal combinations.

2.10.6.3. Protection from Unauthorized State Changes

Under some conditions, NFSv4.0 is vulnerable to a denial of service
issue with respect to its state management.

The attack works via an unauthorized client faking an open_owner4, an
open_owner/lock_owner pair, or stateid, combined with a seqid. The
operation is sent to the NFSv4 server. The NFSv4 server accepts the
state information, and as long as any status code from the result of
this operation is not NFS4ERR_STATE_CLIENTID, NFS4ERR_STATE_STATEID,
NFS4ERR_BAD_STATEID, NFS4ERR_BAD_SEQID, NFS4ERR_BADXDR,
NFS4ERR_RESOURCE, or NFS4ERR_NOFILEHANDLE, the sequence number is
incremented. When the authorized client issues an operation, it gets
back NFS4ERR_BAD_SEQID, because its idea of the current sequence
number is off by one. The authorized client’s recovery options are
pretty limited, with SETCLIENTID, followed by complete reclaim of
state, which may or may not succeed completely. That qualifies as a
denial of service attack.

If the client uses RPCSEC_GSS authentication and integrity, and every
client maps each open_owner and lock_owner one and only one
principal, and the server enforces this binding, then the conditions
leading to vulnerability to the denial of service do not exist. One
should keep in mind that if AUTH_SYS is being used, far simpler
easier denial of service and other attacks are possible.

With NFSv4.1 sessions, the per-operation sequence number is ignored
(see Section 8.13) therefore the NFSv4.0 denial of service
vulnerability described above does not apply. However as described
to this point in the specification, an attacker could forge the
sessionid and issue a SEQUENCE with a slot id that he expects the
legitimate client to use next. The legitimate client could then use
the slotid with the same sequence number, and the server returns the
attacker’s result from the replay cache, thereby disrupting the
legitimate client.

If we give each NFSv4.1 user their own session, and each user uses
RPCSEC_GSS authentication and integrity, then the denial of service
issue is solved, at the cost of additional per session state. The
alternative NFSv4.1 specifies is described as follows.
Transport connections MUST be bound to a session by the client. The server MUST return an error to an operation (other than the operation that binds the connection to the session) that uses an unbound connection. As a simplification, the transport connection used by CREATE_SESSION (see Section 17.36) is automatically bound to the session. Additional connections are bound to a session via BIND_CONN_TO_SESSION (see Section 17.34).

To prevent attackers from issuing BIND_CONN_TO_SESSION operations, the arguments to BIND_CONN_TO_SESSION include a digest of a shared secret called the secret session verifier (SSV) that only the client and server know. The digest is created via a one way, collision resistant hash function, making it intractable for the attacker to forge.

The SSV is sent to the server via SET_SSV (see Section 17.47). To prevent eavesdropping, a SET_SSV for the SSV SHOULD be protected via RPCSEC_GSS with the privacy service. The SSV can be changed by the client at any time, by any principal. However several aspects of SSV changing prevent an attacker from engaging in a successful denial of service attack:

- A SET_SSV on the SSV does not replace the SSV with the argument to SET_SSV. Instead, the current SSV on the server is logically exclusive ORed (XORed) with the argument to SET_SSV. SET_SSV MUST NOT be called with an SSV value that is zero.

- The arguments to and results of SET_SSV include digests of the old and new SSV, respectively.

- Because the initial value of the SSV is zero, therefore known, the client that opts for connecting binding enforcement, MUST issue at least one SET_SSV operation before the first BIND_CONN_TO_SESSION operation. A client SHOULD issue SET_SSV as soon as a session is created.

If a connection is disconnected, BIND_CONN_TO_SESSION is required to bind a connection to the session, even if the connection that was disconnected was the one CREATE_SESSION was created with.

If a client is assigned a machine principal then the client SHOULD use the machine principal’s RPCSEC_GSS context to privacy protect the SSV from eavesdropping during the SET_SSV operation. If a machine principal is not being used, then the client MAY use the non-machine principal’s RPCSEC_GSS context to privacy protect the SSV. The server MUST accept either type of principal. A client SHOULD change the SSV each time a new principal uses the session.
Here are the types of attacks that can be attempted by an attacker named Eve, and how the connection to session binding approach addresses each attack:

- If the Eve creates a connection after the legitimate client establishes an SSV via privacy protection from a machine principal’s RPCSEC_GSS session, she does not know the SSV and so cannot compute a digest that BIND_CONN_TO_SESSION will accept. Users on the legitimate client cannot be disrupted by Eve.

- If Eve is the first one log into the legitimate client, and the client does not use machine principals, then Eve can cause an SSV to be created via the legitimate client’s NFSv4.1 implementation, protected by the RPCSEC_GSS context created by the legitimate client (which uses Eve’s GSS principal and credentials). Eve can then eavesdrop on the network, and because she knows her credentials, she can decrypt the SSV. Eve can compute a digest BIND_CONN_TO_SESSION will accept, and so bind a new connection to the session. Eve can change the slotid, sequence state, and/or the SSV state in such a way that when Bob accesses the server via the legitimate client, the legitimate client will be unable to use the session.

The client’s only recourse is to create a new session, which will cause any state Eve created on the legitimate client over the old (but hijacked) session to be lost. This disrupts Eve, but because she is the attacker, this is acceptable.

Once the legitimate client establishes an SSV over the new session using Bob’s RPCSEC_GSS context, Eve can use the new session via the legitimate client, but she cannot disrupt Bob. Moreover, because the client SHOULD have modified the SSV due to Eve using the new session, Bob cannot get revenge on Eve by binding a rogue connection to the session.

The question is how does the legitimate client detect that Eve has hijacked the old session? When the client detects that a new principal, Bob, wants to use the session, it SHOULD have issued a SET_SSV.

* Let us suppose that from the rogue connection, Eve issued a SET_SSV with the same slotid and sequence that the legitimate client later uses. The server will assume this is a replay, and return to the legitimate client the reply it sent Eve. However, unless Eve can correctly guess the SSV the legitimate client will use, the digest verification checks in the SET_SSV response will fail. That is the clue to the client that the session has been hijacked.
Alternatively, Eve issued a SET_SSV with a different slotid than the legitimate client uses for its SET_SSV. Then the digest verification on the server fails, and the client is again clued that the session has been hijacked.

Alternatively, Eve issued an operation other than SET_SSV, but with the same slotid and sequence that the legitimate client uses for its SET_SSV. The server returns to the legitimate client the response it sent Eve. The client sees that the response is not at all what it expects. The client assumes either session hijacking or server bug, and either way destroys the old session.

- Eve binds a rogue connection to the session as above, and then destroys the session. Again, Bob goes to use the server from the legitimate client. The client has a very clear indication that its session was hijacked, and does not even have to destroy the old session before creating a new session, which Eve will be unable to hijack because it will be protected with an SSV created via Bob’s RPCSEC_GSS protection.

- If Eve creates a connection before the legitimate client establishes an SSV, because the initial value of the SSV is zero and therefore known, Eve can issue a SET_SSV that will pass the digest verification check. However because the new connection has not been bound to the session, the SET_SSV is rejected for that reason.

- The connection to session binding model does not prevent connection hijacking. However, if an attacker can perform connection hijacking, it can issue denial of service attacks that are less difficult than attacks based on forging sessions.

2.10.7. Session Mechanics - Steady State

2.10.7.1. Obligations of the Server

The server has the primary obligation to monitor the state of backchannel resources that the client has created for the server (RPCSEC_GSS contexts and back channel connections). When these resources go away, the server takes action as specified in Section 2.10.8.2.

2.10.7.2. Obligations of the Client

The client has the following obligations in order to utilize the session:
- Keep a necessary session from going idle on the server. A client that requires a session, but nonetheless is not sending operations risks having the session be destroyed by the server. This is because sessions consume resources, and resource limitations may force the server to cull the least recently used session.

- Destroy the session when idle. When a session has no state other than the session, and no outstanding requests, the client should consider destroying the session.

- Maintain GSS contexts for callback. If the client requires the server to use the RPCSEC_GSS security flavor for callbacks, then it needs to be sure the contexts handed to the server via BACKCHANNEL_CTL are unexpired. A good practice is to keep at least two contexts outstanding, where the expiration time of the newest context at the time it was created, is \( N \) times that of the oldest context, where \( N \) is the number of contexts available for callbacks.

- Maintain an active connection. The server requires a callback path in order to gracefully recall recallable state, or notify the client of certain events.

### 2.10.7.3. Steps the Client Takes To Establish a Session

The client issues EXCHANGE_ID to establish a client ID.

The client uses the client ID to issue a CREATE_SESSION on a connection to the server. The results of CREATE_SESSION indicate whether the server will persist the session replay cache through a server reboot or not, and the client notes this for future reference.

The client SHOULD have specified connecting binding enforcement when the session was created. If so, the client SHOULD issue SET_SSV in the first COMPOUND after the session is created. If it is not using machine credentials, then each time a new principal goes to use the session, it SHOULD issue a SET_SSV again.

If the client wants to use delegations, layouts, directory notifications, or any other state that requires a callback channel, then it MUST add a connection to the backchannel if CREATE_SESSION did not already do so. The client creates a connection, and calls BIND_CONN_TO_SESSION to bind the connection to the session and the session’s backchannel. If CREATE_SESSION did not already do so, the client MUST tell the server what security is required in order for the client to accept callbacks. The client does this via BACKCHANNEL_CTL.
If the client wants to use additional connections for the backchannel, then it MUST call BIND_CONN_TO_SESSION on each connection it wants to use with the session. If the client wants to use additional connections for the operation channel, then it MUST call BIND_CONN_TO_SESSION if it specified connection binding enforcement before using the connection.

At this point the client has reached a steady state as far as session use.

2.10.8. Session Mechanics - Recovery

2.10.8.1. Events Requiring Client Action

The following events require client action to recover.

2.10.8.1.1. RPCSEC_GSS Context Loss by Callback Path

If all RPCSEC_GSS contexts granted to by the client to the server for callback use have expired, the client MUST establish a new context via BACKCHANNEL_CTL. The sr_status_flags field of the SEQUENCE results indicates when callback contexts are nearly expired, or fully expired (see Section 17.46.4).

2.10.8.1.2. Connection Disconnect

If the client loses the last connection of the session, then it MUST create a new connection, and if connecting binding enforcement was specified when the session was created, bind it to the session via BIND_CONN_TO_SESSION.

If there were requests outstanding at the time the of connection disconnect, then the client MUST retry the request, as described in Section 2.10.4.2. Note that it is not necessary to retry requests over a connection with the same source network address or the same destination network address as the disconnected connection. As long as the sessionid, slotid, and sequenceid in the retry match that of the original request, the server will recognize the request as a retry if it did see the request prior to disconnect.

If the connection that was bound to the backchannel is lost, the client may need to reconnect, and use BIND_CONN_TO_SESSION, to give the connection to the backchannel. If the connection that was lost was the last one bound to the backchannel, the client MUST reconnect, and bind the connection to the session and backchannel. The server should indicate when it has no callback connection via the sr_status_flags result from SEQUENCE.
2.10.8.1.3. Backchannel GSS Context Loss

Via the sr_status_flags result of the SEQUENCE operation or other means, the client will learn if some or all of the RPCSEC_GSS contexts it assigned to the backchannel have been lost. The client may need to use BACKCHANNEL_CTL to assign new contexts. It MUST assign new contexts if there are no more contexts.

2.10.8.1.4. Loss of Session

The server may lose a record of the session. Causes include:

- Server crash and reboot
- A catastrophe that causes the cache to be corrupted or lost on the media it was stored on. This applies even if the server indicated in the CREATE_SESSION results that it would persist the cache.
- The server purges the session of a client that has been inactive for a very extended period of time. [Comment.11: XXX - Should we add a value to the CREATE_SESSION results that tells a client how long he can let a session stay idle before losing it?]

Loss of replay cache is equivalent to loss of session. The server indicates loss of session to the client by returning NFS4ERR_BADSESSION on the next operation that uses the sessionid associated with the lost session.

After an event like a server reboot, the client may have lost its connections. The client assumes for the moment that the session has not been lost. It reconnects, and if it specified connecting binding enforcement when the session was created, it invokes BIND_CONN_TO_SESSION using the sessionid. Otherwise, it invokes SEQUENCE. If BIND_CONN_TO_SESSION or SEQUENCE returns NFS4ERR_BADSESSION, the client knows the session was lost. If the connection survives session loss, then the next SEQUENCE operation the client issues over the connection will get back NFS4ERR_BADSESSION. The client again knows the session was lost.

When the client detects session loss, it must call CREATE_SESSION to recover. Any non-idempotent operations that were in progress may have been performed on the server at the time of session loss. The client has no general way to recover from this.

Note that loss of session does not imply loss of lock, open, delegation, or layout state. Nor does loss of lock, open, delegation, or layout state imply loss of session state. [Comment.12: Add reference to lock recovery section] . A session
can survive a server reboot, but lock recovery may still be needed. The converse is also true.

It is possible CREATE_SESSION will fail with NFS4ERR_STALE_CLIENTID (for example the server reboots and does not preserve client ID state). If so, the client needs to call EXCHANGE_ID, followed by CREATE_SESSION.

2.10.8.1.5. Failover

[[Comment.13: Dave Noveck requested this section; not sure what is needed here if this refers to failover to a replica. What are the session ramifications?]]

2.10.8.2. Events Requiring Server Action

The following events require server action to recover.

2.10.8.2.1. Client Crash and Reboot

As described in Section 17.35, a rebooted client causes the server to delete any sessions it had.

2.10.8.2.2. Client Crash with No Reboot

If a client crashes and never comes back, it will never issue EXCHANGE_ID with its old client owner. Thus the server has session state that will never be used again. After an extended period of time and if the server has resource constraints, it MAY destroy the old session.

2.10.8.2.3. Extended Network Partition

To the server, the extended network partition may be no different than a client crash with no reboot (see Section 2.10.8.2.2). Unless the server can discern that there is a network partition, it is free to treat the situation as if the client has crashed for good.

2.10.8.2.4. Backchannel Connection Loss

If there were callback requests outstanding at the time the of a connection disconnect, then the server MUST retry the request, as described in Section 2.10.4.2. Note that it is not necessary to retry requests over a connection with the same source network address or the same destination network address as the disconnected connection. As long as the sessionid, slotid, and sequenceid in the retry match that of the original request, the callback target will recognize the request as a retry if it did see the request prior to
disconnect.

If the connection lost is the last one bound to the backchannel, then the server MUST indicate that in the sr_status_flags field of the next SEQUENCE reply.

2.10.8.2.5. GSS Context Loss

The server SHOULD monitor when the last RPCSEC_GSS context assigned to the backchannel is near expiry (i.e. between one and two periods of lease time), and indicate so in the sr_status_flags field of the next SEQUENCE reply. The server MUST indicate when the backchannel’s last RPCSEC_GSS context has expired in the sr_status_flags field of the next SEQUENCE reply.

2.10.9. Parallel NFS and Sessions

A client and server can potentially be a non-pNFS implementation, a metadata server implementation, a data server implementation, or two or three types of implementations. The EXCHGID4_FLAG_USE_NON_PNFS, EXCHGID4_FLAG_USE_PNFS_MDS, and EXCHGID4_FLAG_USE_PNFS_DS flags (not mutually exclusive) are passed in the EXCHANGE_ID arguments and results to allow the client to indicate how it wants to use sessions created under the client ID, and to allow the server to indicate how it will allow the sessions to be used. See Section 13.1 for pNFS sessions considerations.

3. Protocol Data Types

The syntax and semantics to describe the data types of the NFS version 4 protocol are defined in the XDR RFC4506 [3] and RPC RFC1831 [4] documents. The next sections build upon the XDR data types to define types and structures specific to this protocol.

3.1. Basic Data Types

These are the base NFSv4 data types.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32_t</td>
<td>typedef int int32_t;</td>
</tr>
<tr>
<td>uint32_t</td>
<td>typedef unsigned int uint32_t;</td>
</tr>
<tr>
<td>int64_t</td>
<td>typedef hyper int64_t;</td>
</tr>
<tr>
<td>uint64_t</td>
<td>typedef unsigned hyper uint64_t;</td>
</tr>
<tr>
<td>attrlist4</td>
<td>typedef opaque attrlist4&lt;&gt;;</td>
</tr>
<tr>
<td></td>
<td>Used for file/directory attributes</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>bitmap4</td>
<td>Used in attribute array encoding.</td>
</tr>
<tr>
<td>changeid4</td>
<td>Used in definition of change_info</td>
</tr>
<tr>
<td>clientid4</td>
<td>Shorthand reference to client identification</td>
</tr>
<tr>
<td>component4</td>
<td>Represents path name components</td>
</tr>
<tr>
<td>count4</td>
<td>Various count parameters (READ, WRITE, COMMIT)</td>
</tr>
<tr>
<td>length4</td>
<td>Describes LOCK lengths</td>
</tr>
<tr>
<td>linktext4</td>
<td>Symbolic link contents</td>
</tr>
<tr>
<td>mode4</td>
<td>Mode attribute data type</td>
</tr>
<tr>
<td>nfs_cookie4</td>
<td>Opaque cookie value for READDIR</td>
</tr>
<tr>
<td>nfs_fh4</td>
<td>Filehandle definition; NFS4_FHSIZE is defined as 128</td>
</tr>
<tr>
<td>nfs_fstype4</td>
<td>Various defined file types</td>
</tr>
<tr>
<td>nfsstat4</td>
<td>Return value for operations</td>
</tr>
<tr>
<td>offset4</td>
<td>Various offset designations (READ, WRITE, LOCK, COMMIT)</td>
</tr>
<tr>
<td>pathname4</td>
<td>Represents path name for fs_locations</td>
</tr>
<tr>
<td>qop4</td>
<td>Quality of protection designation in SECINFO</td>
</tr>
<tr>
<td>sec_oid4</td>
<td>Security Object Identifier The sec_oid4 data type is not really opaque.</td>
</tr>
<tr>
<td>sequenceid4</td>
<td>Sequence number used for various session operations</td>
</tr>
<tr>
<td>seqid4</td>
<td>Sequence identifier used for file locking</td>
</tr>
<tr>
<td>sessionid4</td>
<td>Session identifier</td>
</tr>
<tr>
<td>slotid4</td>
<td></td>
</tr>
</tbody>
</table>

Internet-Draft            NFSv4 Minor Version 1               March 2007

End of Base Data Types

Table 1

3.2. Structured Data Types

3.2.1. 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.
3.2.2. time_how4

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

3.2.3. settime4

union settime4 switch (time_how4 set_it) {
    case SET_TO_CLIENT_TIME4:
        nftime4       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.

3.2.4. 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.

3.2.5. fsid4

struct fsid4 {
    uint64_t       major;
    uint64_t       minor;
};

3.2.6. fs_location4

struct fs_location4 {
    utf8str_cis    server<>;
    pathname4      rootpath;
};
3.2.7. 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.

3.2.8. fatr4

    struct fatr4 {
        bitmap4   attrmask;
        attrlist4 attr_vals;
    };

The fatr4 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 \((n \mod 32)\).

\[
\begin{array}{c|c|c}
0 & 1 \\
+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+---------------+-------------
3.2.10. netaddr4

struct netaddr4 {
    /* see struct rpcb in RFC1833 */
    string r_netid<>;    /* network id */
    string r_addr<>;     /* universal address */
};

The netaddr4 structure is used to identify TCP/IP based endpoints. The r_netid and r_addr fields are specified in RFC1833 [22], but they are underspecified in RFC1833 [22] as far as what they should look like for specific protocols.

For TCP over IPv4 and for UDP over IPv4, the format of r_addr is the US-ASCII string:

h1.h2.h3.h4.p1.p2

The prefix, "h1.h2.h3.h4", is the standard textual form for representing an IPv4 address, which is always four octets long. Assuming big-endian ordering, h1, h2, h3, and h4, are respectively, the first through fourth octets each converted to ASCII-decimal. Assuming big-endian ordering, p1 and p2 are, respectively, the first and second octets each converted to ASCII-decimal. For example, if a host, in big-endian order, has an address of 0x0A010307 and there is a service listening on, in big endian order, port 0x020F (decimal 527), then complete universal address is "10.1.3.7.2.15".

For TCP over IPv4 the value of r_netid is the string "tcp". For UDP over IPv4 the value of r_netid is the string "udp". That this document specifies the universal address and netid for UDP/IPv4 does not imply that UDP/IPv4 is a legal transport for NFSv4.1 (see Section 2.9).

For TCP over IPv6 and for UDP over IPv6, the format of r_addr is the US-ASCII string:

x1:x2:x3:x4:x5:x6:x7:x8.p1.p2

The suffix "p1.p2" is the service port, and is computed the same way as with universal addresses for TCP and UDP over IPv4. The prefix, "x1:x2:x3:x4:x5:x6:x7:x8", is the standard textual form for representing an IPv6 address as defined in Section 2.2 of RFC1884 [9]. Additionally, the two alternative forms specified in Section 2.2 of RFC1884 [9] are also acceptable.

For TCP over IPv6 the value of r_netid is the string "tcp6". For UDP over IPv6 the value of r_netid is the string "udp6". That this
document specifies the universal address and netid for UDP/IPv6 does not imply that UDP/IPv6 is a legal transport for NFSv4.1 (see Section 2.9).

3.2.11. open_owner4

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

This structure is used to identify the owner of open state. NFS4_OPAQUE_LIMIT is defined as 1024.

3.2.12. lock_owner4

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

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

3.2.13. 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.

3.2.14. stateid4

    struct stateid4 {
        uint32_t        seqid;
        opaque          other[12];
    };

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 starting value of the seqid field is undefined.
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.

3.2.15. layouttype4

enum layouttype4 {
    LAYOUT4_NFSV4_1_FILES = 1,
    LAYOUT4_OSD2_OBJECTS = 2,
    LAYOUT4_BLOCK_VOLUME = 3
};

A layout type specifies the layout being used. The implication is
that clients have "layout drivers" that support one or more layout
types. The file server advertises the layout types it supports
through the fs_layout_type file system attribute (Section 5.13.1). A
client asks for layouts of a particular type in LAYOUTGET, and passes
those layouts to its layout driver.

The layouttype4 structure is 32 bits in length. The range
represented by the layout type is split into three parts. Type 0x0
is reserved. Types within the range 0x00000001-0x7FFFFFFF are
globally unique and are assigned according to the description in
Section 21.1; they are maintained by IANA. Types within the range
0x80000000-0xFFFFFFFF are site specific and for "private use" only.

The LAYOUT4_NFSV4_1_FILES enumeration specifies that the NFSv4.1 file
layout type is to be used. The LAYOUT4_OSD2_OBJECTS enumeration
specifies that the object layout, as defined in [23], is to be used.
Similarly, the LAYOUT4_BLOCK_VOLUME enumeration that the block/volume
layout, as defined in [24], is to be used.

3.2.16. deviceid4

typedef uint32_t deviceid4; /* 32-bit device ID */

Layout information includes device IDs that specify a storage device
through a compact handle. Addressing and type information is
obtained with the GETDEVICEINFO operation. A client must not assume
that device IDs are valid across metadata server reboots. The device
ID is qualified by the layout type and are unique per file system
(FSID). This allows different layout drivers to generate device IDs
without the need for co-ordination. See Section 12.2.12 for more
details.
3.2.17.  device_addr4

struct device_addr4 {
    layouttype4    da_layout_type;
    opaque         da_addr_body<>;
};

The device address is used to set up a communication channel with the
storage device. Different layout types will require different types
of structures to define how they communicate with storage devices.
The opaque da_addr_body field must be interpreted based on the
specified da_layout_type field.

This document defines the device address for the NFSv4.1 file layout
([[Comment.14: need xref]]), which identifies a storage device by
network IP address and port number. This is sufficient for the
clients to communicate with the NFSv4.1 storage devices, and may be
sufficient for other layout types as well. Device types for object
storage devices and block storage devices (e.g., SCSI volume labels)
will be defined by their respective layout specifications.

3.2.18.  devlist_item4

struct devlist_item4 {
    deviceid4       dli_id;
    device_addr4    dli_device_addr<>;
};

An array of these values is returned by the GETDEVICELIST operation.
They define the set of devices associated with a file system for the
layout type specified in the GETDEVICELIST4args.

3.2.19.  layout_content4

struct layout_content4 {
    layouttype4 loc_type;
    opaque       loc_body<>;
};

The loc_body field must be interpreted based on the layout type
(loc_type). This document defines the loc_body for the NFSv4.1 file
layout type is defined; see Section 13.3 for its definition.
3.2.20. layout4

struct layout4 {
    offset4    lo_offset;
    length4    lo_length;
    layoutiomode4 lo_iomode;
    layout_content4 lo_content;
};

The layout4 structure defines a layout for a file. The layout type specific data is opaque within lo_content. Since layouts are subdividable, the offset and length together with the file’s filehandle, the client ID, iomode, and layout type, identifies the layout.

3.2.21. layoutupdate4

struct layoutupdate4 {
    layouttype4    lou_type;
    opaque          lou_body<>
};

The layoutupdate4 structure is used by the client to return ‘updated’ layout information to the metadata server at LAYOUTCOMMIT time. This structure provides a channel to pass layout type specific information (in field lou_body) back to the metadata server. E.g., for block/volume layout types this could include the list of reserved blocks that were written. The contents of the opaque lou_body argument are determined by the layout type and are defined in their context. The NFSv4.1 file-based layout does not use this structure, thus the lou_body field should have a zero length.

3.2.22. layouthint4

struct layouthint4 {
    layouttype4    loh_type;
    opaque          loh_body<>
};

The layouthint4 structure is used by the client to pass in a hint about the type of layout it would like created for a particular file. It is the structure specified by the layout_hint attribute described in Section 5.13.4. The metadata server may ignore the hint, or may selectively ignore fields within the hint. This hint should be provided at create time as part of the initial attributes within OPEN. The loh_body field is specific to the type of layout (loh_type). The NFSv4.1 file-based layout uses the nfsv4_1_file_layouthint4 structure as defined in Section 13.3.
3.2.23. layoutiomode4

    enum layoutiomode4 {
        LAYOUTIOMODE4_READ          = 1,
        LAYOUTIOMODE4_RW            = 2,
        LAYOUTIOMODE4_ANY           = 3
    };

The iomode specifies whether the client intends to read or write
(with the possibility of reading) the data represented by the layout.
The ANY iomode MUST NOT be used for LAYOUTGET, however, it can be
used for LAYOUTRETURN and LAYOUTRECALL. The ANY iomode specifies
that layouts pertaining to both READ and RW iomodes are being
returned or recalled, respectively. The metadata server’s use of the
iomode may depend on the layout type being used. The storage devices
may validate I/O accesses against the iomode and reject invalid
accesses.

3.2.24. nfs_impl_id4

    struct nfs_impl_id4 {
        utf8str_cis   nii_domain;
        utf8str_cs    nii_name;
        nfstime4      nii_date;
    };

This structure is used to identify client and server implementation
detail. The nii_domain field is the DNS domain name that the
implementer is associated with. The nii_name field is the product
name of the implementation and is completely free form. It is
recommended that the nii_name be used to distinguish machine
architecture, machine platforms, revisions, versions, and patch
levels. The nii_date field is the timestamp of when the software
instance was published or built.

3.2.25. threshold_item4

    struct threshold_item4 {
        layouttype4     thi_layout_type;
        bitmap4         thi_hintset;
        opaque          thi_hintlist<>;
    };

This structure contains a list of hints specific to a layout type for
helping the client determine when it should issue I/O directly
through the metadata server vs. the data servers. The hint structure
consists of the layout type (thi_layout_type), a bitmap (thi_hintset)
describing the set of hints supported by the server (they may differ
based on the layout type), and a list of hints (thi_hintlist), whose structure is determined by the hintset bitmap. See the mdsthreshold attribute for more details.

The thi_hintset field is a bitmap of the following values:

<table>
<thead>
<tr>
<th>name</th>
<th>#</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>threshold4_read_size</td>
<td>0</td>
<td>length4</td>
<td>The file size below which it is recommended to read data through the MDS.</td>
</tr>
<tr>
<td>threshold4_write_size</td>
<td>1</td>
<td>length4</td>
<td>The file size below which it is recommended to write data through the MDS.</td>
</tr>
<tr>
<td>threshold4_read_iosize</td>
<td>2</td>
<td>length4</td>
<td>For read I/O sizes below this threshold it is recommended to read data through the MDS.</td>
</tr>
<tr>
<td>threshold4_write_iosize</td>
<td>3</td>
<td>length4</td>
<td>For write I/O sizes below this threshold it is recommended to write data through the MDS.</td>
</tr>
</tbody>
</table>

3.2.26. mdsthreshold4

struct mdsthreshold4 {
    threshold_item4 mth_hints<>
};

This structure holds an array of threshold_item4 structures each of which is valid for a particular layout type. An array is necessary since a server can support multiple layout types for a single file.

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 NFS version 2 protocol RFC1094 [17] and the NFS version 3 protocol RFC1813 [18], 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 [25] and RFC2055 [26]. With the use of the public filehandle in combination with the LOOKUP operation in the NFS version 2 and 3 protocols, it has been demonstrated that the MOUNT protocol is unnecessary for viable interaction between NFS client and server.

Therefore, the NFS version 4 protocol will not use an ancillary protocol for translation from string based path names to a filehandle. Two special filehandles will be used as starting points for the NFS client.

4.1.1. Root Filehandle

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

4.1.2. Public Filehandle

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

In the NFS version 2 and 3 protocols, there was one type of filehandle with a single set of semantics. This type of filehandle is termed "persistent" in NFS Version 4. The semantics of a persistent filehandle remain the same as before. A new type of filehandle introduced in NFS Version 4 is the "volatile" filehandle, which attempts to accommodate certain server environments.

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

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

4.2.1. General Properties of a Filehandle

The filehandle contains all the information the server needs to distinguish an individual file. To the client, the filehandle is opaque. The client stores filehandles for use in a later request and can compare two filehandles from the same server for equality by doing an octet-by-octet 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.

Servers SHOULD try to maintain a one-to-one correspondence between filehandles and files but this is not required. Clients MUST use filehandle comparisons only to improve performance, not for correct behavior. All clients need to be prepared for situations in which it cannot be determined whether two filehandles denote the same object and in such cases, avoid making invalid assumptions which might cause incorrect behavior. Further discussion of filehandle and attribute comparison in the context of data caching is presented in the section "Data Caching and File Identity".

As an example, in the case that two different path names when traversed at the server terminate at the same file system object, the server SHOULD return the same filehandle for each path. This can
occur if a hard link is used to create two file names which refer to
the same underlying file object and associated data. For example, if
paths /a/b/c and /a/d/c refer to the same file, the server SHOULD
return the same filehandle for both path names traversals.

4.2.2. Persistent Filehandle

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

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

4.2.3. Volatile Filehandle

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

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

FH4_PERSISTENT The value of FH4_PERSISTENT is used to indicate a
persistent filehandle, which is valid until the object is removed
from the file system. The server will not return
NFS4ERR_FHEXPIRED for this filehandle. FH4_PERSISTENT is defined
as a value in which none of the bits specified below are set.
FH4_VOLATILE_ANY  The filehandle may expire at any time, except as specifically excluded (i.e. FH4_NO_EXPIRE_WITH_OPEN).

FH4_NO_EXPIRE_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 a file system transition (migration or replication), in those cases in which the continuity of filehandle use is not specified by _handle_ class information within the fs_locations_info attribute. When this bit is set, clients without access to fs_locations_info information should assume filehandles will expire on file system transitions.

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_VOL_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_NO_EXPIRE_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.

Servers which provide volatile filehandles that may expire while open require special care as regards handling of RENAMESs and REMOVEs. This situation can arise if FH4_VOL_MIGRATION or FH4_VOL_RENAME is set, if FH4_VOLATILE_ANY is set and FH4_NO_EXPIRE_WITH_OPEN not set, or if a non-readonly file system has a transition target in a different _handle_ class. In these cases, the server 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, in order to make sure that reclaims of files where filehandles may have expired do not do a reclaim for the wrong file.

4.3. 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_FHEXPRIRED. If slot is out of range, return NFS4ERR_BADHANDLE. If the generation number does not match, return NFS4ERR_FHEXPRIRED.

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.

4.4. Client Recovery from Filehandle Expiration

If possible, the client SHOULD recover from the receipt of an NFS4ERR_FHEXPRIRED 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. File Attributes

To meet the requirements of extensibility and increased interoperability with non-UNIX platforms, attributes must be handled in a flexible manner. The NFS version 3 fatattr3 structure contains a fixed list of attributes that not all clients and servers are able to support or care about. The fatattr3 structure can not be extended as new needs arise and it provides no way to indicate non-support. With the NFS version 4 protocol, the client is able 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: mandatory, recommended, and named. Both mandatory and recommended attributes are supported in the NFS version 4 protocol by a specific and well-defined encoding and are identified by number. They are requested by setting a bit in the bit vector sent in the GETATTR request; the server response includes a bit vector to list what attributes were returned in the response. New mandatory or recommended attributes may be added to the NFS protocol between major revisions by publishing a standards-track RFC which allocates a new attribute number value and defines the encoding for the attribute. See the section "Minor Versioning" for further discussion.

Named attributes are accessed by the new OPENATTR operation, which accesses a hidden directory of attributes associated with a file system object. OPENATTR takes a filehandle for the object and returns the filehandle for the attribute hierarchy. The filehandle for the named attributes is a directory object accessible by LOOKUP or READDIR and contains files whose names represent the named attributes and whose data bytes are the value of the attribute. For example:
<table>
<thead>
<tr>
<th>Command</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOKUP</td>
<td>&quot;foo&quot;</td>
<td>look up file</td>
</tr>
<tr>
<td>GETATTR</td>
<td>attrbits</td>
<td></td>
</tr>
<tr>
<td>OPENATTR</td>
<td></td>
<td>access foo’s named attributes</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>&quot;x11icon&quot;</td>
<td>look up specific attribute</td>
</tr>
<tr>
<td>READ</td>
<td>0,4096</td>
<td>read stream of bytes</td>
</tr>
</tbody>
</table>

Named attributes are intended for data needed by applications rather than by an NFS client implementation. NFS implementors are strongly encouraged to define their new attributes as recommended attributes by bringing them to the IETF standards-track process.

The set of attributes which are classified as mandatory is deliberately small since servers must do whatever it takes to support them. 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 mandatory if the data is both needed by a large number of clients and is not otherwise reasonably computable by the client when support is not provided on the server.

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 the underlying file system at the server has a named attribute directory or not. Therefore, operations such as SETATTR and GETATTR on the named attribute directory are undefined.

5.1. Mandatory Attributes

These MUST be supported by every NFS version 4 client and server in order to ensure a minimum level of interoperability. The server must store and return these attributes and the client must be able to function with an attribute set limited to these attributes. With just the mandatory attributes some client functionality may be impaired or limited in some ways. A client may ask for any of these attributes to be returned by setting a bit in the GETATTR request and the server must return their value.

5.2. Recommended Attributes

These attributes are understood well enough to warrant support in the NFS version 4 protocol. However, they may not be supported on all clients and servers. A client may ask for any of these attributes to be returned by setting a bit in the GETATTR request but must handle the case where the server does not return them. A client may ask for the set of attributes the server supports and should not request
attributes the server does not support. A server should be tolerant of requests for unsupported attributes and simply not return them rather than considering the request an error. It is expected that servers will support all attributes they comfortably can and only fail to support attributes which are difficult to support in their operating environments. A server should provide attributes whenever they don’t have to "tell lies" to the client. For example, a file modification time should be either an accurate time or should not be supported by the server. This will not always be comfortable to clients but the client is better positioned 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 NFS Version 4 protocol but are accessed by string names rather than numbers and correspond to an uninterpreted stream of bytes which are stored with the file system object. The name space for these attributes may be accessed by using the OPENATTR operation. The OPENATTR operation returns a filehandle for a virtual "attribute directory" and further perusal of the name space may be done using READDIR and LOOKUP operations on this filehandle. Named attributes may then be examined or changed by normal READ and WRITE and CREATE operations on the filehandles returned from READDIR and LOOKUP. Named attributes may have attributes.

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

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

5.4. Classification of Attributes

Each of the Mandatory and Recommended attributes can be classified in one of three categories: per server, per file system, or per file system object. Note that it is possible that some per file system attributes may vary within the file system. See the "homogeneous" attribute for its definition. 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.

- The per server attribute is:
  
  `lease_time`

- The per file system attributes are:
  
  `supp_attr, fh_expire_type, link_support, symlink_support, unique_handles, aclsupport, cansettime, caseInsensitive, casePreserving, chownRestricted, filesavail, filesfree, files_total, fs_locations, homogeneous, maxfilesize, maxname, maxread, maxwrite, noTrunc, spaceavail, spacefree, space_total, time_delta, fs_status, fs_layout_type, fs_locations_info`

- 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, dir_notif_delay, dirent_notif_delay, dacl, sacl, layout_type, layout_hint, layout_blksize, layout_alignment, mdsthreshold, retention_get, retention_set, retentevt_get, retentevt_set, retention_hold, mode_set_masked`

For `quota_avail_hard`, `quota_avail_soft`, and `quota_used` see their definitions below for the appropriate classification.
5.5. Mandatory Attributes - Definitions

<table>
<thead>
<tr>
<th>name</th>
<th>#</th>
<th>Data Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>supp_attr</td>
<td>0</td>
<td>bitmap</td>
<td>READ</td>
<td>The bit vector which would retrieve all mandatory and recommended attributes that are supported for this object. The scope of this attribute applies to all objects with a matching fsid.</td>
</tr>
<tr>
<td>type</td>
<td>1</td>
<td>nfs4_ftype</td>
<td>READ</td>
<td>The type of the object (file, directory, symlink, etc.)</td>
</tr>
<tr>
<td>fh_expire_type</td>
<td>2</td>
<td>uint32</td>
<td>READ</td>
<td>Server uses this to specify filehandle expiration behavior to the client. See the section &quot;Filehandles&quot; for additional description.</td>
</tr>
<tr>
<td>change</td>
<td>3</td>
<td>uint64</td>
<td>READ</td>
<td>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 can not be updated more frequently than the resolution of time_metadata.</td>
</tr>
<tr>
<td>size</td>
<td>4</td>
<td>uint64</td>
<td>R/W</td>
<td>The size of the object in bytes.</td>
</tr>
<tr>
<td>---------</td>
<td>---</td>
<td>--------</td>
<td>-----</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>link_support</td>
<td>5</td>
<td>bool</td>
<td>READ</td>
<td>True, if the object’s file system supports hard links.</td>
</tr>
<tr>
<td>symlink_support</td>
<td>6</td>
<td>bool</td>
<td>READ</td>
<td>True, if the object’s file system supports symbolic links.</td>
</tr>
<tr>
<td>named_attr</td>
<td>7</td>
<td>bool</td>
<td>READ</td>
<td>True, if this object has named attributes. In other words, object has a non-empty named attribute directory.</td>
</tr>
<tr>
<td>fsid</td>
<td>8</td>
<td>fsid4</td>
<td>READ</td>
<td>Unique file system identifier for the file system holding this object. fsid contains major and minor components each of which are uint64.</td>
</tr>
<tr>
<td>unique_handles</td>
<td>9</td>
<td>bool</td>
<td>READ</td>
<td>True, if two distinct filehandles guaranteed to refer to two different file system objects.</td>
</tr>
<tr>
<td>lease_time</td>
<td>10</td>
<td>nfs_lease4</td>
<td>READ</td>
<td>Duration of leases at server in seconds.</td>
</tr>
<tr>
<td>rdattr_error</td>
<td>11</td>
<td>enum</td>
<td>READ</td>
<td>Error returned from getattr during readdir.</td>
</tr>
<tr>
<td>filehandle</td>
<td>19</td>
<td>nfs_fh4</td>
<td>READ</td>
<td>The filehandle of this object (primarily for readdir requests).</td>
</tr>
</tbody>
</table>

5.6. Recommended Attributes - Definitions
<table>
<thead>
<tr>
<th>name</th>
<th>#</th>
<th>Data Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>12</td>
<td>nfsace4&lt;&gt;</td>
<td>R/W</td>
<td>The access control list for the object.</td>
</tr>
<tr>
<td>aclsupport</td>
<td>13</td>
<td>uint32</td>
<td>READ</td>
<td>Indicates what types of ACLs are supported on the current file system.</td>
</tr>
<tr>
<td>archive</td>
<td>14</td>
<td>bool</td>
<td>R/W</td>
<td>True, if this file has been archived since the time of last modification.</td>
</tr>
<tr>
<td>canonical</td>
<td></td>
<td></td>
<td></td>
<td>(deprecated in favor of time_backup).</td>
</tr>
<tr>
<td>case_insensitive</td>
<td>16</td>
<td>bool</td>
<td>READ</td>
<td>True, if filename comparisons on this file system are case insensitive.</td>
</tr>
<tr>
<td>case_preserving</td>
<td>17</td>
<td>bool</td>
<td>READ</td>
<td>True, if filename case on this file system are preserved.</td>
</tr>
<tr>
<td>chown_restricted</td>
<td>18</td>
<td>bool</td>
<td>READ</td>
<td>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, &quot;root&quot; in UNIX operating environments or in Windows 2000 the &quot;Take Ownership&quot; privilege).</td>
</tr>
<tr>
<td>------------------</td>
<td>----</td>
<td>------</td>
<td>------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dacl</td>
<td>58</td>
<td>nfsacl41</td>
<td>R/W</td>
<td>Automatically inheritable access control list used for determining access to file system objects.</td>
</tr>
<tr>
<td>dir_notif_delay</td>
<td>56</td>
<td>nfstime4</td>
<td>READ</td>
<td>notification delays on directory attributes</td>
</tr>
<tr>
<td>dirent_notif_delay</td>
<td>57</td>
<td>nfstime4</td>
<td>READ</td>
<td>notification delays on child attributes</td>
</tr>
<tr>
<td>fileid</td>
<td>20</td>
<td>uint64</td>
<td>READ</td>
<td>A number uniquely identifying the file within the file system.</td>
</tr>
<tr>
<td>files_avail</td>
<td>21</td>
<td>uint64</td>
<td>READ</td>
<td>File slots available to this user on the file system containing this object - this should be the smallest relevant limit.</td>
</tr>
<tr>
<td>files_free</td>
<td>22</td>
<td>uint64</td>
<td>READ</td>
<td>Free file slots on the file system containing this object - this should be the smallest relevant limit.</td>
</tr>
<tr>
<td>files_total</td>
<td>23</td>
<td>uint64</td>
<td>READ</td>
<td>Total file slots on the file system containing this object.</td>
</tr>
<tr>
<td>fs_absent</td>
<td>60</td>
<td>bool</td>
<td>READ</td>
<td>Is current file system present or absent.</td>
</tr>
<tr>
<td>fs_layout_type</td>
<td>62</td>
<td>layouttype4&lt;&gt;</td>
<td>READ</td>
<td>Layout types available for the file system.</td>
</tr>
<tr>
<td>fs_locations</td>
<td>24</td>
<td>fs_locations</td>
<td>READ</td>
<td>Locations where this file system may be found. If the server returns NFS4ERR_MOVED as an error, this attribute MUST be supported.</td>
</tr>
<tr>
<td>fs_locations_info</td>
<td>67</td>
<td></td>
<td>READ</td>
<td>Full function file system location.</td>
</tr>
<tr>
<td>fs_status</td>
<td>61</td>
<td>fs4_status</td>
<td>READ</td>
<td>Generic file system type information.</td>
</tr>
<tr>
<td>hidden</td>
<td>25</td>
<td>bool</td>
<td>R/W</td>
<td>True, if the file is considered hidden with respect to the Windows API?</td>
</tr>
<tr>
<td>homogeneous</td>
<td>26</td>
<td>bool</td>
<td>READ</td>
<td>True, if this object’s file system is homogeneous, i.e. are per file system attributes the same for all file system’s objects.</td>
</tr>
<tr>
<td>layout_alignment</td>
<td>66</td>
<td>uint32_t</td>
<td>READ</td>
<td>Preferred alignment for layout related I/O.</td>
</tr>
<tr>
<td>layout_blksize</td>
<td>65</td>
<td>uint32_t</td>
<td>READ</td>
<td>Preferred block size for layout related I/O.</td>
</tr>
<tr>
<td>layout_hint</td>
<td>63</td>
<td>layouthint4</td>
<td>WRITE</td>
<td>Client specified hint for file layout.</td>
</tr>
<tr>
<td>layout_type</td>
<td>64</td>
<td>layouttype4&lt;&gt;</td>
<td>READ</td>
<td>Layout types available for the file.</td>
</tr>
<tr>
<td>maxfilesize</td>
<td>27</td>
<td>uint64</td>
<td>READ</td>
<td>Maximum supported file size for the file system of this object.</td>
</tr>
<tr>
<td>maxlink</td>
<td>28</td>
<td>uint32</td>
<td>READ</td>
<td>Maximum number of links for this object.</td>
</tr>
<tr>
<td>maxname</td>
<td>29</td>
<td>uint32</td>
<td>READ</td>
<td>Maximum filename size supported for this object.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Offset</td>
<td>Data Type</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>--------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>maxread</td>
<td>30</td>
<td>uint64</td>
<td>READ</td>
<td>Maximum read size supported for this object.</td>
</tr>
<tr>
<td>maxwrite</td>
<td>31</td>
<td>uint64</td>
<td>READ</td>
<td>Maximum write size supported for this object. This attribute SHOULD be supported if the file is writable. Lack of this attribute can lead to the client either wasting bandwidth or not receiving the best performance.</td>
</tr>
<tr>
<td>mdsthreshold</td>
<td>68</td>
<td>mdsthreshold4</td>
<td>READ</td>
<td>Hint to client as to when to write through the pnfs metadata server.</td>
</tr>
<tr>
<td>mimetype</td>
<td>32</td>
<td>utf8&lt;&gt;</td>
<td>R/W</td>
<td>MIME body type/subtype of this object.</td>
</tr>
<tr>
<td>mode</td>
<td>33</td>
<td>mode4</td>
<td>R/W</td>
<td>UNIX-style mode including permission bits for this object.</td>
</tr>
<tr>
<td>mode_set_masked</td>
<td>74</td>
<td>mode_masked4</td>
<td>WRITE</td>
<td>Allows setting or resetting a subset of the bits in a UNIX-style mode</td>
</tr>
<tr>
<td>Field</td>
<td>Size</td>
<td>Type</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>mounted_on_fileid</td>
<td>55</td>
<td>uint64</td>
<td>READ</td>
<td>Like fileid, but if the target filehandle is the root of a file system return the fileid of the underlying directory.</td>
</tr>
<tr>
<td>no_trunc</td>
<td>34</td>
<td>bool</td>
<td>READ</td>
<td>True, if a name longer than name_max is used, an error be returned and name is not truncated.</td>
</tr>
<tr>
<td>numlinks</td>
<td>35</td>
<td>uint32</td>
<td>READ</td>
<td>Number of hard links to this object.</td>
</tr>
<tr>
<td>owner</td>
<td>36</td>
<td>utf8&lt;&gt;</td>
<td>R/W</td>
<td>The string name of the owner of this object.</td>
</tr>
<tr>
<td>owner_group</td>
<td>37</td>
<td>utf8&lt;&gt;</td>
<td>R/W</td>
<td>The string name of the group ownership of this object.</td>
</tr>
<tr>
<td>quota_avail_hard</td>
<td>38</td>
<td>uint64</td>
<td>READ</td>
<td>For definition see &quot;Quota Attributes&quot; section below.</td>
</tr>
<tr>
<td>quota_avail_soft</td>
<td>39</td>
<td>uint64</td>
<td>READ</td>
<td>For definition see &quot;Quota Attributes&quot; section below.</td>
</tr>
<tr>
<td>quota_used</td>
<td>40</td>
<td>uint64</td>
<td>READ</td>
<td>For definition see &quot;Quota Attributes&quot; section below.</td>
</tr>
<tr>
<td>rawdev</td>
<td>41</td>
<td>specdata4</td>
<td>READ</td>
<td>Raw device identifier. UNIX device major/minor node information. If the value of type is not NF4BLK or NF4CHR, the value return SHOULD NOT be considered useful.</td>
</tr>
<tr>
<td>----------</td>
<td>----</td>
<td>-----------------</td>
<td>------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>retentevt_get</td>
<td>71</td>
<td>Get the event-based retention duration, and if enabled, the event-based retention begin time of the file object. GETATTR use only.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>retention_get4</td>
<td>READ</td>
<td></td>
</tr>
<tr>
<td>retentevt_set</td>
<td>72</td>
<td>retention_set4</td>
<td>WRITE</td>
<td>Set the event-based retention duration, and optionally enable event-based retention on the file object. SETATTR use only.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Offset</td>
<td>Structure</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------</td>
<td>-----------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>retention_get</td>
<td>69</td>
<td>retention_get4</td>
<td>READ</td>
<td>Get the retention duration, and if enabled, the retention begin time of the file object. GETATTR use only.</td>
</tr>
<tr>
<td>retention_hold</td>
<td>73</td>
<td>uint64_t</td>
<td>R/W</td>
<td>Get or set administrative retention holds, one hold per bit position.</td>
</tr>
<tr>
<td>retention_set</td>
<td>70</td>
<td>retention_set4</td>
<td>WRITE</td>
<td>Set the retention duration, and optionally enable retention on the file object. SETATTR use only.</td>
</tr>
<tr>
<td>sacl</td>
<td>59</td>
<td>nfsacl41</td>
<td>R/W</td>
<td>Automatically inheritable access control list used for auditing access to files.</td>
</tr>
<tr>
<td>space_avail</td>
<td>42</td>
<td>uint64</td>
<td>READ</td>
<td>Disk space in bytes available to this user on the file system containing this object - this should be the smallest relevant limit.</td>
</tr>
<tr>
<td>Name</td>
<td>Column</td>
<td>Type</td>
<td>Mode</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------</td>
<td>--------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>space_free</td>
<td>43</td>
<td>uint64</td>
<td>READ</td>
<td>Free disk space in bytes on the file system containing this object. This should be the smallest relevant limit.</td>
</tr>
<tr>
<td>space_total</td>
<td>44</td>
<td>uint64</td>
<td>READ</td>
<td>Total disk space in bytes on the file system containing this object.</td>
</tr>
<tr>
<td>space_used</td>
<td>45</td>
<td>uint64</td>
<td>READ</td>
<td>Number of file system bytes allocated to this object.</td>
</tr>
<tr>
<td>system</td>
<td>46</td>
<td>bool</td>
<td>R/W</td>
<td>True, if this file is a &quot;system&quot; file with respect to the Windows API?</td>
</tr>
<tr>
<td>time_access</td>
<td>47</td>
<td>nfstime4</td>
<td>READ</td>
<td>The time of last access to the object by a read that was satisfied by the server.</td>
</tr>
<tr>
<td>time_access_set</td>
<td>48</td>
<td>settime4</td>
<td>WRITE</td>
<td>Set the time of last access to the object. SETATTR use only.</td>
</tr>
<tr>
<td>time_backup</td>
<td>49</td>
<td>nfstime4</td>
<td>R/W</td>
<td>The time of last backup of the object.</td>
</tr>
</tbody>
</table>
### 5.7. Time Access

As defined above, the `time_access` attribute represents the time of last access to the object by a read that was satisfied by the server. The notion of what is an "access" depends on server’s operating environment and/or the server’s file system semantics. For example, for servers obeying POSIX semantics, `time_access` would be updated only by the `READLINK`, `READ`, and `REaddir` operations and not any of the operations that modify the content of the object. 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 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. Interpreting owner and owner_group

The recommended attributes "owner" and "owner_group" (and also users and groups within the "acl" attribute) are represented in terms of a UTF-8 string. To avoid a representation that is tied to a particular underlying implementation at the client or server, the use of the UTF-8 string has been chosen. Note that section 6.1 of RFC2624 [27] provides additional rationale. It is expected that the client and server will have their own local representation of owner and owner_group that is used for local storage or presentation to the end user. Therefore, it is expected that when these attributes are transferred between the client and server that the local representation is translated to a syntax of the form "user@dns_domain". This will allow for a client and server that do not use the same local representation the ability to translate to a common syntax that can be interpreted by both.

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 between a numeric id to the user@dns_domain syntax. A name service may also be used to accomplish the translation. A server may provide a more general service, not limited by any particular translation (which would only translate a limited set of possible strings) by storing the owner and owner_group attributes in local storage without any translation or it may augment a translation method by storing the entire string for attributes for which no translation is available while using the local representation for those cases in which a translation is available.

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

The "dns_domain" portion of the owner string is meant to be a DNS domain name. For example, user@ietf.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.

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

To provide a greater degree of compatibility with previous versions of NFS (i.e. v2 and v3), which identified users and groups by 32-bit unsigned uid’s and gid’s, owner and group strings that consist of decimal numeric values with no leading zeros can be given a special interpretation by clients and servers which 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 a v2/v3 uid or gid having the corresponding numeric value. A server is not obligated to accept such a string, but may return an NFS4ERR_BADOWNER instead. To avoid this mechanism being used to subvert user and group translation, so that a client might pass all of the owners and groups in numeric form, a 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 name@domain string and not the special form for compatibility.

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.
5.9. Character Case Attributes

With respect to the case_insensitive and case_preserving attributes, each UCS-4 character (which UTF-8 encodes) has a "long descriptive name" [RFC1345] which may or may not include the word "CAPITAL" or "SMALL". The presence of SMALL or CAPITAL allows an NFS server to implement unambiguous and efficient table driven mappings for case insensitive comparisons, and non-case-preserving storage. For general character handling and internationalization issues, see the section "Internationalization".

5.10. Quota Attributes

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

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

quota_avail_hard The value in bytes which 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.

quota_used The value in bytes which represent 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 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".
5.11. mounted_on_fileid

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 stat() would have returned before any file systems were mounted on the mount point.

Unlike NFS version 3, NFS version 4 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 NFS version 4 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.

5.12. Directory Notification Attributes

As described in Section 17.39, the client can request a minimum delay for notifications of changes to attributes, but the server is free to ignore what the client requests. The client can determine in advance what notification delays the server will accept by issuing a GETATTR for either or both of two directory notification attributes. When the client calls the GET_DIR_DELEGATION operation and asks for attribute change notifications, it should request notification delays that are no less than the values in the server-provided attributes.

5.12.1. dir_notif_delay

The dir_notif_delay attribute is the minimum number of seconds the server will delay before notifying the client of a change to the directory’s attributes.

5.12.2. dirent_notif_delay

The dirent_notif_delay attribute is the minimum number of seconds the server will delay before notifying the client of a change to a file object that has an entry in the directory.

5.13. PNFS Attributes

5.13.1. fs_layout_type

The fs_layout_type attribute (data type layouttype4, see Section 3.2.15) applies to a file system and indicates what layout types are supported by the file system. This attribute is expected to be queried when a client encounters a new fsid. This attribute is used by the client to determine if it supports the layout type.

5.13.2. layout_alignment

The layout_alignment attribute indicates the preferred alignment for I/O to files on the file system the client has layouts for. Where possible, the client should issue READ and WRITE operations with offsets that are whole multiples of the layout_alignment attribute.
5.13.3. layout_blksize

The layout_blksize attribute indicates the preferred block size for I/O to files on the file system the client has layouts for. Where possible, the client should issue READ operations with a count argument that is a whole multiple of layout_blksize, and WRITE operations with a data argument of size that is a whole multiple of layout_blksize.

5.13.4. layout_hint

The layout_hint attribute (data type layouthint4, see Section 3.2.22) may be set on newly created files to influence the metadata server’s choice for the file’s layout. It is suggested that this attribute is set as one of the initial attributes within the OPEN call. The metadata server may ignore this attribute. This attribute is a subset of the layout structure returned by LAYOUTGET. For example, instead of specifying particular devices, this would be used to suggest the stripe width of a file. It is up to the server implementation to determine which fields within the layout it uses.

5.13.5. layout_type

This attribute indicates the particular layout type(s) used for a file. This is for informational purposes only. The client needs to use the LAYOUTGET operation in order to get enough information (e.g., specific device information) in order to perform I/O.

5.13.6. mdsthreshold

This attribute acts as a hint to the client to help it determine when it is more efficient to issue read and write requests to the metadata server vs. the data server. Two types of thresholds are described: file size thresholds and I/O size thresholds. If a file’s size is smaller than the file size threshold, data accesses should be issued to the metadata server. If an I/O is below the I/O size threshold, the I/O should be issued to the metadata server. Each threshold can be specified independently for read and write requests. For either threshold type, a value of 0 indicates no read or write should be issued to the metadata server, while a value of all 1s indicates all reads or writes should be issued to the metadata server.

The attribute is available on a per filehandle basis. If the current filehandle refers to a non-pNFS file or directory, the metadata server should return an attribute that is representative of the filehandle’s file system. It is suggested that this attribute is queried as part of the OPEN operation. Due to dynamic system changes, the client should not assume that the attribute will remain
constant for any specific time period, thus it should be periodically refreshed.

5.14. Retention Attributes

Retention is a concept whereby a file object can be placed in an immutable, undeletable, unrenamable state for a fixed or infinite duration of time. Once in this "retained" state, the file cannot be moved out of the state until the duration of retention has been reached.

When retention is enabled, retention MUST extend to the data of the file, and the name of file. The server MAY extend retention any other property of the file, including any subset of mandatory, recommended, and named attributes, with the exceptions noted in this section.

Servers MAY support or not support retention on any file object type.

There are five retention attributes:

- retention_get. This attribute is only readable via GETATTR and not setable via SETATTR. The value of the attribute consists of:

  ```
  const RET4_DURATION_INFINITE = 0xffffffffffffffff;
  struct retention_get4 {
    uint64_t rg_duration;
    nfstime4 rg_begin_time<1>;
  };
  ```

  The field rg_duration is duration in seconds indicating how long the file will be retained once retention is enabled. The field rg_begin_time is an array of up to one absolute time value. If the array is zero length, no beginning retention time has been established, and retention is not enabled. If rg_duration is equal to RET4_DURATION_INFINITE, the file, once retention is enabled, will be retained for an infinite duration.

- retention_set. This attribute corresponds to retention_get. This attribute is only setable via SETATTR and not readable via GETATTR. The value of the attribute consists of:

  ```
  struct retention_set4 {
    bool rs_enable;
    uint64_t rs_duration<1>;
  };
  ```
If the client sets rs_enable to TRUE, then it is enabling retention on the file object with the begin time of retention commencing from the server's current time and date. The duration of the retention can also be provided if the rs_duration array is of length one. The duration is time in seconds from the begin time of retention, and if set to RET4_DURATION_INFINITE, the file is to be retained forever. If retention is enabled, with no duration specified in either this SETATTR or a previous SETATTR, the duration defaults to zero seconds. The server MAY restrict the enabling of retention or the duration of retention on the basis of the ACE4_WRITE RETENTION ACL permission. The enabling of retention does not prevent the enabling of event-based retention nor the modification of the retention_hold attribute.

- **retentevt_get.** This attribute is like retention_get, but refers to event-based retention. The event that triggers event-based retention is not defined by the NFSv4.1 specification.

- **retentevt_set.** This attribute corresponds to retentevt_get, is like retention_set, but refers to event-based retention. When event-based retention is set, the file MUST be retained even if non-event-based retention has been set, and the duration of non-event-based retention has been reached. Conversely, when non-event-based retention has been set, the file MUST be retained even the event-based retention has been set, and the duration of event-based retention has been reached. The server MAY restrict the enabling of event-based retention or the duration of event-based retention on the basis of the ACE4_WRITE_RETENTION ACL permission. The enabling of event-based retention does not prevent the enabling of non-event-based retention nor the modification of the retention_hold attribute.

- **retention_hold.** This attribute allows one to 64 administrative holds, one hold per bit on the attribute. If retention_hold is not zero, then the file MUST NOT be deleted, renamed, or modified, even if the duration on enabled event or non-event-based retention has been reached. The server MAY restrict the modification of retention_hold on the basis of the ACE4_WRITE RETENTION_HOLD ACL permission. The enabling of administration retention holds does not prevent the enabling of event-based or non-event-based retention.
6. Access Control Lists

Access Control Lists (ACLs) are a file attribute that specify fine
grained access control. This chapter covers the "acl", "dacl",
"sacl", "aclsupport", "mode", "mode_set_masked" file attributes, and
their interactions.

6.1. Goals

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

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

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

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

- On servers that support the mode attribute, if the ACL attribute
  has 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
      should fail, regardless of a previously existing or inherited
      ACL.

- This minor version of NFSv4 should not introduce significantly
different semantics relating to the mode and ACL attributes, nor
should it render invalid any existing implementations. Rather,
this chapter provides clarifications based on previous
implementations and discussions around them.

- If a server supports the ACL attribute, then at any time, the
  server can provide an ACL attribute when requested. The ACL
  attribute will describe all permissions on the file object, except
for the three high-order bits of the mode attribute (described in Section 6.2.3). The ACL attribute will not conflict with the mode attribute, on servers that support the mode attribute.

- If a server supports the mode attribute, then at any time, the server can provide a mode attribute when requested. The mode attribute will not conflict with the ACL attribute, on servers that support the ACL attribute.

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

6.2. File Attributes Discussion

6.2.1. ACL Attribute

The NFS version 4 ACL attribute is an array of access control entries (ACEs). 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 attribute is defined as follows:

```c
typedef uint32_t  acetype4;
typedef uint32_t  aceflag4;
typedef uint32_t  acemask4;

struct nfsace4 {
    acetype4       type;
    aceflag4       flag;
    acemask4       access_mask;
    utf8str混合  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, all AUDIT and ALARM ACEs are processed until end of the ACL.

The NFS version 4 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 indicate that it supports ACLs as long as it follows the guidelines for mapping between its ACL model and the NFS version 4 ACL model.

The situation is complicated by the fact that a server may have multiple modules that enforce ACLs. For example, the enforcement for NFS version 4 access may be different from the enforcement for local access, and both may be different from the enforcement for access through other protocols such as 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 in all cases is that the server must not accept ACLs that appear to make the file more secure than it really is.

6.2.1.1. ACE Type

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

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

+-------------------------------+--------------+---------------------+
<table>
<thead>
<tr>
<th>Value</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE4_ACCESS_ALLOWED_ACE_TYPE</td>
<td>ALLOW</td>
<td>Explicitly grants</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the access defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in acemask4 to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>file or directory.</td>
</tr>
<tr>
<td>ACE4_ACCESS_DENIED_ACE_TYPE</td>
<td>DENY</td>
<td>Explicitly denies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the access defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in acemask4 to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>file or directory.</td>
</tr>
<tr>
<td>ACE4_SYSTEM_AUDIT_ACE_TYPE</td>
<td>AUDIT</td>
<td>LOG (system dependent) any access attempt to a file or directory which uses any of the access methods specified in acemask4.</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ACE4_SYSTEM_ALARM_ACE_TYPE</td>
<td>ALARM</td>
<td>Generate a system ALARM (system dependent) when any access attempt is made to a file or directory for the access methods specified in acemask4.</td>
</tr>
</tbody>
</table>

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

6.2.1.2. The aclsupport Attribute

A server need not support all of the above ACE types. 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;
```

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.

Example: suppose a server can enforce NFS ACLs for NFS access but cannot enforce ACLs for local access. If arbitrary processes can run on the server, then the server SHOULD NOT indicate ACL support. On the other hand, if only trusted administrative programs run locally, then the server may indicate ACL support.
6.2.1.3. ACE Access Mask

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

```c
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_WRITE_RETENTION      = 0x00000200;
const ACE4_WRITE_RETENTION_HOLD = 0x00000400;
const ACE4_DELETE               = 0x00010000;
const ACE4_READ_ACL             = 0x00020000;
const ACE4_WRITE_ACL            = 0x00040000;
const ACE4_WRITE_OWNER          = 0x00080000;
const ACE4_SYNCHRONIZE          = 0x00100000;
```

6.2.1.3.1. Discussion of Mask Attributes

**ACE4_READ_DATA**

Operation(s) affected:
- READ
- OPEN

Discussion:
Permission to read the data of the file.

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

**ACE4_LIST_DIRECTORY**

Operation(s) affected:
- READDIR

Discussion:
Permission to list the contents of a directory.

**ACE4_WRITE_DATA**

Operation(s) affected:
- WRITE
- OPEN
- SETATTR of size
Discussion:
Permission to modify a file’s data anywhere in the file’s offset range. This includes the ability to write to any arbitrary offset and as a result to grow the file.

ACE4_ADD_FILE
Operation(s) affected:
CREATE
OPEN
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.

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

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:
LOOKUP
READ
OPEN
Discussion:
Permission to execute a file or traverse/search a directory.

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

As an example, following a SETATTR of the following ACL:
   nfsuser:ACE4_EXECUTE:ALLOW
A subsequent GETATTR of ACL for that file SHOULD return:
   nfsuser:ACE4_EXECUTE:ALLOW
Rather than:
   nfsuser:ACE4_EXECUTE/ACE4_READ_DATA:ALLOW

ACE4_DELETE_CHILD
Operation(s) affected:
REMOVE
Discussion:
Permission to delete a file or directory within a directory. See section "ACE4_DELETE vs. ACE4_DELETE_CHILD" for information on how these two access mask bits interact.

ACE4_READ_ATTRIBUTES
Operation(s) affected:
GETATTR of file system object attributes
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.

ACE4_WRITE_ATTRIBUTES
Operation(s) affected:
SEATTR 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 permission, but lacking ACE4_WRITE_ATTRIBUTES must be allowed to implicitly set the times associated with a file.

ACE4_WRITE_RETENTION
Operation(s) affected:
SEATTR of retention_set, retentevt_set.
Discussion:
Permission to modify the durations of event and non-event-based retention. Also permission to enable event and non-event-based retention. A server MAY map ACE4_WRITE_ATTRIBUTES to ACE_WRITE_RETENTION.

ACE4_WRITE_RETENTION_HOLD
Operation(s) affected:
SEATTR of retention_hold.
Discussion:
Permission to modify the administration retention holds. A server MAY map ACE4_WRITE_ATTRIBUTES to ACE_WRITE_RETENTION_HOLD.

ACE4_DELETE
Operation(s) affected:
REMOVE
Discussion:
Permission to delete the file or directory. See section
"ACE4_DELETE vs. ACE4_DELETE_CHILD" for information on how these two access mask bits interact.

ACE4_READ_ACL
Operation(s) affected:
GETATTR of acl
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
Discussions:
Permission to write the owner and owner_group attributes.
On UNIX systems, this is the ability to execute chown().

ACE4_SYNCHRONIZE
Operation(s) affected:
NONE
Discussion:
Permission to access file locally at the server with synchronized reads and writes.

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 error in the direction of more restricted access. For example, suppose a server cannot distinguish overwriting data from appending new data, as described in the previous paragraph. If a client submits an ACE where ACE4_APPEND_DATA is set but ACE4_WRITE_DATA is not (or vice versa), the server should reject the request with NFS4ERR_ATTRNOTSUPP. Nonetheless, if the ACE has type DENY, the server may silently turn on the other bit, so that both ACE4_APPEND_DATA and ACE4_WRITE_DATA are denied.
6.2.1.3.2. ACE4_DELETE vs. ACE4_DELETE_CHILD

Two access mask bits govern the ability to delete a file or directory object: ACE4_DELETE on the object itself, and ACE4_DELETE_CHILD on the object’s parent directory.

Many systems also consult the "sticky bit" (MODE4_SVTX) and write mode bit on the parent directory when determining whether to allow a file to be deleted. The mode bit for write corresponds to ACE4_WRITE_DATA, which is the same physical bit as ACE4_ADD_FILE. Therefore, ACE4_ADD_FILE can come into play when determining permission to delete.

In the algorithm below, the strategy is that ACE4_DELETE and ACE4_DELETE_CHILD take precedence over the sticky bit, and the sticky bit takes precedence over the "write" mode bits (reflected in ACE4_ADD_FILE).

Server implementations SHOULD grant or deny permission to delete based on the following algorithm.

```plaintext
if ACE4_EXECUTE is denied by the parent directory ACL:
    deny delete
else if ACE4_DELETE is allowed by the target object ACL:
    allow delete
else if ACE4_DELETE_CHILD is allowed by the parent directory ACL:
    allow delete
else if ACE4_DELETE_CHILD is denied by the parent directory ACL:
    deny delete
else if ACE4_ADD_FILE is allowed by the parent directory ACL:
    if MODE4_SVTX is set for the parent directory:
        if the principal owns the parent directory OR
        the principal owns the target object OR
        ACE4_WRITE_DATA is allowed by the target object ACL:
            allow delete
        else:
            deny delete
    else:
        allow delete
else:
    deny delete
```
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;
- const ACE4_INHERITED_ACE = 0x00000080;

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. Again, the guiding principle is that the file not appear to be more secure than it really is.

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

ACE4_DIRECTORY_INHERIT_ACE
Can be placed on a directory and indicates that this ACE should be added to each new directory created.

ACE4_INHERIT_ONLY_ACE
Can be placed on a directory but does not apply to the directory; 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.
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_INHERITED_ACE
Indicates that this ACE is inherited from a parent directory. A server that supports automatic inheritance will place this flag on any ACEs inherited from the parent directory when creating a new object. Client applications will use this to perform automatic inheritance. Clients and servers MUST clear this bit in the acl attribute; it may only be used in the dacl and sacl attributes.

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 relate only to 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 that of the ACCESS operation as well, the difference being that "success" or "failure" does not mean whether ACCESS returns NFS4_OK or not. Success means whether ACCESS returns all requested and supported bits. Failure means whether ACCESS failed 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.

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

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

6.2.1.5.1.  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.  dacl and sacl Attributes

The dacl and sacl attributes are like the acl attribute, but dacl and sacl each allow only certain types of ACEs. The dacl attribute allows just ALLOW and DENY ACEs. The sacl attribute allows just
AUDIT and ALARM ACEs. The dacl and sacl attributes also have improved support for automatic inheritance (see Section 6.4.3.2). The separation of ACE types and inheritance support make dacl and sacl a better choice (over acl) for clients when setting ACEs on a file.

6.2.3. mode Attribute

The NFS version 4 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 README operation, and it MUST return NFS4ERR_INVAL if bits other than those defined above are set in a SETATTR, CREATE, or OPEN operation.

6.2.4. mode_set_masked Attribute

The mode_set_masked attribute is a write-only attribute that allows individual bits in the mode attribute to be set or reset, without changing others. It allows, for example, the bits MODE4_SUID, MODE4_SGID, and MODE4_SVTX to be modified while leaving unmodified any of the nine low-order mode bits devoted to permissions.

The mode_set_masked attribute consists of two words each in the form of a mode4. The first consists of the value to be applied to the
current mode value and the second is a mask. Only bits set to one in
the mask word are changed (set or reset) in the file’s mode. All
other bits in the mode remain unchanged. Bits in the first word that
correspond to bits which are zero in the mask are ignored, except
that undefined bits are checked for validity and can result in
NFSERR_INVAL as described below.

The mode_set_masked attribute is only valid in a SETATTR operation.
If it is used in a CREATE or OPEN operation, the server MUST return
NFS4ERR_INVAL.

Bits not defined as valid in the mode attribute are not valid in
either word of the mode_set_masked attribute. The server MUST return
NFS4ERR_INVAL if any of those are on in a SETATTR. If the mode and
mode_set_masked attributes are both specified in the same SETATTR,
the server MUST also return NFS4ERR_INVAL.

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:

- 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.

- Server implementations MAY grant ACE4_WRITE_ACL and ACE4_READ_ACL
permissions in order to prevent the owner from getting into the
situation where they can’t ever modify the ACL.

- 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).

- 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.
6.3.1.2. Client Considerations

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

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

6.3.2. Computing a Mode Attribute from an ACL

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

1. To determine MODE4_ROTH, MODE4_WOTH, and MODE4_XOTH:

   1. If the special identifier EVERYONE@ is granted ACE4_READ_DATA, then the bit MODE4_ROTH SHOULD be set. Otherwise, MODE4_ROTH SHOULD NOT be set.

   2. If the special identifier EVERYONE@ is granted ACE4_WRITE_DATA or ACE4_APPEND_DATA, then the bit MODE4_WOTH SHOULD be set. Otherwise, MODE4_WOTH SHOULD NOT be set.

   3. If the special identifier EVERYONE@ is granted ACE4_EXECUTE, then the bit MODE4_XOTH SHOULD be set. Otherwise, MODE4_XOTH SHOULD NOT be set.

2. To determine MODE4_RGRP, MODE4_WGRP, and MODE4_XGRP, note that the EVERYONE@ special identifier SHOULD be taken into account. In other words, when determining if the GROUP@ special identifier is granted a permission, ACEs with the identifier EVERYONE@ should take effect just as ACEs with the special identifier GROUP@ would.

   1. If the special identifier GROUP@ is granted ACE4_READ_DATA, then the bit MODE4_RGRP SHOULD be set. Otherwise, MODE4_RGRP SHOULD NOT be set.
2. If the special identifier GROUP@ is granted ACE4_WRITE_DATA or ACE4_APPEND_DATA, then the bit MODE4_WGRP SHOULD be set. Otherwise, MODE4_WGRP SHOULD NOT be set.

3. If the special identifier GROUP@ is granted ACE4_EXECUTE, then the bit MODE4_XGRP SHOULD be set. Otherwise, MODE4_XGRP SHOULD NOT be set.

3. To determine MODE4_RUSR, MODE4_WUSR, and MODE4_XUSR, note that the EVERYONE@ special identifier SHOULD be taken into account. In other words, when determining if the OWNER@ special identifier is granted a permission, ACEs with the identifier EVERYONE@ should take effect just as ACEs with the special identifier OWNER@ would.

1. If the special identifier OWNER@ is granted ACE4_READ_DATA, then the bit MODE4_RUSR SHOULD be set. Otherwise, MODE4_RUSR SHOULD NOT be set.

2. If the special identifier OWNER@ is granted ACE4_WRITE_DATA or ACE4_APPEND_DATA, then the bit MODE4_WUSR SHOULD be set. Otherwise, MODE4_WUSR SHOULD NOT be set.

3. If the special identifier OWNER@ is granted ACE4_EXECUTE, then the bit MODE4_XUSR SHOULD be set. Otherwise, MODE4_XUSR SHOULD NOT be set.

6.3.2.1. Discussion

The nine low-order mode bits (MODE4_R*, MODE4_W*, MODE4_X*) correspond to ACE4_READ_DATA, ACE4_WRITE_DATA/ACE4_APPEND_DATA, and ACE4_EXECUTE for OWNER@, GROUP@, and EVERYONE@. On some implementations, mode bits may represent a superset of these permissions, e.g. if a specific user is granted ACE4_WRITE_DATA, then MODE4_WGRP will be set, even though the file’s owner_group is not granted ACE4_WRITE_DATA.

Server implementations are discouraged from doing this, as experience has shown that this is confusing and annoying to end users. The specifications above also discourage this practice to enforce the semantic that setting the mode attribute effectively specifies read, write, and execute for owner, group, and other.

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.

In this section, much is made of the methods in Section 6.3.2. Many requirements refer to this section. 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 (1003.1e draft 17), 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 mode permission bits are subject to change, either because the mode attribute was set or because the mode_set_masked attribute was set and the mask included one or more bits from the low-order nine mode bits that control permissions, and the ACL attribute is not explicitly set, the ACL attribute must be modified in accordance with the updated value of the permissions bits within the mode. This must happen even if the value of the permission bits within the mode is the same after the mode is set as before.

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 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 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 attribute.

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 an ACL attribute and not setting the mode or mode_set_masked attributes, 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 (includes use of either the mode attribute or the mode_set_masked attribute) and the ACL attribute in the same operation, the attributes MUST be applied in this order: mode (or mode_set_masked), 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 the ACL attribute.

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 the ACL attribute, it may use the ACL attribute 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.

Implementors should standardize on what the behavior of CREATE and OPEN must be depending on the presence or absence of the mode and ACL attributes.

1. If just mode is given:

   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 ACL is given:

   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 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 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. It is worth
noting that if no inheritable ACEs exist on the parent directory,
the file will be created with an empty ACL, thus granting no
access.

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.

If when a new directory is created and it inherits ACEs from its
parent, for each inheritable ACE which affects the directory’s
permissions, a server MAY create two ACEs on the directory being
created; one effective and one which is only inheritable (i.e. has
ACE4_INHERIT_ONLY_ACE flag set). This gives the user and the server,
in the cases which it must mask certain permissions upon creation,
the ability to modify the effective permissions without modifying the
ACE which is to be inherited to the new directory’s children.

When a newly created object is created with attributes, and those
attributes contain an ACL attribute and/or a mode attribute, the
server MUST apply those attributes to the newly created object, as
described in Section 6.4.1.

6.4.3.2. Automatic Inheritance

Unlike the acl attribute, the sacl and dacl (see Section 6.2.2)
attributes both have an additional flag field. The flag field
applies to the entire sacl or dacl; three flag values are defined

const ACL4_AUTO_INHERIT = 0x00000001;
const ACL4_PROTECTED    = 0x00000002;
const ACL4_DEFAULTED    = 0x00000004;

and all other bits must be cleared. The ACE4_INHERITED_ACE flag may
be set in the ACEs of the sacl or dacl (whereas it must always be
cleared in the acl).
Together these features allow a server to support automatic inheritance, which we now explain in more detail.

Inheritable ACEs are normally inherited by child objects only at the time that the child objects are created; later modifications to inheritable ACEs do not result in modifications to inherited ACEs on descendents.

However, the dacl and sacl provide an optional mechanism which allows a client application to propagate changes to inheritable ACEs to an entire directory hierarchy.

A server that supports this performs inheritance at object creation time in the normal way, but also sets the ACE4_INHERITED_ACE flag on any inherited ACEs as they are added to the new object.

A client application such as an ACL editor may then propagate changes to inheritable ACEs on a directory by recursively traversing that directory’s descendents and modifying each ACL encountered to remove any ACEs with the ACE4_INHERITED_ACE flag and to replace them by the new inheritable ACEs (also with the ACE4_INHERITED_ACE flag set). It uses the existing ACE inheritance flags in the obvious way to decide which ACEs to propagate. (Note that it may encounter further inheritable ACEs when descending the directory hierarchy, and that those will also need to be taken into account when propagating inheritable ACEs to further descendents.)

The reach of this propagation may be limited in two ways: first, automatic inheritance is not performed from any directory ACL that has the ACL4_AUTO_INHERIT flag cleared; and second, automatic inheritance stops wherever an ACL with the ACL4_PROTECTED flag is set, preventing modification of that ACL and also (if the ACL is set on a directory) of the ACL on any of the object’s descendents.

This propagation is performed independently for the sacl and the dacl attributes; thus the ACL4_AUTO_INHERIT and ACL4_PROTECTED flags may be independently set for the sacl and the dacl, and propagation of one type of acl may continue down a hierarchy even where propagation of the other acl has stopped.

New objects should be created with a dacl and a sacl that both have the ACL4_PROTECTED flag cleared and the ACL4_AUTO_INHERIT flag set to the same value as that on, respectively, the sacl or dacl of the parent object.

Both the dacl and sacl attributes are RECOMMENDED, and a server may support one without supporting the other.
A server that supports both the old acl attribute and one or both of the new dacl or sacl attributes must do so in such a way as to keep all three attributes consistent with each other. Thus the ACEs reported in the acl attribute should be the union of the ACEs reported in the dacl and sacl attributes, except that the ACE4_INHERITED_ACE flag must be cleared from the ACEs in the acl. And of course a client that queries only the acl will be unable to determine the values of the sacl or dacl flag fields.

When a client performs a SETATTR for the acl attribute, the server SHOULD set the ACL4_PROTECTED flag to true on both the sacl and the dacl. By using the acl attribute, as opposed to the dacl or sacl attributes, the client signals that it may not understand automatic inheritance, and thus cannot be trusted to set an ACL for which automatic inheritance would make sense.

When a client application queries an ACL, modifies it, and sets it again, it should leave any ACEs marked with ACE4_INHERITED_ACE unchanged, in their original order, at the end of the ACL. If the application is unable to do this, it should set the ACL4_PROTECTED flag. This behavior is not enforced by servers, but violations of this rule may lead to unexpected results when applications perform automatic inheritance.

If a server also supports the mode attribute, it SHOULD set the mode in such a way that leaves inherited ACEs unchanged, in their original order, at the end of the ACL. If it is unable to do so, it SHOULD set the ACL4_PROTECTED flag on the file’s dacl.

Finally, in the case where the request that creates a new file or directory does not also set permissions for that file or directory, and there are also no ACEs to inherit from the parent’s directory, then the server’s choice of ACL for the new object is implementation-dependent. In this case, the server SHOULD set the ACL4_DEFAULTED flag on the ACL it chooses for the new object. An application performing automatic inheritance takes the ACL4_DEFAULTED flag as a sign that the ACL should be completely replaced by one generated using the automatic inheritance rules.

7. Single-server Name Space

This chapter describes the NFSv4 single-server name space. Single-server namespaces may be presented directly to clients, or they may be used as a basis to form larger multi-server namespaces (e.g. site-wide or organization-wide) to be presented to clients, as described in Section 10.
7.1. Server Exports

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

7.2. Browsing Exports

The NFS version 4 protocol provides a root filehandle that clients can use to obtain filehandles for the exports of a particular server, via a series of LOOKUP operations within a COMPOUND, to traverse a path. A common user experience is to use a graphical user interface (perhaps a file "Open" dialog window) to find a file via progressive browsing through a directory tree. The client must be able to move from one export to another export via single-component, progressive LOOKUP operations.

This style of browsing is not well supported by the NFS version 2 and 3 protocols. The client expects all LOOKUP operations to remain within a single server file system. For example, the device attribute will not change. This prevents a client from taking name space paths that span exports.

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

7.3. Server Pseudo File System

NFS version 4 servers avoid this name space inconsistency by presenting all the exports for a given server within the framework of a single namespace, for that server. An NFS version 4 client uses LOOKUP and READDIR operations to browse seamlessly from one export to
another. Portions of the server name space that are not exported are bridged via a "pseudo file system" that provides a view of exported directories only. A pseudo file system has a unique fsid and behaves like a normal, read only file system.

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

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

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

7.4. Multiple Roots

The DOS and Windows operating environments are sometimes described as having "multiple roots". File Systems are commonly represented as disk letters. MacOS represents file systems as top level names. NFS version 4 servers for these platforms can construct a pseudo file system above these root names so that disk letters or volume names are simply directory names in the pseudo root.

7.5. Filehandle Volatility

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

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

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

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

It is the server’s responsibility to present the pseudo file system
that is complete to the client. If the client sends a lookup request
for the path "/a/b/c/d", the server’s response is the filehandle of
the file system "/a/b/c/d". In previous versions of the NFS
protocol, the server would respond with the filehandle of directory
"/a/b/c/d" within the file system "/a/b".

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

7.8. Security Policy and Name Space Presentation

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

As suggested practice, the server should apply the security policy of

a shared resource in the server’s namespace to the components of the resource’s ancestors. For example:

/  
/a/b  
/a/b/c  

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

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

8. File Locking and Share Reservations

Integrating locking into the NFS protocol necessarily causes it to be stateful. With the inclusion of such features as share reservations, file and directory delegations, recallable layouts, and support for mandatory record locking the protocol becomes substantially more dependent on state than the traditional combination of NFS and NLM [XNFS]. There are three components to making this state manageable:

- Clear division between client and server
- Ability to reliably detect inconsistency in state between client and server
- Simple and robust recovery mechanisms

In this model, the server owns the state information. The client requests changes in locks and the server responds with the changes made. Non-client-initiated changes in locking state are infrequent and the client receives prompt notification of them and can adjust its view of the locking state to reflect the server’s changes.

To support Win32 share reservations it is necessary to provide operations which atomically OPEN or CREATE files. Having a separate share/unshare operation would not allow correct implementation of the Win32 OpenFile API. In order to correctly implement share semantics, the previous NFS protocol mechanisms used when a file is opened or created (LOOKUP, CREATE, ACCESS) need to be replaced. The NFS version 4.1 protocol defines OPEN operation which looks up or creates
a file and establishes locking state on the server.

8.1. Locking

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

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

8.1.1. Client and Session ID

A client must establish a client ID (see Section 2.4) and then one or more sessionids (see Section 2.10) before performing any operations to open, lock, or delegate a file object. The sessionid services as a shorthand referral to an NFSv4.1 client.

8.1.2. State-owner Definition

When opening a file or requesting a record lock, the client must specify an identifier which represents the owner of the requested lock. This identifier is in the form of a state-owner, represented in the protocol by a state_owner4, a variable-length opaque array which, when concatenated with the current client ID uniquely defines the owner of lock managed by the client. This may be a thread id, process id, or other unique value.

Owners of opens and owners of record 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).

Each open is associated with a specific open-owner while each record 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. Delegations and layouts, on the other hand, are not associated with a specific owner but are associated the client as a whole.
8.1.3. Stateid Definition

When the server grants a lock of any type (including opens, record locks, delegations, and layouts) 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 record locks on a file owned by a specific lock-owner and gotten via an open for a specific open-owner, has its own identifying stateid. Delegations and layouts 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 client 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.

The server may assign stateids independently for different clients and a stateid with the same bit pattern for one client may designate an entirely different set of locks for a different client. The stateid is always interpreted with respect to the client ID associated with the current session. Stateids apply to all sessions associated with the given client ID and the client may use a stateid obtained from one session on another session associated with the same client ID.

8.1.3.1. 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, to be discussed below, the purpose of the sequence value within NFSv4.1 is to allow the server to communicate to the client the order in which operations that modified locking state associated with a stateid have been processed.

In the case of stateids associated with opens, i.e. the stateids returned by OPEN (the state for the open, rather than that for the delegation), OPEN_DOWNGRADE, or CLOSE, the server MUST provide an "seqid" value starting at one for the first use of a given "other" value and incremented by one with each subsequent operation returning a stateid.

In the case of other sorts of stateids (i.e. stateids associated with record locks and delegations), the server MAY provide an incrementing sequence value on successive stateids returned with same identifying field, or it may return the value zero. If it does return a non-zero
"seqid" value it MUST start at one and be incremented by one with each subsequent operation returning a stateid with same "other" value, just as is done with open state.

The client when using a stateid as a parameter to an operation, must, except in the case of a special stateid, set the sequence value to zero. If the value is non-zero, the server MUST return the error NFS4ERR_BAD_STATEID.

8.1.3.2. Special Stateids

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

The following combinations of "other" and "seqid" are defined in NFSv4.1:

- 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.

- 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.

- When "other" is zero and "seqid" is one, the stateid represents the current stateid, which is whatever value is the last stateid returned by an operation within the COMPOUND. In the case of an OPEN, the stateid returned for the open file, and not the delegation is used. The stateid passed to the operation in place of the special value has its "seqid" value set to zero. If there is no operation in the COMPOUND which has returned a stateid value, the server MUST return the error NFS4ERR_BAD_STATEID.

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 ID’s or filehandles and can be used with all valid
client ID’s and filehandles. In the case of a special stateid designating the current current stateid, the current stateid value substituted for the special stateid is associated with a particular client ID and filehandle.

8.1.3.3. Stateid Lifetime and Validation

Stateids must remain valid until either a client reboot or a sever reboot 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 the stateid remains a valid designation of that revoked state until the client frees it by using FREE_STATEID. Stateids associated with record locks are an exception. They remain valid even if a LOCKU free all remaining locks, so long as the open file with which they are associated remains open, unless the client does a FREE_STATEID to cause the stateid to be freed.

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, in all case in which it is required to return incrementing "seqid" values in general.

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.

- An index into a table of locking-state structures.
- A generation number which is incremented on each allocation of a table entry for a particular use.

And then store in each table entry,

- The current generation number.
- The client ID with which the stateid is associated.
- The filehandle of the file on which the locks are taken.
- An indication of the type of stateid (open, record lock, file delegation, directory delegation, layout).
- The last "seqid" value returned corresponding to the current "other" value.
With this information, the following procedure would be used to validate an incoming stateid and return an appropriate error, when necessary:

- If the server has restarted resulting in loss of all lessed state but the sessionid and clientID are still valid, return NFS4ERR_STALE_STATEID. (If server restart has resulted in an invalid client ID or sessionid is invalid, SEQUENCE will return an error - not NFS4ERR_STATE_STATEID - and the operation that takes a stateid as an argument will never be processed.)

- If the "other" field is all zeros or all ones, check that the "other" and "seqid" match a defined combination for a special stateid and that that stateid can be used in the current context. If not, then return NFS4ERR_BAD_STATEID.

- If the "seqid" field is not zero, return NFS4ERR_BAD_STATEID.

- Otherwise divide the "other" into a table index and an entry generation.

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

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

  - If the selected table entry does not match the current file handle, return NFS4ERR_BAD_STATEID.

  - If the client ID in the table entry does not match the client ID associated with the current session, return NFS4ERR_BAD_STATEID.

  - If the stateid type is not valid for the context in which the stateid appears, return NFS4ERR_BAD_STATEID.

  - Otherwise, the stateid is valid and the table entry should contain any additional information about 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 octet ranges.

8.1.4. 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.

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 record locks and share reservations, are held by the state-owner. If no state is established by the client, either record lock or share reservation, a special stateid for anonymous state (zero as "other" and "seqid") is used. Regardless whether a stateid for anonymous state or a stateid returned by the server is used, if there is a conflicting share reservation or mandatory record 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. Record 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, record locks are required on the file before I/O is possible). When record locks are advisory, they only prevent the granting of conflicting lock requests and have no effect on READs or WRITES. Mandatory record 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).

Note that for UNIX environments that support mandatory file locking, the distinction between advisory and mandatory locking is subtle. In fact, advisory and mandatory record 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) record 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 record locks, so the
server always checks for record locks during I/O requests.

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

Every stateid with the exception of special stateid values, 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 caused the allocation of the open stateid and as modified by subsequent OPENS and OPEN_DOWNGRADEs for the same open-owner/file pair. Stateids returned by record lock operations imply the access mode for the open stateid associated with the lock set represented by the stateid. Delegation stateids have an access mode based on the type of delegation. 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 specify 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.

The read bypass special stateid (all bits of "other" and "seqid" set to one) stateid indicates a desire to bypass locking checks. The server MAY allow READ operations to bypass locking checks at the server, when this special stateid is used. However, WRITE operations with this special stateid value MUST NOT bypass locking checks and are treated exactly the same as if a special stateid for anonymous state 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.

8.2. Lock Ranges

The protocol allows a lock owner to request a lock with an octet range and then either upgrade, downgrade, 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 filesystems may not be able to support sub-range lock semantics. In the event that a server receives a locking request that represents a sub-range of current locking state for the lock owner, the server is allowed to return the error NFS4ERR_LOCK_RANGE to signify that it does not support sub-range lock operations. Therefore, the client should be prepared to receive this error and, if appropriate, report the error to the requesting application.

The client is discouraged from combining multiple independent locking ranges that happen to be adjacent into a single request since the server may not support sub-range requests and for reasons related to the recovery of file locking state in the event of server failure. As discussed in the section "Server Failure and Recovery" below, the server may employ certain optimizations during recovery that work effectively only when the client’s behavior during lock recovery is similar to the client’s locking behavior prior to server failure.

8.3. 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.
8.4. Blocking Locks

Some clients require the support of blocking locks. NFSv4.1 does not provide a callback when a previously unavailable lock becomes available. Clients thus have no choice but to continually poll for the lock. This presents a fairness problem. Two new lock types are added, READW and WRITEW, and are used to indicate to the server that the client is requesting a blocking lock. The server should maintain an ordered list of pending blocking locks. When the conflicting lock is released, the server may wait the lease period for the first waiting client to re-request the lock. After the lease period expires the next waiting client request is allowed the lock. Clients are required to poll at an interval sufficiently small that it is likely to acquire the lock in a timely manner. The server is not required to maintain a list of pending blocked locks as it is used to increase fairness and not correct operation. Because of the unordered nature of crash recovery, storing of lock state to stable storage would be required to guarantee ordered granting of blocking locks.

Servers may also note the lock types and delay returning denial of the request to allow extra time for a conflicting lock to be released, allowing a successful return. In this way, clients can avoid the burden of needlessly frequent polling for blocking locks. The server should take care in the length of delay in the event the client retransmits the request.

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.

8.5. 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.

Since each session is associated with a specific client, any
operation issued on that session is an indication that the associated client is reachable. When a request is issued for a given session, execution of a SEQUENCE operation will result in all leases for the associated client to be implicitly renewed. 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, via a COMPOUND that consists solely of a single SEQUENCE operation. The number of locks held by the client is not a factor since all state for the client is involved with the lease renewal action.

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

8.6. 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.

8.6.1. Client Failure and Recovery

In the event that a client fails, the server may release the client’s locks when the associated leases have expired. Conflicting locks from another client may only be granted after this lease expiration. When a client has not failed and re-establishes his lease before expiration occurs, requests for conflicting locks will not be granted.

To minimize client delay upon restart, lock requests are associated with an instance of the client by a client supplied verifier. This verifier is part of the initial EXCHANGE_ID call made by the client. The server returns a client ID as a result of the EXCHANGE_ID operation. The client then confirms the use of the client ID by establishing a session associated with that client ID. All locks, including opens, record locks, delegations, and layout obtained by sessions using that client ID are associated with that client ID.

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. At this point conflicting locks from other clients, kept waiting while the leaser had not yet expired, can be granted.

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

8.6.2. 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 a 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.

A client can determine that loss of locking state has occurred via several methods.

1. When a SEQUENCE succeeds, but sr_status_flags in the reply to SEQUENCE indicates SEQ4_STATUS_RESTART_RECLAIM_NEEDED (see Section 17.46.4). The client’s client ID and session are valid (have persisted through server restart) and the client can now re-establish its lock state (Section 8.6.2.1).

2. When an operation returns NFS4ERR_STALE_STATEID, this indicates a stateid invalidated by a server reboot or restart. Since the operation that returned NFS4ERR_STALE_STATEID MUST have been preceded by SEQUENCE, and SEQUENCE did not return an error, this means the client ID and session are valid. The client can now re-establish lock state as described in Section 8.6.2.1. Note that the server should (MUST) have set SEQ4_STATUS_RESTART_RECLAIM_NEEDED in the sr_status_flags of the results of the SEQUENCE operation, and thus this situation should be the same as that described above.

3. When a SEQUENCE operation returns NFS4ERR_STALE_CLIENTID, this means both sessionid SEQUENCE refers to (field sa_sessionid) and the implied client ID are now invalid, where the client ID was invalidated by server reboot or restart or by lease expiration. When SEQUENCE returns NFS4ERR_STALE_CLIENTID, the client must establish a new client ID (see Section 8.1.1) and re-establish its lock state (Section 8.6.2.1).
4. When a SEQUENCE operation returns NFS4ERR_BADSESSION, this may mean the session has been destroyed, but the client ID is still valid. The client issues a CREATE_SESSION request with the client ID to re-establish the session. If CREATE_SESSION fails with NFS4ERR_STALE_CLIENTID, the client must establish a new client ID (see Section 8.1.1) and re-establish its lock state (Section 8.6.2.1). If CREATE_SESSION succeeds, the client must then re-establish its lock state (Section 8.6.2.1).

5. When a operation, neither SEQUENCE nor preceded by SEQUENCE (for example, CREATE_SESSION, DESTROY_SESSION) returns NFS4ERR_STALE_CLIENTID. The client MUST establish a new client ID (Section 8.1.1) and re-establish its lock state (Section 8.6.2.1).

8.6.2.1. State Reclaim

Once a session is established using the new client ID, the client will use reclaim-type locking requests (i.e. LOCK requests with reclaim set to true and OPEN operations with a claim type of CLAIM_PREVIOUS) to re-establish its locking state. Once this is done, or if there is no such locking state to reclaim, the client does a RECLAIM_COMPLETE operation to indicate that it has reclaimed all of the locking state that it will reclaim. Once a client does a RECLAIM_COMPLETE operation, it may attempt non-reclaim locking operations, although it may get NFS4ERR_GRACE errors on these until the period of special handling is over.

The period of special handling of locking and READs and WRITEs, is referred to as the "grace period". During the grace period, clients recover locks and the associated state using reclaim-type locking requests. During this 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, unless it is able to guarantee that these may be done safely, as described below.

The grace period may last until all clients who are known to possibly have had locks have done a RECLAIM_COMPLETE operation, indicating that they have finished reclaiming the locks they held before the server reboot. The server is assumed to maintain in stable storage a list of clients who may have such locks. The server may also terminate the grace period before all clients have done RECLAIM_COMPLETE. The server SHOULD NOT terminate the grace period before a time equal to the lease period in order to give clients an opportunity to find out about the server reboot. Some additional time in order to allow time to establish a new client ID and session and to effect lock reclams may be added.
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 even within the grace period, although NFS4ERR_GRACE must always be returned to clients attempting a non-reclaim lock request before doing their own RECLAIM_COMPLETE. 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 a potential 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 the server maintained on stable storage summary information on whether mandatory locks exist, either mandatory record locks, or share reservations specifying deny modes, many requests could be allowed during the grace period. If it is known that no such share reservations exist, OPEN request that do not specify deny modes may be safely granted. If, in addition, it is known that no mandatory record locks exist, either through information stored on stable storage or simply because the server does not support such locks, READ and WRITE requests may be safely processed during the grace period.

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 can not do so.

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

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.

8.6.3. 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 free all locks held for the client, or it may allow the lock state to remain for a considerable period, subject to the constraint that if a request for a conflicting lock is made, locks associated with expired leases do not prevent such a conflicting lock from being granted but are revoked as necessary so as not to interfere with such conflicting requests.

If the server chooses to delay freeing of lock state until there is a conflict, it may either free all of the clients locks once there is a conflict, or it may only revoke the minimum set of locks necessary to allow conflicting requests. When it adopts the finer-grained approach, it must revoke all locks associated with a given stateid, as long as it revokes a single such lock.

When the server chooses to free all of a client’s lock state, either immediately upon lease expiration, or a result of the first attempt to get a lock, 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, the status returned by the SEQUENCE operation will indicate a loss of locking state. In addition all I/O submitted by the client with the now invalid stateids will fail with the server returning the error NFS4ERR_EXPIRED. Once the client learns of the loss of locking state, it will suitably notify the applications that held the invalidated locks. The client should then take action to free invalidated stateids, either by establishing a new client ID using a new verifier or by doing a FREE_STATEID operation to release each of the invalidated stateids.

When the server adopts a finer-grained approach to revocation of locks when lease have expired, only a subset of stateids will normally become invalid during a network partition. When the client is able to communicate with the server after such a network
partition, the status returned by the SEQUENCE operation will indicate a partial loss of locking state. In addition, operations, including I/O submitted by the client with the now invalid stateids will fail with the server returning the error NFS4ERR_EXPIRED. Once the client learns of the loss of locking state, it will use the TEST_STATEID operation on all of its stateids to determine which locks have been lost and then suitably notify the applications that held the invalidated locks. The client can then release the invalidated locking state and acknowledge the revocation of the associated locks by doing a FREE_STATEID operation on each of the invalidated stateids.

When a network partition is combined with a server reboot, there are edge conditions that place requirements on the server in order to avoid silent data corruption following the server reboot. Two of these edge conditions are known, and are discussed below.

The first edge condition arises as a result of the scenarios such as the following:

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, and the server releases lock.
4. Client B acquires a lock that would have conflicted with that of Client A.
5. Client B releases its lock.
8. Client A connects to new server instance and finds out about server reboot.
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.

The second known edge condition arises in situations such as the following:
1. Client A acquires one or more locks.

2. Server reboots.

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

4. Server’s reclaim grace period ends. Client A has either no locks or an incomplete set of locks known to the server.

5. Client B acquires a lock that would have conflicted with a lock of client A that was not reclaimed.

6. Client B releases the lock.

7. Server reboots a second time.


9. Client A connects to new server instance and finds out about server reboot.

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.

Solving the first and second edge conditions requires that the server either always assumes after it reboots that some 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 intends to be whenever edge conditions arise. 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 released. For the two edge conditions discussed above, the harshest a server can be, and still support a grace period for reclaims, requires that the server record in stable storage information some minimal information. For example, a server implementation could, for each client, save in stable storage a record containing:

- the client’s id string
a boolean that indicates if the client’s lease expired or if there was administrative intervention (see Section 8.7) to revoke a record lock, share reservation, or delegation and there has been no acknowledgement (via FREE_STATEID) of such revocation.

a boolean that indicates whether the client may have locks that it believes to be reclaimable in situations which the grace period was terminated, making the server’s view of lock reclaimability suspect. The server will set this for any client record in stable storage where the client has not done a RECLAIM_COMPLETE, before it grants any new (i.e. not reclaimed) lock to any client.

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.

For the second edge condition, after the server reboots for a second time, the indication that the client had not completed its reclaims at the time at which the grace period ended means that the server must reject a reclaim from client A with the error NFS4ERR_NO_GRACE.

When either edge condition occurs, the client’s attempt to reclaim locks will result in the error NFS4ERR_NO_GRACE. When this is received, or after the client reboots with no lock state, the client will issue a RECLAIM_COMPLETE. When the RECLAIM_COMPLETE is received, the server and client are again in agreement regarding reclaimable locks and both booleans in persistent storage can be reset, to be set again only when there is a subsequent event that causes lock reclaim operations to be questionable.

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, record locks, and delegations):

1. Reject all reclaims with NFS4ERR_NO_GRACE. This is extremely unforgiving, but necessary if the server does not record lock state in stable storage.

2. Record sufficient state in stable storage such that all known edge conditions involving server reboot, including the two noted in this section, are detected. False positives are acceptable. Note that at this time, it is not known if there are other edge conditions.

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

A mandate for the client’s handling of the NFS4ERR_NO_GRACE error 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 receives NFS4ERR_NO_GRACE, it could examine the change attribute of the objects the client is trying to reclaim state for, and use that to determine whether to re-establish the state via normal OPEN or LOCK requests. This is acceptable provided the client’s operating environment allows it. In other words, the client implementor is advised to document for his users the behavior. The client could also inform the application that its record 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 the section, "Data Caching and Revocation" 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 8.7.

8.7. Server Revocation of Locks

At any point, the server can revoke locks held by a client and the client must be prepared for this event. When the client detects that its locks have been or may have been revoked, the client is responsible for validating the state information between itself and the server. Validating locking state for the client means that it must verify or reclaim state for each lock currently held.

The first occasion of lock revocation is upon server reboot or restart. In this instance the client will receive an error (NFS4ERR_STALE_STATEID on an operation that takes a stateid as an argument or NFS4ERR_STALE_CLIENTID on an operation that takes a sessionid or client ID) and the client will proceed with normal crash recovery as described in the Section 8.6.2.1.

The second occasion of lock revocation is the inability to renew the lease before expiration, as discussed above. While this is considered a rare or unusual event, the client must be prepared to recover. The server is responsible for determining lease expiration, and deciding exactly how to deal with it, informing the client of the scope of the lock revocation. The client then uses the status information provided by the server in the SEQUENCE results (field sr_status_flags, see Section 17.46.4) to synchronize its locking state with that of the server, in order to recover.
The third occasion of lock revocation can occur as a result of revocation of locks within the lease period, either because of administrative intervention, or because a recallable lock (a delegation or layout) was not returned within the lease period after having been recalled. While these are considered rare events, they are possible and the client must be prepared to deal with them. When either of these events occur, the client finds out about the situation through the status returned by the SEQUENCE operation. Any use of stateids associated with revoked locks will receive the error NFS4ERR_ADMIN_REVOKED or NFS4ERR_DELEG_REVOKED, as appropriate.

In all situations in which a subset of locking state may have been revoked, which include all cases in which locking state is revoked within the lease period, it is up to the client to determine which locks have been revoked and which have not. It does this by using the TEST_STATEID operation on the appropriate set of stateids. Once the set of revoked locks has been determined, the applications can be notified, and the invalidated stateids can be freed and lock revocation acknowledged by using FREE_STATEID.

8.8. Share Reservations

A share reservation is a mechanism to control access to a file. It is a separate and independent mechanism from record locking. When a client opens a file, it issues an OPEN operation to the server specifying the type of access required (READ, WRITE, or BOTH) and the type of access to deny others (deny NONE, READ, WRITE, or BOTH). If the OPEN fails the client will fail the application’s open request.

Pseudo-code definition of the semantics:

```plaintext
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:
8.9. OPEN/CLOSE Operations

To provide correct share semantics, a client MUST use the OPEN operation to obtain the initial filehandle and indicate the desired access and what access, if any, to deny. Even if the client intends to use a special stateid for anonymous state or read bypass, it must still obtain the filehandle for the regular file with the OPEN operation so the appropriate share semantics can be applied. For clients that do not have a deny mode built into their open programming interfaces, deny equal to NONE should be used.

The OPEN operation with the CREATE flag, also subsumes the CREATE operation for regular files as used in previous versions of the NFS protocol. This allows a create with a share to be done atomically.

The CLOSE operation removes all share reservations held by the open-owner on that file. If record locks are held, the client SHOULD release all locks before issuing a CLOSE. The server MAY free all outstanding locks on CLOSE but some servers may not support the CLOSE of a file that still has record locks held. The server MUST return failure, 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, a file opened with deny READ/WRITE using a filehandle obtained through LOOKUP could only be read using the special read bypass stateid and could not be written at all because it would not have a valid stateid and the special anonymous stateid would not be allowed access.

8.10. 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.

8.11. Short and Long Leases

When determining the time period for the server lease, the usual
lease tradeoffs apply. Short leases are good for fast server
recovery at a cost of increased operations to effect lease renewal
(when there are no other operations during the period to effect lease
renewal as a side-effect). Long leases are certainly kinder and
gentler to servers trying to handle very large numbers of clients.
The number of extra requests to effect lock renewal drop in inverse
proportion to the lease time. The disadvantages of long leases
include the possibility of slower recovery after certain failures.
After server failure, a longer grace period may be required when some
clients do not promptly reclaim their locks and do a
RECLAIM_COMPLETE. In the event of client failure, it can longer
period for leases to expire thus forcing conflicting requests to
wait.

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.
8.12. 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.

8.13. Vestigial Locking Infrastructure From V4.0

There are a number of operations and fields within existing operations that no longer have a function in minor version one. In one way or another, these changes are all due to the implementation of sessions which provides client context and replay protection as a base feature of the protocol, separate from locking itself.

The following operations have become mandatory-to-not-implement. The server should return NFS4ERR_NOTSUPP if these operations are found in an NFSv4.1 COMPOUND.

- SETCLIENTID since its function has been replaced by EXCHANGE_ID.
- SETCLIENTID_CONFIRM since client ID confirmation now happens by means of CREATE_SESSION.
- OPEN_CONFIRM because OPEN's no longer require confirmation to establish an owner-based sequence value.
- RELEASE_LOCKOWNER because lock-owners with no associated locks have any sequence-related state and so can be deleted by the
server at will.

- RENEW because every SEQUENCE operation for a session causes lease renewal, making a separate operation useless.

Also, there are a number of fields, present in existing operations related to locking that have no use in minor version one. They were used in minor version zero to perform functions now provided in a different fashion.

- Sequence id’s used to sequence requests for a given state-owner and to provide replay protection, now provided via sessions.

- Client IDs used to identify the client associated with a given request. Client identification is now available using the client ID associated with the current session, without needing an explicit client ID field.

Such vestigial fields in existing operations should be set by the client to zero. When they are not, the server MUST return an NFS4ERR_INVAL error.

9. Client-Side Caching

Client-side caching of data, of file attributes, and of file names is essential to providing good performance with the NFS protocol. Providing distributed cache coherence is a difficult problem and previous versions of the NFS protocol have not attempted it. Instead, several NFS client implementation techniques have been used to reduce the problems that a lack of coherence poses for users. These techniques have not been clearly defined by earlier protocol specifications and it is often unclear what is valid or invalid client behavior.

The NFS version 4 protocol uses many techniques similar to those that have been used in previous protocol versions. The NFS version 4 protocol does not provide distributed cache coherence. However, it defines a more limited set of caching guarantees to allow locks and share reservations to be used without destructive interference from client side caching.

In addition, the NFS version 4 protocol introduces a delegation mechanism which allows many decisions normally made by the server to be made locally by clients. This mechanism provides efficient support of the common cases where sharing is infrequent or where sharing is read-only.
9.1. Performance Challenges for Client-Side Caching

Caching techniques used in previous versions of the NFS protocol have been successful in providing good performance. However, several scalability challenges can arise when those techniques are used with very large numbers of clients. This is particularly true when clients are geographically distributed which classically increases the latency for cache revalidation requests.

The previous versions of the NFS protocol repeat their file data cache validation requests at the time the file is opened. This behavior can have serious performance drawbacks. A common case is one in which a file is only accessed by a single client. Therefore, sharing is infrequent.

In this case, repeated reference to the server to find that no conflicts exist is expensive. A better option with regards to performance is to allow a client that repeatedly opens a file to do so without reference to the server. This is done until potentially conflicting operations from another client actually occur.

A similar situation arises in connection with file locking. Sending file lock and unlock requests to the server as well as the read and write requests necessary to make data caching consistent with the locking semantics (see the section "Data Caching and File Locking") can severely limit performance. When locking is used to provide protection against infrequent conflicts, a large penalty is incurred. This penalty may discourage the use of file locking by applications.

The NFS version 4 protocol provides more aggressive caching strategies with the following design goals:

- Compatibility with a large range of server semantics.
- Provide the same caching benefits as previous versions of the NFS protocol when unable to provide the more aggressive model.
- Requirements for aggressive caching are organized so that a large portion of the benefit can be obtained even when not all of the requirements can be met.

The appropriate requirements for the server are discussed in later sections in which specific forms of caching are covered. (see the section "Open Delegation").

9.2. Delegation and Callbacks

Recallable delegation of server responsibilities for a file to a client improves performance by avoiding repeated requests to the server in the absence of inter-client conflict. With the use of a "callback" RPC from server to client, a server recalls delegated responsibilities when another client engages in sharing of a
delegated file.

A delegation is passed from the server to the client, specifying the object of the delegation and the type of delegation. There are different types of delegations but each type contains a stateid to be used to represent the delegation when performing operations that depend on the delegation. This stateid is similar to those associated with locks and share reservations but differs in that the stateid for a delegation is associated with a 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.

The callback path or backchannel is established by CREATE_SESSION and BINDCONN_TO_SESSION, and the client is required to maintain it. Because the backchannel may be down, even temporarily, correct protocol operation does not depend on them. Preliminary testing of callback functionality by means of a CB_COMPOUND procedure with a single operation, CB_SEQUENCE, can be used to check the continuity of the backchannel. A server avoids delegating responsibilities until it has determined that the backchannel exists. 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, WANT_DELEGATIONS, and GET_DIR_DELEGATIONS to be processed without any delegations being granted.

Once granted, a delegation behaves in many ways like a lock. There is an associated lease that is subject to renewal together with all of the other leases held by that client.

Unlike locks, an operation by a second client to a delegated file will cause the server to recall a delegation through a callback.

On recall, the client holding the delegation must flush modified state (such as modified data) to the server and return the delegation. The conflicting request will not receive a response until the recall is complete. The recall is considered complete when the client returns the delegation or the server times out on the recall and revokes the delegation as a result of the timeout. Following the resolution of the recall, the server has the information necessary to grant or deny the second client's request.

At the time the client receives a delegation recall, it may have substantial state that needs to be flushed to the server. Therefore, the server should allow sufficient time for the delegation to be returned since it may involve numerous RPCs to the server. If the server is able to determine that the client is diligently flushing
state to the server as a result of the recall, the server may extend
the usual time allowed for a recall. However, the time allowed for
recall completion should not be unbounded.

An example of this is when responsibility to mediate opens on a given
file is delegated to a client (see the section "Open Delegation").
The server will not know what opens are in effect on the client.
Without this knowledge the server will be unable to determine if the
access and deny state for the file allows any particular open until
the delegation for the file has been returned.

A client failure or a network partition can result in failure to
respond to a recall callback. In this case, the server will revoke
the delegation which in turn will render useless any modified state
still on the client.

9.2.1. Delegation Recovery

There are three situations that delegation recovery must deal with:

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

In the event the client reboots or restarts, the failure to renew
leases will result in the revocation of record locks and share
reservations. Delegations, however, may be treated a bit
differently.

There will be situations in which delegations will need to be
reestablished after a client reboots or restarts. The reason for
this is the client may have file data stored locally and this data
was associated with the previously held delegations. The client will
need to reestablish the appropriate file state on the server.

To allow for this type of client recovery, the server MAY extend the
period for delegation recovery beyond the typical lease expiration
period. This implies that requests from other clients that conflict
with these delegations will need to wait. Because the normal recall
process may require significant time for the client to flush changed
state to the server, other clients need be prepared for delays that
occur because of a conflicting delegation. This longer interval
would increase the window for clients to reboot and consult stable
storage so that the delegations can be reclaimed. For open
delegations, such delegations are reclaimed using OPEN with a claim
type of CLAIM_DELEGATE_PREV. (See the sections on "Data Caching and
Revocation" and "Operation 18: OPEN" for discussion of open
delegation and the details of OPEN respectively).

A server MAY support a claim type of CLAIM_DELEGATE_PREV, but if it
does, it MUST NOT remove delegations upon a CREATE_SESSION that
confirms a client ID created by EXCHANGE_ID, and instead MUST, for a
period of time no less than that of the value of the lease_time
attribute, maintain the client’s delegations to allow time for the
client to issue CLAIM_DELEGATE_PREV requests. The server that
supports CLAIM_DELEGATE_PREV MUST support the DELEGPURGE operation.

When the server reboots or restarts, delegations are reclaimed (using
the OPEN operation with CLAIM_PREVIOUS) in a similar fashion to
record locks and share reservations. However, there is a slight
semantic difference. In the normal case if the server decides that a
delegation should not be granted, it performs the requested action
(e.g. OPEN) without granting any delegation. For reclaim, the
server grants the delegation but a special designation is applied so
that the client treats the delegation as having been granted but
recalled by the server. Because of this, the client has the duty to
write all modified state to the server and then return the
delegation. This process of handling delegation reclaim reconciles
three principles of the NFS version 4 protocol:

- Upon reclaim, a client reporting resources assigned to it by an
  earlier server instance must be granted those resources.
- The server has unquestionable authority to determine whether
delegations are to be granted and, once granted, whether they are
to be continued.
- The use of callbacks is not to be depended upon until the client
  has proven its ability to receive them.

When a network partition occurs, delegations are subject to freeing
by the server when the lease renewal period expires. This is similar
to the behavior for locks and share reservations. For delegations,
however, the server may extend the period in which conflicting
requests are held off. Eventually the occurrence of a conflicting
request from another client will cause revocation of the delegation.
A loss of the callback path (e.g. by later network configuration
change) will have the same effect. A recall request will fail and
revocation of the delegation will result.

A client normally finds out about revocation of a delegation when it
uses a stateid associated with a delegation and receives the error
NFS4ERR_EXPIRED. It also may find out about delegation revocation
after a client reboot when it attempts to reclaim a delegation and
receives that same error. Note that in the case of a revoked write open delegation, there are issues because data may have been modified by the client whose delegation is revoked and separately by other clients. See the section "Revocation Recovery for Write Open Delegation" for a discussion of such issues. Note also that when delegations are revoked, information about the revoked delegation will be written by the server to stable storage (as described in the section "Crash Recovery"). This is done to deal with the case in which a server reboots after revoking a delegation but before the client holding the revoked delegation is notified about the revocation.

9.3. Data Caching

When applications share access to a set of files, they need to be implemented so as to take account of the possibility of conflicting access by another application. This is true whether the applications in question execute on different clients or reside on the same client.

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

9.3.1. Data Caching and OPENs

In order to avoid invalidating the sharing assumptions that applications rely on, NFS version 4 clients should not provide cached data to applications or modify it on behalf of an application when it would not be valid to obtain or modify that same data via a READ or WRITE operation.

Furthermore, in the absence of open delegation (see the section "Open Delegation") two additional rules apply. Note that these rules are obeyed in practice by many NFS version 2 and version 3 clients.

- First, cached data present on a client must be revalidated after doing an OPEN. Revalidating means that the client fetches the change attribute from the server, compares it with the cached change attribute, and if different, declares the cached data (as well as the cached attributes) as invalid. This is to ensure that the data for the OPENed file is still correctly reflected in the client’s cache. This validation must be done at least when the client’s OPEN operation includes DENY=WRITE or BOTH thus terminating a period in which other clients may have had the
opportunity to open the file with WRITE access. Clients may choose to do the revalidation more often (i.e. at OPENs specifying DENY=NONE) to parallel the NFS version 3 protocol’s practice for the benefit of users assuming this degree of cache revalidation.

Since the change attribute is updated for data and metadata modifications, some client implementors may be tempted to use the time_modify attribute and not change to validate cached data, so that metadata changes do not spuriously invalidate clean data. The implementor is cautioned in this approach. The change attribute is guaranteed to change for each update to the file, whereas time_modify is guaranteed to change only at the granularity of the time_delta attribute. Use by the client’s data cache validation logic of time_modify and not change runs the risk of the client incorrectly marking stale data as valid.

Second, modified data must be flushed to the server before closing a file OPENed for write. This is complementary to the first rule. If the data is not flushed at CLOSE, the revalidation done after client OPENs as file is unable to achieve its purpose. The other aspect to flushing the data before close is that the data must be committed to stable storage, at the server, before the CLOSE operation is requested by the client. In the case of a server reboot or restart and a CLOSED file, it may not be possible to retransmit the data to be written to the file. Hence, this requirement.

9.3.2. Data Caching and File Locking

For those applications that choose to use file locking instead of share reservations to exclude inconsistent file access, there is an analogous set of constraints that apply to client side data caching. These rules are effective only if the file locking is used in a way that matches in an equivalent way the actual READ and WRITE operations executed. This is as opposed to file locking that is based on pure convention. For example, it is possible to manipulate a two-megabyte file by dividing the file into two one-megabyte regions and protecting access to the two regions by file locks on octets zero and one. A lock for write on octet zero of the file would represent the right to do READ and WRITE operations on the first region. A lock for write on octet one of the file would represent the right to do READ and WRITE operations on the second region. As long as all applications manipulating the file obey this convention, they will work on a local file system. However, they may not work with the NFS version 4 protocol unless clients refrain from data caching.

The rules for data caching in the file locking environment are:
First, when a client obtains a file lock for a particular region, the data cache corresponding to that region (if any cache data exists) must be revalidated. If the change attribute indicates that the file may have been updated since the cached data was obtained, the client must flush or invalidate the cached data for the newly locked region. A client might choose to invalidate all of non-modified cached data that it has for the file but the only requirement for correct operation is to invalidate all of the data in the newly locked region.

Second, before releasing a write lock for a region, all modified data for that region must be flushed to the server. The modified data must also be written to stable storage.

Note that flushing data to the server and the invalidation of cached data must reflect the actual octet 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 record locking in non-standard ways (e.g. using a record lock as a global semaphore) by flushing to the server more data upon an LOCKU than is covered by the locked range. This may include modified data within files other than the one for which the unlocks are being done. In such cases, the client must not interfere with applications whose READs and WRITEs are being done only within the bounds of record locks which the application holds. For example, an application locks a single octet of a file and proceeds to write that single octet. A client that chose to handle a LOCKU by flushing all modified data to the server could validly write that single octet in response to an unrelated unlock. However, it would not be valid to write the entire...
block in which that single written octet was located since it includes an area that is not locked and might be locked by another client. Client implementations can avoid this problem by dividing files with modified data into those for which all modifications are done to areas covered by an appropriate record lock and those for which there are modifications not covered by a record lock. Any writes done for the former class of files must not include areas not locked and thus not modified on the client.

9.3.3. Data Caching and Mandatory File Locking

Client side data caching needs to respect mandatory file locking when it is in effect. The presence of mandatory file locking for a given file is indicated 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.

9.3.4. 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 NFS version 3 clients, the typical practice has been to assume for the purpose of caching that distinct filehandles represent distinct file system objects. The client then has the choice to organize and maintain the data cache on this basis.

In the NFS version 4 protocol, there is now the possibility to have significant deviations from a "one filehandle per object" model because a filehandle may be constructed on the basis of the object’s pathname. Therefore, clients need a reliable method to determine if two filehandles designate the same file system object. If clients were simply to assume that all distinct filehandles denote distinct objects and proceed to do data caching on this basis, caching inconsistencies would arise between the distinct client side objects which mapped to the same server side object.

By providing a method to differentiate filehandles, the NFS version 4 protocol alleviates a potential functional regression in comparison with the NFS version 3 protocol. Without this method, caching
inconsistencies within the same client could occur and this has not been present in previous versions of the NFS protocol. Note that it is possible to have such inconsistencies with applications executing on multiple clients but that is not the issue being addressed here.

For the purposes of data caching, the following steps allow an NFS version 4 client to determine whether two distinct filehandles denote the same server side object:

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

- If GETATTR for any file with an fsid that matches the fsid of the two filehandles in question returns a unique_handles attribute with a value of TRUE, then the two objects are distinct.

- If GETATTR directed to the two filehandles does not return the fileid attribute for 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.

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

- Otherwise they are the same object.

9.4. Open Delegation

When a file is being OPENed, the server may delegate further handling of opens and closes for that file to the opening client. Any such delegation is recallable, since the circumstances that allowed for the delegation are subject to change. In particular, the server may receive a conflicting OPEN from another client, the server must recall the delegation before deciding whether the OPEN from the other client may be granted. Making a delegation is up to the server and clients should not assume that any particular OPEN either will or will not result in an open delegation. The following is a typical set of conditions that servers might use in deciding whether OPEN should be delegated:

- The client must be able to respond to the server’s callback requests. The server will use the CB_NULL procedure for a test of callback ability.
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, read and write. A read open delegation allows a client to handle, on its own, requests to open a file for reading that do not deny read access to others. Multiple read open delegations may be outstanding simultaneously and do not conflict. A write open delegation allows the client to handle, on its own, all opens. Only one write open delegation may exist for a given file at a given time and it is inconsistent with any read open delegations.

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

When a client has an open delegation, it does not send OPENs or CLOSEs to the server but updates the appropriate status internally. For a read open delegation, opens that cannot be handled locally (opens for write or that deny read access) must be sent to the server.

When an open delegation is made, the response to the OPEN contains an open delegation structure which specifies the following:

o the type of delegation (read or write)

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

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.
9.4.1. Open Delegation and Data Caching

OPEN delegation allows much of the message overhead associated with the opening and closing files to be eliminated. An open when an open delegation is in effect does not require that a validation message be sent to the server. The continued endurance of the "read open delegation" provides a guarantee that no OPEN for write and thus no write has occurred. Similarly, when closing a file opened for write and if write open delegation is in effect, the data written does not have to be flushed to the server until the open delegation is recalled. The continued endurance of the open delegation provides a guarantee that no open and thus no read or write has been done by another client.

For the purposes of open delegation, READs and WRITEs done without an OPEN are treated as the functional equivalents of a corresponding type of OPEN. This refers to the READs and WRITEs that use the special stateids consisting of all zero bits or all one bits. Therefore, READs or WRITEs with a special stateid done by another client will force the server to recall a write open delegation. A WRITE with a special stateid done by another client will force a recall of read open delegations.

With delegations, a client is able to avoid writing data to the server when the CLOSE of a file is serviced. The 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 an alternative method be in place for the same type of communication to occur between client and server.

In the delegation response, the server provides either the limit of the size of the file or the number of modified blocks and associated block size. The server must ensure that the client will be able to flush data to the server of a size equal to that provided in the original delegation. The server must make this assurance for all outstanding delegations. Therefore, the server must be careful in its management of available space for new or modified data taking into account available file system space and any applicable quotas. The server can recall delegations as a result of managing the available file system space. The client should abide by the server’s state space limits for delegations. If the client exceeds the stated limits for the delegation, the server’s behavior is undefined.
Based on server conditions, quotas or available file system space, the server may grant write open delegations with very restrictive space limitations. The limitations may be defined in a way that will always force modified data to be flushed to the server on close.

With respect to authentication, flushing modified data to the server after a CLOSE has occurred may be problematic. For example, the user of the application may have logged off the client and unexpired authentication credentials may not be present. In this case, the client may need to take special care to ensure that local unexpired credentials will in fact be available. This may be accomplished by tracking the expiration time of credentials and flushing data well in advance of their expiration or by making private copies of credentials to assure their availability when needed.

9.4.2. Open Delegation and File Locks

When a client holds a write open delegation, lock operations are performed locally. This includes those required for mandatory file locking. This can be done since the delegation implies that there can be no conflicting locks. Similarly, all of the revalidations that would normally be associated with obtaining locks and the flushing of data associated with the releasing of locks need not be done.

When a client holds a read open delegation, lock operations are not performed locally. All lock operations, including those requesting non-exclusive locks, are sent to the server for resolution.

9.4.3. Handling of CB_GETATTR

The server needs to employ special handling for a GETATTR where the target is a file that has a write open delegation in effect. The reason for this is that the client holding the 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 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 write delegation:

- The value of the change attribute will be obtained from the server and cached. Let this value be represented by c.
- 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.
- 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 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, change is defined as an unsigned integer rather than an opaque array of octets.

For the server, the following steps will be taken when providing a write delegation:
Upon providing a 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 \( sc \).

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 \( cc \).

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.

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 \( \geq sc + 1 \). 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 \( cc \neq sc \), 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;
}

return to client (that sent GETATTR) the attributes it requested, but make sure size comes from what CB_GETATTR returned. Do not update the file’s metadata with the client’s modified size.

In the case that the file attribute size is different than the server’s current value, the server treats this as a modification regardless of the value of the change attribute retrieved via CB_GETATTR and responds to the second client as in the last step.

This methodology resolves issues of clock differences between client and server and other scenarios where the use of CB_GETATTR break down.

It should be noted that the server is under no obligation to use CB_GETATTR and therefore the server MAY simply recall the delegation to avoid its use.

9.4.4. Recall of Open Delegation

The following events necessitate recall of an open delegation:

- Potentially conflicting OPEN request (or READ/WRITE done with "special" stateid)
- SETATTR issued by another client
- REMOVE request for the file
- RENAME request for the file as either source or target of the RENAME

Whether a RENAME of a directory in the path leading to the file
results in recall of an open delegation depends on the semantics of the server file system. If that file system denies such RENAMEs when a file is open, the recall must be performed to determine whether the file in question is, in fact, open.

In addition to the situations above, the server may choose to recall open delegations at any time if resource constraints make it advisable to do so. Clients should always be prepared for the possibility of recall.

When a client receives a recall for an open delegation, it needs to update state on the server before returning the delegation. These same updates must be done whenever a client chooses to return a delegation voluntarily. The following items of state need to be dealt with:

- If the file associated with the delegation is no longer open and no previous CLOSE operation has been sent to the server, a CLOSE operation must be sent to the server.

- If a file has other open references at the client, then OPEN operations must be sent to the server. The appropriate stateids will be provided by the server for subsequent use by the client since the delegation stateid will not longer be valid. These OPEN requests are done with the claim type of CLAIM_DELEGATE_CUR. This will allow the presentation of the delegation stateid so that the client can establish the appropriate rights to perform the OPEN. (see the section "Operation 18: OPEN" for details.)

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

- For a write open delegation, if at the time of recall the file is not open for write, all modified data for the file must be flushed to the server. If the delegation had not existed, the client would have done this data flush before the CLOSE operation.

- For a write open delegation when a file is still open at the time of recall, any modified data for the file needs to be flushed to the server.

- With the write open delegation in place, it is possible that the file was truncated during the duration of the delegation. For example, the truncation could have occurred as a result of an OPEN UNCHECKED with a size attribute value of zero. Therefore, if a truncation of the file has occurred and this operation has not been propagated to the server, the truncation must occur before
any modified data is written to the server.

In the case of write open delegation, file locking imposes some additional requirements. 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 write delegation was in effect. However, because the write open delegation implies no other locking by other clients, a simpler implementation is to flush all modified data for the file (as described just above) if any write lock has been released while the write open delegation was in effect.

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.

9.4.5. 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 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:

- 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 record 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.

When the client holds a delegation, it can not 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.

9.4.6. Delegation Revocation

At the point a delegation is revoked, if there are associated opens on the client, the applications holding these opens need to be notified. This notification usually occurs by returning errors for READ/WRITE operations or when a close is attempted for the open file.

If no opens exist for the file at the point the delegation is revoked, then notification of the revocation is unnecessary. However, if there is modified data present at the client for the file, the user of the application should be notified. Unfortunately, it may not be possible to notify the user since active applications may not be present at the client. See the section "Revocation Recovery for Write Open Delegation" for additional details.

9.5. Data Caching and Revocation

When locks and delegations are revoked, the assumptions upon which successful caching depend are no longer guaranteed. 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.

9.5.1. Revocation Recovery for Write Open Delegation

Revocation recovery for a write open delegation poses the special issue of modified data in the client cache while the file is not open. In this situation, any client which does not flush modified data to the server on each close must ensure that the user receives appropriate notification of the failure as a result of the revocation. Since such situations may require human action to correct problems, notification schemes in which the appropriate user or administrator is notified may be necessary. Logging and console messages are typical examples.

If there is modified data on the client, it must not be flushed normally to the server. A client may attempt to provide a copy of the file data as modified during the delegation under a different name in the file system name space to ease recovery. 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 file in question, such a saved copy of the client’s view of the file may be of particular value for recovery. In other case, 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.

9.6. Attribute Caching

The attributes discussed in this section do not include named attributes. Individual named attributes are analogous to files and caching of the data for these needs to be handled just as data caching is for ordinary files. Similarly, LOOKUP results from an OPENATTR directory are to be cached on the same basis as any other pathnames and similarly for directory contents.

Clients may cache file attributes obtained from the server and use them to avoid subsequent GETATTR requests. Such caching is write through in that modification to file attributes is always done by means of requests to the server and should not be done locally and cached. The exception to this are modifications to attributes that are intimately connected with data caching. Therefore, extending a file by writing data to the local data cache is reflected immediately in the size as seen on the client without this change being immediately reflected on the server. Normally such changes are not propagated directly to the server but when the modified data is flushed to the server, analogous attribute changes are made on the server. When open delegation is in effect, the modified attributes may be returned to the server in the response to a CB_RECALL call.

The result of local caching of attributes is that the attribute caches maintained on individual clients will not be coherent. Changes made in one order on the server may be seen in a different order on one client and in a third order on a different client.

The typical file system application programming interfaces do not provide means to atomically modify or interrogate attributes for multiple files at the same time. The following rules provide an environment where the potential incoherences mentioned above can be reasonably managed. These rules are derived from the practice of previous NFS protocols.

- All attributes for a given file (per-fsid attributes excepted) are cached as a unit at the client so that no non-serializability can arise within the context of a single file.

- An upper time boundary is maintained on how long a client cache entry can be kept without being refreshed from the server.

- When operations are performed that change attributes at the server, the updated attribute set is requested as part of the containing RPC. This includes directory operations that update
attributes indirectly. This is accomplished by following the
modifying operation with a GETATTR operation and then using the
results of the GETATTR to update the client’s cached attributes.

Note that if the full set of attributes to be cached is requested by
READDIR, the results can be cached by the client on the same basis as
attributes obtained via GETATTR.

A client may validate its cached version of attributes for a file by
fetching 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.
9.7. 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 whether the file is local file or is being access remotely). A client or server MAY fail to update attributes of a file that is being accessed via memory mapped I/O. This has several implications:

- 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.

- 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.

- If there is another client that is memory mapping the file, and if that client is holding a 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.

- The mixture of memory mapping and file locking on the same file is problematic. Consider the following scenario, where a page size on each client is 8192 octets.

  * Client A memory maps first page (8192 octets) of file X
  * Client B memory maps first page (8192 octets) of file X
  * Client A write locks first 4096 octets
  * Client B write locks second 4096 octets
  * 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 record 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 record locking, at least until the client unlocks a region in the middle of the file.

Given the above issues the following are permitted:

- Clients and servers MAY deny memory mapping a file they know there are record locks for.
Clients and servers MAY deny a record lock on a file they know is memory mapped.

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.

9.8. 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.

9.9. 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:

- Cached READDIR information for a directory which is not obtained in a single READDIR operation must always be a consistent snapshot of directory contents. This is determined by using a GETATTR before the first READDIR and after the last of READDIR that contributes to the cache.

- An upper time boundary is maintained to indicate the length of time a directory cache entry is considered valid before the client must revalidate the cached information.

The revalidation technique parallels that discussed in the case of name caching. When the client is not changing the directory in question, checking the change attribute of the directory with GETATTR is adequate. The lifetime of the cache entry can be extended at these checkpoints. When a client is modifying the directory, the client needs to use the change_info4 data to determine whether there are other clients modifying the directory. If it is determined that no other client modifications are occurring, the client may update its directory cache to reflect its own changes.

As demonstrated previously, directory caching requires that the client revalidate directory cache data by inspecting the change attribute of a directory at the point when the directory was cached. This requires that the server update the change attribute for directories when the contents of the corresponding directory is modified. For a client to use the change_info4 information appropriately and correctly, the server must report the pre and post operation change attribute values atomically. When the server is unable to report the before and after values atomically with respect to the directory operation, the server must indicate that fact in the change_info4 return value. When the information is not atomically
reported, the client should not assume that other clients have not changed the directory.

10. Multi-Server Name Space

NFSv4.1 supports attributes that allow a namespace to extend beyond the boundaries of a single server. Use of such multi-server namespaces is optional, and for many purposes, single-server namespace are perfectly acceptable. Use of multi-server namespaces can provide many advantages, however, by separating a file system’s logical position in a name space from the (possibly changing) logistical and administrative considerations that result in particular file systems being located on particular servers.

10.1. 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 systems by specifying a server name (either a DNS name or an IP address) together with the path of that file system within that server’s single-server name space.

The fs_locations_info recommended attribute allows specification of one more file systems instance locations where the data corresponding to a given file system may be found. This attribute provides to the client, in addition to information about file system instance locations, extensive information about the various file system instance choices (e.g. priority for use, writability, currency, etc.) as well as information to help the client efficiently effect as seamless a transition as possible among multiple file system instances, when and if that should be necessary.

The fs_locations recommended attribute is inherited from NFSv4.0 and only allows specification of the file system locations where the data corresponding to a given file system may be found. Servers should make this attribute available whenever fs_locations_info is supported, but client use of fs_locations_info is to be preferred.

10.2. 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 or fs_locations_info 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 and any reference to it, except to access a small set of attributes useful in determining alternate locations, will result in an error, NFS4ERR_MOVED. Note that if the server ever returns NFS4ERR_MOVED (i.e. file systems may be absent), it MUST support the fs_locations attribute and SHOULD support the fs_locations_info and fs_absent attributes.

While the error name suggests that we have a case of a file system which 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 file system(s) designated by the location attributes the only realization. The name NFS4ERR_MOVED reflects an earlier, more limited conception of its function, but this error will be returned whenever the referenced file system is absent, whether it has moved or not.

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

The recommended file system attribute fs_absent can used to interrogate the present/absent status of a given file system.
10.3. 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 those that give information about the correct current locations for this file system, fs_locations and fs_locations_info.

10.3.1. 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 one attribute bit that indicates the client is interested in a result regarding an absent file system: fs_locations, fs_locations_info, or fs_absent. If none of these attributes is 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 mandatory will not be available on an absent file system. In addition to the attributes mentioned above (fs_locations, fs_locations_info, fs_absent), the following attributes SHOULD be available on absent file systems, in the case of recommended attributes at least to the same degree that they are available on present file systems.

change: This attribute is useful for absent file systems and can be helpful in summarizing to the client when any of the location-related attributes changes.

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.

mounted_on_fileid: For objects at the top of an absent file system this attribute needs to be available. Since the fileid is one which 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 one of the attributes fs_locations, fs_locations_info, or absent, but where the
bit mask includes attributes which 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, fs_locations_info, or fs_absent, the error NFS4ERRMOVED 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 mandatory attributes), will also cause an NFS4ERRMOVED result.

10.3.2. READDIR and Absent File Systems

A READDIR performed when the current filehandle is within an absent file system will result in an NFS4ERRMOVED 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:

- If the attribute set requested includes one of the attributes fs_locations, fs_locations_info, or fs_absent, then fetching of attributes proceeds normally and no NFS4ERRMOVED indication is returned, even when the rdattr_error attribute is requested.

- If the attribute set requested does not include one of the attributes fs_locations, fs_locations_info, or fs_absent, then if the rdattr_error attribute is requested, each directory entry for the root of an absent file system, will report NFS4ERRMOVED as the value of the rdattr_error attribute.

- If the attribute set requested does not include any of the attributes fs_locations, fs_locations_info, fs_absent, 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 NFSERRMOVED error.

- The unavailability of an attribute because of a file system’s absence, even one that is ordinarily mandatory, 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.
10.4. Uses of Location Information

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

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

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

10.4.1. File System Replication

The fs_locations and fs_locations_info attributes provide alternative locations, to be used to access data in place of or in a addition to the current file system instance. On first access to a file system, the client should obtain the value of the set alternate locations by interrogating the fs_locations or fs_locations_info attribute, with the latter being preferred.

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

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

When multiple server addresses correspond to the same actual server, as shown by a common so_major_id field within the eir_server_owner field returned by EXCHANGE_ID, 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, even in the absence of explicit listing in fs_locations and fs_locations_info. Such corresponding file system locations can be used as alternate locations, just as those explicitly specified via the fs_locations and fs_locations_info attributes. Where these specific locations are designated in the fs_locations_info attribute, the conditions of use specified in this attribute (e.g. priorities, specification of simultaneous use) may limit the clients use of these alternate locations.

When multiple replicas exist and are used simultaneously or in succession by a client, they must designate the same data (with metadata being the same to the degree indicated by the fs_locations_info attribute). Where file systems are writable, a change made on one instance must be visible on all instances, immediately upon the earlier of the return of the modifying request or the visibility of that change on any of the associated replicas. 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. It must be guaranteed that any change visible on the original file system instance must be immediately visible on any replica before the client transitions access to that replica, to avoid any possibility, that a client in effecting a transition to a replica, will see any reversion in file system state. The specific means by which this will be prevented varies based on fs4_status_type reported as part of the fs_status attribute. (See Section 10.11).
10.4.2. File System Migration

When a file system is present and becomes absent, clients can be given the opportunity to have continued access to their data, at an alternate location, as specified by the fs_locations or fs_locations_info attribute. Typically, a client will be accessing the file system in question, get an NFS4ERR_MOVED error, and then use the fs_locations or fs_locations_info attribute to determine the new location of the data. When fs_locations_info is used, additional information will be available which will define the nature of the client’s handling of the transition to a new server.

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 NFSv4.1 protocol specifies the method used to communicate the migration event between client and server.

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

When multiple server addresses correspond to the same actual server, as shown by a common value for so_major_id field of the eir_server_owner field returned by EXCHANGE_ID, the location or locations may designate alternate server addresses in the form of specific server network addresses, when the file system in question is available at those addresses, and no longer accessible at the original address.

Although a single successor location is typical, multiple locations may be provided, together with information that allows priority among the choices to be indicated, via information in the fs_locations_info attribute. Where suitable clustering mechanisms make it possible to provide multiple identical file systems or paths to them, this allows the client the opportunity to deal with any resource or communications issues that might limit data availability.

When an alternate location is designated as the target for migration, it must designate the same data (with metadata being the same to the degree indicated by the fs_locations_info attribute). 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.

10.4.3. Referrals

Referrals provide a way of placing a file system in a location essentially without respect to its physical location on a given server. This allows a single server of 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 that 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 or fs_locations_info 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. The server, in the fs_locations_info attribute may specify priorities to be associated with various file system location choices. The server may assign different priorities to different locations as reported to individual clients, in order to adapt to client physical location or to effect load balancing. When both read-only and read-write file systems are present, some of the read-only locations may not absolutely up-to-date (as they would have to be in the case of replication and migration). Servers may also specify file system locations that include client-substituted variable so that different clients are referred to different file systems (with different data contents) based on client attributes such as cpu architecture.

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 section of the name space. Any segment or the top-level referral file system 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. There are however facilities provided which allow different client to be directed to different sets of data, so as to adapt to such client characteristics as cpu architecture.

10.5. Additional Client-side Considerations

When clients make use of servers that implement referrals, replication, and migration, care should be taken so that a user who mounts a given file system that includes a referral or a relocated file system continue to see a coherent picture of that user-side file system despite the fact that it contains a number of server-side file systems which 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). The client needs to determine when it hits an fsid root going up the file tree. When at such a point, and needs to ascend to the parent, it must do so locally instead of sending a LOOKUPP call to the server. The LOOKUPP would normally return the ancestor of the target file system on the target server, which may not be part of the space that the client mounted.

A related issue is upward navigation from named attribute directories. The named attribute directories are essentially detached from the namespace and this property should be safely represented in the client operating environment. LOOKUPP on a named attribute directory may return the filehandle of the associated file and conveying this to applications might be unsafe as many applications expect the parent of a directory to be a directory by itself. Therefore the client may want to hide the parent of named attribute directories (represented as ".." in UNIX) or represent the named attribute directory as its own parent (as typically done for the file system root directory in UNIX).

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 lookup caching. Clients should
periodically purge this data for referral points in order to detect changes in location information. When the change attribute changes for directories that hold referral entries or for the referral entries themselves, clients should consider any associated cached referral information to be out of date.

10.6. Effecting File System Transitions

Transitions between file system instances, whether due to switching between replicas upon server unavailability, or in response to a server-initiated migration events are best dealt with together. Even though the prototypical use cases of replication and migration contain distinctive sets of features, when all possibilities for these operations are considered, the underlying unity of these operations, from the client’s point of view is clear, even though for the server pragmatic considerations will normally force different implementation strategies for planned and unplanned transitions.

A number of methods are possible for servers to replicate data and to track client state in order to allow clients to transition between file system instances with a minimum of disruption. Such methods vary between those that use inter-server clustering techniques to limit the changes seen by the client, to those that are less aggressive, use more standard methods of replicating data, and impose a greater burden on the client to adapt to the transition.

The NFSv4.1 protocol does not impose choices on clients and servers with regard to that spectrum of transition methods. In fact, there are many valid choices, depending on client and application requirements and their interaction with server implementation choices. The NFSv4.1 protocol does define the specific choices that can be made, how these choices are communicated to the client and how the client is to deal with any discontinuities.

In the sections below, references will be made to various possible server implementation choices as a way of illustrating the transition scenarios that clients may deal with. The intent here is not to define or limit server implementations but rather to illustrate the range of issues that clients may face.

In the discussion below, references will be made to a file system having a particular property or of two file systems (typically the source and destination) belonging to a common class of any of several types. Two file systems that belong to such a class share some important aspect of file system behavior that clients may depend upon when present, to easily effect a seamless transition between file system instances. Conversely, where the file systems do not belong to such a common class, the client has to deal with various sorts of
implementation discontinuities which may cause performance or other issues in effecting a transition.

Where the fs_locations_info attribute is available, such file system classification data will be made directly available to the client. See Section 10.10 for details. When only fs_locations is available, default assumptions with regard to such classifications have to be inferred. See Section 10.9 for details.

In cases in which one server is expected to accept opaque values from the client that originated from another server, it is a wise implementation practice for the servers to encode the "opaque" values in big endian octet order. If this is done, servers acting as replicas or immigrating file systems will be able to parse values like stateids, directory cookies, filehandles, etc. even if their native octet order is different from that of other servers cooperating in the replication and migration of the file system.

10.6.1. File System Transitions and Simultaneous Access

When a single file system may be accessed at multiple locations, whether this is because of an indication of file system identity as reported by the fs_locations or fs_locations_info attributes or because two file systems instances have corresponding locations on server addresses which connect to the same server as indicated by a common so_major_id field in the eir_server_owner field returned by EXCHANGE_ID, the client will, depending on specific circumstances as discussed below, either:

- Access multiple instances simultaneously, as representing alternate paths to the same data and metadata.
- The client accesses one instance (or set of instances) and then transitions to an alternative instance (or set of instances) as a result of network issues, server unresponsiveness, or server-directed migration. The transition may involve changes in filehandles, fileids, the change attribute, and or locking state, depending on the attributes of the source and destination file system instances, as specified in the fs_locations_info attribute.

Which of these choices is possible, and how a transition is effected is governed by equivalence classes of file system instances as reported by the fs_locations_info attribute, and, for file systems instances in the same location within multiple single-server namespace, by the so_major_id field in the eir_server_owner field returned by EXCHANGE_ID.
10.6.2. Simultaneous Use and Transparent Transitions

When two file system instances have the same location within their respective single-server namespaces and those two server IP addresses return the so_major_id value in the eir_server_owner value returned in response to EXCHANGE_ID, those file systems instances can be treated as the same, and either used together simultaneously or serially with no transition activity required on the part of the client.

Whether simultaneous use of the two file system instances is valid is controlled by whether the fs_locations_info attribute shows the two instances as having the same _simultaneous-use_ class.

Note that for two such file systems, any information within the fs_locations_info attribute that indicates the need for special transition activity, i.e. the appearance of the two file system instances with different _handle_, _fileid_, _verifier_, _change_ classes, MUST be ignored by the client. The server SHOULD not indicate that these instances belong to different _handle_, _fileid_, _verifier_, _change_ classes, whether the two instances are shown belonging to the same _simultaneous-use_ class or not.

Where these conditions do not apply, a non-transparent file system instance transition is required with the details depending on the respective _handle_, _fileid_, _verifier_, _change_ classes of the two file system instances and whether the two servers in question have the same eir_server_scope value as reported by EXCHANGE_ID.

10.6.2.1. Simultaneous Use of File System Instances

When the conditions above hold, in either of the following two cases, the client may use the two file system instances simultaneously.

o The fs_locations_info attribute does not contain separate per-IP address entries for file systems instances at the distinct IP addresses. This includes the case in which the fs_locations_info attribute is unavailable.

o The fs_locations_info attribute indicates that two file system instances belong to the same _simultaneous-use_ class.

In this case, the client may use both file system instances simultaneously, as representations of the same file system, whether that happens because the two IP addresses connect to the same physical server or because different servers connect to clustered file systems and export their data in common. When simultaneous use is in effect, any change made to one file system instance must be
immediately reflected in the other file system instance(s). Locks are treated as part of a common lease, associated with a common client ID. Depending on the details of the eir_server_owner returned by EXCHANGE_ID, the two server instances may be accessed by different sessions or a single session in common.

10.6.2.2. Transparent File System Transitions

When the conditions above hold and the fs_locations_info attribute explicitly shows the file system instances for these distinct IP addresses as belonging to different _simultaneous-use_ classes, the file system instances should not be used by the client simultaneously, but rather serially with one being used unless and until communication difficulties, lack of responsiveness, or an explicit migration event causes another file system instance (or set of file system instances sharing a common _simultaneous-use_ class to be used.

When a change in file system instance is to be done, the client will use the same client ID already in effect. If it already has connections to the new server address, these will be used. Otherwise new connections to existing sessions or new sessions associated with the existing client ID are established as indicated by the eir_server_owner returned by EXCHANGE_ID.

In all such transparent transition cases, the following apply:

- File handles stay the same if persistent and if volatile are only subject to expiration, if they would be in the absence of file system transition.

- Fileid values do not change across the transition.

- The file system will have the same fsid in both the old and new locations.

- Change attribute values are consistent across the transition and do not have to be refetched. When change attributes indicate that a cached object is still valid, it can remain cached.

- Client, and state identifier retain their validity across the transition, except where their staleness is recognized and reported by the new server. Except where such staleness requires it, no lock reclamation is needed.

- Write verifiers are presumed to retain their validity and can be presented to COMMIT, with the expectation that if COMMIT on the new server accept them as valid, then that server has all of the
data unstably written to the original server and has committed it to stable storage as requested.

10.6.3. Filehandles and File System Transitions

There are a number of ways in which filehandles can be handled across a file system transition. These can be divided into two broad classes depending upon whether the two file systems across which the transition happens share sufficient state to effect some sort of continuity of file system handling.

When there is no such co-operation in filehandle assignment, the two file systems are reported as being in different _handle_ classes. In this case, all filehandles are assumed to expire as part of the file system transition. Note that this behavior does not depend on fh_expire_type attribute and supersedes the specification of FH4_VOL_MIGRATION bit, which only affects behavior when fs_locations_info is not available.

When there is co-operation in filehandle assignment, the two file systems are reported as being in the same _handle_ classes. In this case, persistent filehandle remain valid after the file system transition, while volatile filehandles (excluding those while are only volatile due to the FH4_VOL_MIGRATION bit) are subject to expiration on the target server.

10.6.4. Fileid’s and File System Transitions

In NFSv4.0, the issue of continuity of fileid’s in the event of a file system transition was not addressed. The general expectation had been that in situations in which the two file system instances are created by a single vendor using some sort of file system image copy, fileid’s will be consistent across the transition while in the analogous multi-vendor transitions they will not. This poses difficulties, especially for the client without special knowledge of the of the transition mechanisms adopted by the server.

It is important to note that while clients themselves may have no trouble with a fileid changing as a result of a file system transition event, applications do typically have access to the fileid (e.g. via stat), and the result of this is that an application may work perfectly well if there is no file system instance transition or if any such transition is among instances created by a single vendor, yet be unable to deal with the situation in which a multi-vendor transition occurs, at the wrong time.

Providing the same fileid’s in a multi-vendor (multiple server vendors) environment has generally been held to be quite difficult.
While there is work to be done, it needs to be pointed out that this difficulty is partly self-imposed. Servers have typically identified fileid with inode number, i.e. with a quantity used to find the file in question. This identification poses special difficulties for migration of an fs between vendors where assigning the same index to a given file may not be possible. Note here that a fileid does not require that it be useful to find the file in question, only that it is unique within the given fs. Servers prepared to accept a fileid as a single piece of metadata and store it apart from the value used to index the file information can relatively easily maintain a fileid value across a migration event, allowing a truly transparent migration event.

In any case, where servers can provide continuity of fileids, they should and the client should be able to find out that such continuity is available, and take appropriate action. Information about the continuity (or lack thereof) of fileid’s across a file system is represented by specifying whether the file systems in question are of the same _fileid_ class.

10.6.5. Fsids and File System Transitions

Since fsids are only unique within a per-server basis, it is to be expected that they will change during a file system transition. Clients should not make the fsid’s received from the server visible to application since they may not be globally unique, and because they may change during a file system transition event. Applications are best served if they are isolated from such transitions to the extent possible.

When a file system transition is made and the fs_locations_info indicates that file system in question may be split into multiple file systems (via the FSLI4F_MULTI_FS flag), client should do GETATTR’s on all known objects within the file system undergoing transition, to determine the new file system boundaries. Clients may maintain the fsid’s passed to existing applications by mapping all of the fsid for the descendent file systems to a the common fsid used for the original file system.

10.6.6. The Change Attribute and File System Transitions

Since the change attribute is defined as a server-specific one, change attributes fetched from one server are normally presumed to be invalid on another server. Such a presumption is troublesome since it would invalidate all cached change attributes, requiring refetching. Even more disruptive, the absence of any assured continuity for the change attribute means that even if the same value is gotten on refetch no conclusions can drawn as to whether the
object in question has changed. The identical change attribute could be merely an artifact, of a modified file with a different change attribute construction algorithm, with that new algorithm just happening to result in an identical change value.

When the two file systems have consistent change attribute formats, and this fact is communicated to the client by reporting as in the same _change_ class, the client may assume a continuity of change attribute construction and handle this situation just as it would be handled without any file system transition.

### 10.6.7. Lock State and File System Transitions

In a file system transition, the client needs to handle cases in which the two servers have cooperated in state management and in which they have not. Cooperation by two servers in state management requires coordination of clientids. Before the client attempts to use a client ID associated with one server in a request to the server of the other file system, it must eliminate the possibility that two non-cooperating servers have assigned the same client ID by accident. The client needs to compare the eir_server_scope values returned by each server. If the scope values do not match, then the servers have not cooperated in state management. If the scope values match, then this indicates the servers have cooperated in assigning clientids to the point that they will reject clientids that refer to state they do not know about.

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. When this done, it must be done in a way that is transparent to the client. With replication, such a degree of common state is typically not the case. Clients, however should use the information provided by the eir_server_scope returned by EXCHANGE_ID to determine whether such sharing may be in effect, rather than making assumptions based on the reason for the transition.

This state transfer will reduce disruption to the client when a file system transition. If the servers are successful in transferring all state, the client can attempt to establish sessions associated with the client ID used for the source file system instance. If the server accepts that as a valid client ID, then the client may use the existing stateid’s associated with that client ID for the old file system instance in connection with the that same client ID in connection with the file system instance.

When the two servers belong to the same server scope, it does necessarily mean that when dealing with the transition, the client will not have to reclaim state. However it does mean that the client
may proceed using his current client ID when establishing
communication with the new server and that that new server will
either recognize that client ID as valid, or reject it, in which case
locks must be reclaimed by the client.

File systems co-operating in state management may actually share
state or simply divide the id space so as to recognize (and reject as
stale) each others state and clients id’s. Servers which do share
state may not do so under all conditions or at all times. The
requirement for the server is that if it cannot be sure in accepting
a client ID that it reflects the locks the client was given, it must
treat all associated state as stale and report it as such to the
client.

When the two file systems instances are on servers that do not share
a server scope value the client must establish a new client ID on the
destination, if it does not have one already and reclaim if possible.
In this case, old stateids and client ID’s should not be presented to
the new server since there is no assurance that they will not
conflict with IDs valid on that server.

In either case, when actual locks are not known to be maintained, the
destination server may establish a grace period specific to the given
file system, with non-reclaim locks being rejected for that file
system, even though normal locks are being granted for other file
systems. Clients should not infer the absence of a grace period for
file systems being transitioned to a server from responses to
requests for other file systems.

In the case of lock reclamation for a given file system after a file
system transition, edge conditions can arise similar to those for
reclaim after server reboot (although in the case of the planned
state transfer associated with migration, these can be avoided by
securely recording lock state as part of state migration. Where the
destination server cannot guarantee that locks will not be
incorrectly granted, the destination server should not establish a
file-system-specific grace period.

In place of a file-system-specific version of RECLAIM_COMPLETE,
servers may assume that an attempt to obtain a new lock, other than
be reclaim, indicate the end of the client’s attempt to reclaim locks
for that file system. [NOTE: The alternative would be to adapt
RECLAIM_COMPLETE to this task].

Information about client identity that may be propagated between
servers in the form of client_owner4 and associated verifiers, under
the assumption that the client presents the same values to all the
servers with which it deals.
Servers are encouraged to provide facilities to allow locks to be reclaimed on the new server after a file system transition. Often, however, in cases in which the two servers do not share a server scope value, such facilities may not be available and client should be prepared to re-obtain locks, even though it is possible that the client may have his LOCK or OPEN request denied due to a conflicting lock. In some environments, such as the transition between read-only 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. In all cases in which the lock is granted, the client cannot assume that no conflicting could have been granted in the interim. Where change attribute continuity is present, the client may check the change attribute to check for unwanted file modifications. Where even this is not available, and the file system is not read-only, a client may reasonably treat all pending locks as having been revoked.

10.6.7.1. Leases and File System Transitions

In the case of lease renewal, the client may not be submitting requests for a file system that has been transferred to another server. This can occur because of the lease renewal mechanism. The client renews leases for all file systems when submitting a request on an associated session, regardless of the specific file system being referenced.

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, the SEQUENCE operation will return the status bit SEQ4_STATUSLEASE_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 NFS4ERRMOVED error and the server receives the subsequent GETATTR for the fs_locations or fs_locations_info attribute for an access to each file system for which a lease has been moved to a new server.

When a client receives an SEQ4_STATUSLEASE_MOVED indication, it should perform an operation on each file system associated with the server in question. When the client receives an NFS4ERRMOVED error, the client can follow the normal process to obtain the new server information (through the fs_locations and fs_locations_info attributes) and perform renewal of those leases on the new server, unless information in fs_locations_info attribute shows that no state could have been transferred. If the server has not had state transferred to it transparently, the client will receive NFS4ERRSTALECLIENTID from the new server, as described above, and
the client can then reclaim locks as is done in the event of server failure.

10.6.7.2. Transitions and the Lease_time Attribute

In order that the client may appropriately manage its leases in the case of a file system transition, 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 transitions 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, either because the associated servers are show as have different eir_server_scope strings or because the client ID is rejected when presented to the new server, 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 lock before potentially conflicting non-reclaimed locks are granted.

10.6.8. Write Verifiers and File System Transitions

In a file system transition, the two file systems may be clustered in the handling of unstably written data. When this is the case, and the two file systems belong to the same _verifier_ class, valid verifiers from one system may be recognized by the other and superfluous writes avoided. There is no requirement that all valid verifiers be recognized, but it cannot be the case that a verifier is recognized as valid when it is not. [NOTE: We need to resolve the issue of proper verifier scope].

When two file systems belong to different _verifier_ classes, the client must assume that all unstable writes in existence at the time file system transition, have been lost since there is no way the old verifier can recognized as valid (or not) on the target server.

10.7. Effecting File System Referrals

Referrals are effected when an absent file system is encountered, and one or more alternate locations are made available by the
fs_locations or fs_locations_info attributes. The client will typically get an NFS4ERR_MOVED error, fetch the appropriate location information and proceed to access the file system on different server, even though it retains its logical position within the original namespace.

The examples given in the sections below are somewhat artificial in that an actual client will not typically do a multi-component lookup, 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.

10.7.1. Referral Example (LOOKUP)

Let us suppose that the following COMPOUND is issued 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.

- PUTROOTFH
- LOOKUP "this"
- LOOKUP "is"
- LOOKUP "the"
- LOOKUP "path"
- GETFH
- GETATTR fsid,fileid,size,ctime

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

- PUTROOTFH --> NFS_OK. The current fh is now the root of the pseudo-fs.
- LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.
- 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 fs, but ... the client will never see the value of that fh.

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

o  GETATTR fsid,fileid,size,ctime. 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 he 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 lookup of "path" succeeded is that the file system was not absent on that op but was moved between the last LOOKUP and the GETFH (since COMPOUND is not atomic). Even if we had the fsid’s for all of the intermediate directories, we could have no way of knowing that /this/is/the/path was the root of a new fs, since we don’t yet have its fsid.

In order to get the necessary information, let us re-issue the chain of lookup’s with GETFH’s and GETATTR’s to at least get the fsid’s so we can be sure where the appropriate fs boundaries are. The client could choose to get fs_locations_info at the same time but in most cases the client will have a good guess as to where fs boundaries are (because of where NFS4ERR_MOVED was gotten and where not) making fetching of fs_locations_info 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 fs 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 fs 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 fs 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 fs, but ...
- The client will never see the value of that fh

OP13: GETATTR(fsid, fs_locations_info) --> NFS_OK
- We are getting the fsid to know where the fs boundaries are. 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 fs might a valid fsid of a different fs 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 another reliable source information on the boundary of the fs which 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_info.

- We are fetching fs_locations_info because the fact that we got an NFS4ERR_MOVED at this point means that it 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 which 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 fs at the start of the operation and the spec makes no exception for GETFH. Note that this has the happy consequence that we don’t have to worry about the volatility or lack thereof of the fh. If the root of the fs on the new location is a persistent fh, then we can assume that this fh, which we never saw is a persistent fh, which, if we could see it, would exactly match the new fh. At least, there is no evidence to disprove that. On the other hand, if we find a volatile root at the new location, then the filehandle which we never saw must have been volatile or at least nobody can prove otherwise.

Given the above, the client knows where the root of the absent file system is, by noting where the change of fsid occurred. The fs_locations_info 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 basically transient information with the
function of enabling the referral.

10.7.2. Referral Example (READDR)

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

Suppose such a directory is read as follows:

- PUTROOTFH
- LOOKUP "this"
- LOOKUP "is"
- LOOKUP "the"
- READDR (fsid, size, ctime, mounted_on_fileid)

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

- PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.
- LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.
- LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.
- LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.
- READDR (fsid, size, ctime, mounted_on_fileid) --> NFS4ERR_MOVED. Note that the same error would have been returned if /this/is/the had migrated, when in fact it is because the directory contains the root of an absent fs.

So now suppose that we reissue with rdattr_error:

- PUTROOTFH
- LOOKUP "this"
o  LOOKUP "is"

o  LOOKUP "the"

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

So suppose we do another READDIR to get fs_locations_info, although we could have used a GETATTR directly, as in the previous section.

o  PUTROOTFH

o  LOOKUP "this"

o  LOOKUP "is"

o  LOOKUP "the"

o  READDIR (rdattr_error, fs_locations_info, mounted_on_fileid, fsid, size, ctime)

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_info, mounted_on_fileid, fsid, size, ctime) --> NFS_OK. The attributes will be as shown below.

The attributes for "path" will only contain

o  rdattr_error (value: NFS4ERRMOVED)

o  fs_locations_info

o  mounted_on_fileid (value: unique fileid within referring fs)

o  fsid (value: unique value within referring server)

The attribute entry for "latest" will not contain size or ctime.

10.8. The Attribute fs_absent

In order to provide the client information about whether the current file system is present or absent, the fs_absent attribute may be interrogated.

As noted above, this attribute, when supported, may be requested of absent file systems without causing NFS4ERRMOVED to be returned and it should always be available. Servers are strongly urged to support this attribute on all file systems if they support it on any file system.

10.9. The Attribute fs_locations

The fs_locations attribute is structured in the following way:

```c
struct fs_location {
    utf8str cis server<>
    pathname4 rootpath;
};
```

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

The fs_location struct is used to represent the location of a file
system by providing a server name and the path to the root of the
file system 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 an UTF8 string and represents one of a
traditional DNS host name, IPv4 address, or IPv6 address. It is not
a requirement that all servers that share the same rootpath be listed
in one fs_location struct. The array of server names is provided for
convenience. Servers that share the same rootpath may also be listed
in separate fs_location entries in the fs_locations attribute.

The fs_locations struct and attribute contains an array of such
locations. Since the name space of each server may be constructed
differently, the "fs_root" field is provided. The path represented
by fs_root represents the location of the file system in the current
server’s name space, 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.

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 name space at "/a/b/c".
When the client switches to servB, it will need to determine that the
directory it first referenced at servA is now represented by the path
"/x/y/z/d" on servB. To facilitate this, the fs_locations attribute
provided by servA would have a fs_root value of "/a/b/c" and two
entries in fs_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.

Since fs_locations attribute lacks information defining various
attributes of the various file system choices presented, it should
only be interrogated and used when fs_locations_info is not
available. When fs_locations is used, information about the specific
locations should be assumed based on the following rules.

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

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

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

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

- When the transition is due to migration, the target should be treated as being of the same _verifier_ class as the source.

- When the transition is due to failover to another replica, the target should be treated as being of a different _verifier_ class from the source.

The specific choices reflect typical implementation patterns for failover and controlled migration respectively. Since other choices are possible and useful, this information is better obtained by using fs_locations_info.

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

10.10. The Attribute fs_locations_info

The fs_locations_info attribute is intended as a more functional replacement for fs_locations which will continue to exist and be supported. Clients can use it get a more complete set of information about alternative file system locations. When the server does not support fs_locations_info, fs_locations can be used to get a subset of the information. A server which supports fs_locations_info MUST support fs_locations as well.
There is additional information present in fs_locations_info, that is not available in fs_locations:

- Attribute continuity information to allow a client to select a location which meets the transparency requirements of the applications accessing the data and to take advantage of optimizations that server guarantees as to attribute continuity may provide (e.g. change attribute).

- File System identity information which indicates when multiple replicas, from the clients point of view, correspond to the same target file system, allowing them to be used interchangeably, without disruption, as multiple paths to the same thing.

- Information which will bear on the suitability of various replicas, depending on the use that the client intends. For example, many applications need an absolutely up-to-date copy (e.g. those that write), while others may only need access to the most up-to-date copy reasonably available.

- Server-derived preference information for replicas, which can be used to implement load-balancing while giving the client the entire fs list to be used in case the primary fails.

The fs_locations_info attribute consists of a root pathname (just like fs_locations), together with an array of fs_location_item4 structures.
struct fs_locations_server4 {
    int32_t fls_currency;
    opaque fls_info<>;
    utf8str cis fls_server;
};

const FSLI4BX_GFLAGS = 0;
const FSLI4BX_TFLAGS = 1;
const FSLI4BX_CLSIMUL = 2;
const FSLI4BX_CLHANDLE = 3;
const FSLI4BX_CLFILEID = 4;
const FSLI4BX_CLVERIFIER = 5;
const FSLI4BX_CHANGE = 6;
const FSLI4BX_READRANK = 7;
const FSLI4BX_WRITERANK = 8;
const FSLI4BX_READORDER = 9;
const FSLI4BX_WRITEORDER = 10;
const FSLI4GF_WRITABLE = 0x01;
const FSLI4GF_CUR_REQ = 0x02;
const FSLI4GF_ABSENT = 0x04;
const FSLI4GF_GOING = 0x08;
const FSLI4GF_SPLIT = 0x10;
const FSLI4TF_RDMA = 0x01;

struct fs_locations_item4 {
    fs_locations_server4 fli_entries<>;
    pathname4 fli_rootpath;
};

struct fs_locations_info4 {
    uint32_t fli_flags;
    pathname4 fli_fs_root;
    fs_locations_item4 fli_items<>;
};

cast FSLI4IF_VAR_SUB = 0x00000001;

typedef fs_locations_info4 fattr4_fs_locations_info;

The fs_locations_info attribute is structured similarly to the fs_locations attribute. A top-level structure (fs_locations_info4) contains the entire attribute including the root pathname of the fs and an array of lower-level structures that define replicas that share a common root path on their respective servers. The lower-
level structure in turn (fs_locations_item4) contain a specific pathname and information on one or more individual server replicas. For that last lowest-level fs_locations_info has a fs_locations_server4 structure that contains per-server-replica information in addition to the server name.

As noted above, the fs_locations_info attribute, when supported, may be requested of absent file systems without causing NFS4ERR_MOVED to be returned and it is generally expected that it will be available for both present and absent file systems even if only a single fs_locations_server4 entry is present, designating the current (present) file system, or two fs_locations_server4 entries designating the current (and now previous) location of an absent file system and its successor location. Servers are strongly urged to support this attribute on all file systems if they support it on any file system.

10.10.1. The fs_locations_server4 Structure

The fs_locations_server4 structure consists of the following items:

- An indication of file system up-to-date-ness (fls_currency) in terms of approximate seconds before the present. A negative value indicates that the server is unable to give any reasonably useful value here. A zero indicates that file system is the actual writable data or a reliably coherent and fully up-to-date copy. Positive values indicate how out-of-date this copy can normally be before it is considered for update. Such a value is not a guarantee that such updates will always be performed on the required schedule but instead serve as a hint about how far behind the most up-to-date copy of the data, this copy would normally be expected to be.

- A counted array of one-octet values (fls_info) containing information about the particular file system instance. This data includes general flags, transport capability flags, file system equivalence class information, and selection priority information. The encoding will be discussed below.

- The server string (fls_server). For the case of the replica currently being accessed (via GETATTR), a null string may be used to indicate the current address being used for the RPC call.

Data within the fls_info array, is in the form of 8-bit data items with constants giving the offsets within the array of various values describing this particular file system instance. This style of definition was chosen, in preference to explicit XDR structure definitions for these values for a number of reasons.
The kinds of data in the fls_info array, representing, flags, file system classes and priorities among set of file systems representing the same data are such that eight bits provides a quite acceptable range of values. Even where there might be more than 256 such file system instances, having more than 256 distinct classes or priorities is unlikely.

Explicit definition of the various specific data items within XDR would limit expandability in that any extension within a subsequent minor version would require yet another attribute, leading to specification and implementation clumsiness.

Such explicit definitions would also make it impossible to propose standards-track extensions apart from a full minor version.

This encoding scheme can be adapted to the specification of multi-octet numeric values, even though none are currently defined. If extensions are made via standards-track RFC’s, multi-octet quantities will be encoded as a range of octet with a range of indices with the octet interpreted in big endian octet order.

The set of fls_info data is subject to expansion in a future minor version, or in a standard-track RFC, within the context of a single minor version. The server SHOULD NOT send and the client MUST not use indices within the fls_info array that are not defined in standards-track RFC’s.

The fls_info array contains within it:

- Two 8-bit flag fields, one devoted to general file-system characteristics and a second reserved for transport-related capabilities.
- Four 8-bit class values which define various file system equivalence classes as explained below.
- Four 8-bit priority values which govern file system selection as explained below.

The general file system characteristics flag (at octet index FSLI4BX_GFLAGS) has the following bits defined within it:

- FSLI4GF_WRITABLE indicates that this fs target is writable, allowing it to be selected by clients which may need to write on this file system. When the current file system instance is writable, then any other file system to which the client might switch must incorporate within its data any committed write made on the current file system instance. See the section on verifier...
class, for issues related to uncommitted writes. While there is no harm in not setting this flag for a file system that turns out to be writable, turning the flag on for read-only file system can cause problems for clients who select a migration or replication target based on it and then find themselves unable to write.

- FSLI4GF_CUR_REQ indicates that this replica is the one on which the request is being made. Only a single server entry may have this flag set and in the case of a referral, no entry will have it.

- FSLI4GF_ABSENT indicates that this entry corresponds an absent file system replica. It can only be set if FSLI4GF_CUR_REQ is set. When both such bits are set it indicates that a file system instance is not usable but that the information in the entry can be used to determine the sorts of continuity available when switching from this replica to other possible replicas. Since this bit can only be true if FSLI4GF_CUR_REQ is true, the value could be determined using the fs_absent attribute but the information is also made available here for the convenience of the client. An entry with this bit, since it represents a true file system (albeit absent) does not appear in the event of a referral, but only where a file system has been accessed at this location and subsequently been migrated.

- FSLI4GF_GOING indicates that a replica, while still available, should not be used further. The client, if using it, should make an orderly transfer to another file system instance as expeditiously as possible. It is expected that file systems going out of service will be announced as FSLI4GF_GOING some time before the actual loss of service and that the valid_for value will be sufficiently small to allow clients to detect and act on scheduled events while large enough that the cost of the requests to fetch the fs_locations_info values will not be excessive. Values on the order of ten minutes seem reasonable.

- FSLI4GF_SPLIT indicates that when a transition occurs from the current file system instance to this one, the replacement may consist of multiple file systems. In this case, the client has to be prepared for the possibility that objects on the same fs before migration will be on different ones after. Note that FSLI4GF_SPLIT is not incompatible with the file systems belong to the same _fileid_ class since, if one has a set of fileid's that are unique within an fs, each subset assigned to a smaller fs after migration would not have any conflicts internal to that fs.

A client, in the case of a split file system will interrogate existing files with which it has continuing connection (it is free
simply forget cached filehandles). If the client remembers the directory filehandle associated with each open file, it may proceed upward using LOOKUPP to find the new fs boundaries.

Once the client recognizes that one file system has been split into two, it could maintain applications running without disruption by presenting the two file systems as a single one until a convenient point to recognize the transition, such as a reboot. This would require a mapping of fsids from the server’s fsids to fsids as seen by the client but this is already necessary for other reasons. As noted above, existing fileids within the two descendant fs’s will not conflict. Creation of new files in the two descendent fs’s may require some amount of fileid mapping which can be performed very simply in many important cases.

The transport-flag field (at octet index FSLI4BX_TFLAGS) contains the following bits related to the transport capabilities of the specific file system.

- FSLI4TF_RDMA indicates that this file system provides NFSv4.1 file system access using an RDMA-capable transport.

Attribute continuity and file system identity information are expressed by defining equivalence relations on the sets of file systems presented to the client. Each such relation is expressed as a set of file system equivalence classes. For each relation, a file system has an 8-bit class number. Two file systems belong to the same class if both have identical non-zero class numbers. Zero is treated as non-matching. Most often, the relevant question for the client will be whether a given replica is identical-to/continuous-with the current one in a given respect but the information should be available also as to whether two other replicas match in that respect as well.

The following fields specify the file system’s class numbers for the equivalence relations used in determining the nature of file system transitions. See Section 10.6 for details about how this information is to be used.

- The field with octet index FSLI4BX_CLSIMUL defines the simultaneous-use class for the file system.
- The field with octet index FSLI4BX_CLHANDLE defines the handle class for the file system.
- The field with octet index FSLI4BX_CLFILEID defines the fileid class for the file system.
The field with octet index FSLI4BX_CLVERIFIER defines the verifier class for the file system.

The field with octet index FSLI4BX_CLCHANGE defines the change class for the file system.

Server-specified preference information is also provided via 8-bit values within the fls_info array. The values provide a rank and an order (see below) to be used with separate values specifiable for the cases of read-only and writable file systems. These values are compared for different file systems to establish the server-specified preference, with lower values indicating "more preferred".

Rank is used to express a strict server-imposed ordering on clients, with lower values indicating "more preferred." Clients should attempt to use all replicas with a given rank before they use one with a higher rank. Only if all of those file systems are unavailable should the client proceed to those of a higher rank.

Within a rank, the order value is used to specify the server’s preference to guide the client’s selection when the client’s own preferences are not controlling, with lower values of order indicating "more preferred." If replicas are approximately equal in all respects, clients should defer to the order specified by the server. When clients look at server latency as part of their selection, they are free to use this criterion but it is suggested that when latency differences are not significant, the server-specified order should guide selection.

The field at octet index FSLI4BX_READRANK gives the rank value to be used for read-only access.

The field at octet index FSLI4BX_READOREDER gives the order value to be used for read-only access.

The field at octet index FSLI4BX_WRITERANK gives the rank value to be used for writable access.

The field at octet index FSLI4BX_WRITEOREDER gives the order value to be used for writable access.

Depending on the potential need for write access by a given client, one of the pairs of rank and order values is used. The read rank and order should only be used if the client knows that only reading will ever be done or if it is prepared to switch to a different replica in the event that any write access capability is required in the future.
10.10.2. The fs_locations_info4 Structure

The fs_locations_info4 structure, encoding the fs_locations_info attribute, contains the following:

- The fli_flags field which contains general flags that affect the interpretation of this fs_locations_info4 structure and all fs_locations_item4 structures within it. The only flag currently defined is FSLI4IF_VAR_SUB. All bits in the fli_flags field which are not defined should always be returned as zero.

- The fli_fs_root field which contains the pathname of the root of the current file system on the current server, just as it does the fs_locations4 structure.

- An array called fli_items of fs_locations4_item structures, which contain information about replicas of the current file system. Where the current file system is actually present, or has been present, i.e. this is not a referral situation, one of the fs_locations_item4 structures will contain an fs_locations_server4 for the current server. This structure will have FSLI4GF_ABSENT set if the current file system is absent, i.e. normal access to it will return NFS4ERRMOVED.

- The fli_valid_for field specifies a time in seconds for which it is reasonable for a client to use the fs_locations_info attribute without refetch. The fli_valid_for value does not provide a guarantee of validity since servers can unexpectedly go out of service or become inaccessible for any number of reasons. Clients are well-advised to refetch this information for actively accessed file system at every fli_valid_for seconds. This is particularly important when file system replicas may go out of service in a controlled way using the FSLI4GF_GOING flag to communicate an ongoing change. The server should set fli_valid_for to a value which allows well-behaved clients to notice the FSLI4GF_GOING flag and make an orderly switch before the loss of service becomes effective. If this value is zero, then no refetch interval is appropriate and the client need not refetch this data on any particular schedule. In the event of a transition to a new file system instance, a new value of the fs_locations_info attribute will be fetched at the destination and it is to be expected that this may have a different valid_for value, which the client should then use, in the same fashion as the previous value.

The FSLI4IF_VAR_SUB flag within fli_flags controls whether variable substitution is to be enabled.
10.10.3. The fs_locations_item4 Structure

The fs_locations_item4 structure contains a pathname (in the field fli_rootpath) which encodes the path of the target file system replicas on the set of servers designated by the included fs_locations_server4 entries. The precise manner in which this target location is specified depends on the value of the FSLI4IF_VAR_SUB flag within the associated fs_locations_info4 structure.

If this flag is not set, then fli_rootpath simply designates the location of the target file system within each server’s single-server namespace just as it does for the rootpath within the fs_location structure. When this bit is set, however, component entries of a certain form are subject to client-specific variable substitution so as to allow a degree of namespace non-uniformity in order to accommodate the selection of client-specific file system targets to adapt to different client architectures or other characteristics.

When such substitution is in effect a variable beginning with the string "${" and ending with the string "}" and containing a colon is to be replaced by the client-specific value associated with that variable. The string "unknown" should be used by the client when it has no value for such a variable. The pathname resulting from such substitutions is used to designate the target file system, so that different clients may have different file systems, corresponding to that location in the multi-server namespace.

As mentioned above, such substituted pathname variables contain a colon. The part before the colon is to be a DNS domain name with the part after being a case-insensitive alphanumeric string.

Where the domain is "ietf.org", only variable names defined in this document or subsequent standards-track RFC’s are subject to such substitution. Organizations are free to use their domain names to create their own sets of client-specific variables, to be subject to such substitution. In case where such variables are intended to be used more broadly than a single organization, publication of an informational RFC defining such variables is recommended.

The variable ${ietf.org:CPU_ARCH} is used to denote the CPU architecture object files are compiled. This specification does not limit the acceptable values (except that they must be valid UTF-8 strings) but such values as "x86", "x86_64" and "sparc" would be expected to be used in line with industry practice.

The variable ${ietf.org:OS_TYPE} is used to denote the operating system and thus the kernel and library API’s for which code might be
compiled. This specification does not limit the acceptable values (except that they must be valid UTF-8 strings) but such values as "linux" and "freebsd" would be expected to be used in line with industry practice.

The variable `${ietf.org:OS_VERSION}` is used to denote the operating system version and the thus the specific details of versioned interfaces for which code might be compiled. This specification does not limit the acceptable values (except that they must be valid UTF-8 strings) but combinations of numbers and letters with interspersed dots would be expected to be used in line with industry practice, with the details of the version format depending on the specific value of the variable `${ietf.org:OS_TYPE}` with which it is used.

Use of these variable could result in direction of different clients to different file systems on the same server, as appropriate to particular clients. In cases in which the target file systems are located on different servers, a single server could serve as a referral point so that each valid combination of variable values would designate a referral hosted on a single server, with the targets of those referrals on a number of different servers.

Although variable substitution is most suitable for use in the context of referrals, it may be used in the context of replication and migration. If it is used in these contexts, the server must ensure that no matter what values the client presents for the substituted variables, the result is always a valid successor file system instance to that from which a transition is occurring, i.e. that the data is identical or represents a later image of a writable file system.

Note that when fli_rootpath is a null pathname (that is, one with zero components), the file system designated is at the root of the specified server, whether the FS Li4_IF_VAR_SUB flag within the associated fs_locations_info4 structure is set or not.

10.11. The Attribute fs_status

In an environment in which multiple copies of the same basic set of data are available, information regarding the particular source of such data and the relationships among different copies, can be very helpful in providing consistent data to applications.
enum fs4_status_type {
    STATUS4_FIXED = 1,
    STATUS4_VERSIONED = 2,
    STATUS4_UPDATED = 3,
    STATUS4_WRITABLE = 4,
    STATUS4_ABSENT = 5
};

struct fs4_status {
    fs4_status_type fsstat_type;
    utf8str_cs      fsstat_source;
    utf8str_cs      fsstat_current;
    int32_t         fsstat_age;
    nfstime4        fsstat_version;
};

The type value indicates the kind of file system image represented. This is of particular importance when using the version values to determine appropriate succession of file system images. Five types are distinguished:

- **STATUS4_FIXED** which indicates a read-only image in the sense that it will never change. The possibility is allowed that as a result of migration or switch to a different image, changed data can be accessed but within the confines of this instance, no change is allowed. The client can use this fact to aggressively cache.

- **STATUS4_VERSIONED** which indicates that the image, like the **STATUS4_UPDATED** case, is updated exogenously, but it provides a guarantee that the server will carefully update an associated version value so that the client can protect itself from a situation in which it reads data from one version of the file system, and then later reads data from an earlier version of the same file system. See below for a discussion of how this can be done.

- **STATUS4_UPDATED** which indicates an image that cannot be updated by the user writing to it but may be changed exogenously, typically because it is a periodically updated copy of another writable file system somewhere else. In this case, version information is not provided and the client does not have the responsibility of making sure that this version only advances upon a file system instance transition. In this case, it is the responsibility of the server to make sure that the data presented after a file system instance transition is a proper successor image and includes all changes seen by the client and any change made before all such changes.
o STATUS4_WRITABLE which indicates that the file system is an actual writable one. The client need not of course actually write to the file system, but once it does, it should not accept a transition to anything other than a writable instance of that same file system.

o STATUS4_ABSENT which indicates that the information is the last valid for a file system which is no longer present.

The opaque strings source and current provide a way of presenting information about the source of the file system image being present. It is not intended that client do anything with this information other than make it available to administrative tools. It is intended that this information be helpful when researching possible problems with a file system image that might arise when it is unclear if the correct image is being accessed and if not, how that image came to be made. This kind of debugging information will be helpful, if, as seems likely, copies of file systems are made in many different ways (e.g. simple user-level copies, file system-level point-in-time copies, cloning of the underlying storage), under a variety of administrative arrangements. In such environments, determining how a given set of data was constructed can be very helpful in resolving problems.

The opaque string 'source' is used to indicate the source of a given file system with the expectation that tools capable of creating a file system image propagate this information, when that is possible. It is understood that this may not always be possible since a user-level copy may be thought of as creating a new data set and the tools used may have no mechanism to propagate this data. When a file system is initially created associating with it data regarding how the file system was created, where it was created, by whom, etc. can be put in this attribute in a human-readable string form so that it will be available when propagated to subsequent copies of this data.

The opaque string 'current' should provide whatever information is available about the source of the current copy. Such information as the tool creating it, any relevant parameters to that tool, the time at which the copy was done, the user making the change, the server on which the change was made etc. All information should be in a human-readable string form.

The age provides an indication of how out-of-date the file system currently is with respect to its ultimate data source (in case of cascading data updates). This complements the fls_currency field of fs_locations_server4 (See Section 10.10) in the following way: the information in fls_currency gives a bound for how out of date the data in a file system might typically get, while the age gives a
bound on how out of date that data actually is. Negative values imply no information is available. A zero means that this data is known to be current. A positive value means that this data is known to be no older than that number of seconds with respect to the ultimate data source.

The version field provides a version identification, in the form of a time value, such that successive versions always have later time values. When the file system type is anything other than STATUS4_VERSIONED, the server may provide such a value but there is no guarantee as to its validity and clients will not use it except to provide additional information to add to ‘source’ and ‘current’.

When the type is STATUS4_VERSIONED, servers should provide a value of version which progresses monotonically whenever any new version of the data is established. This allows the client, if reliable image progression is important to it, to fetch this attribute as part of each COMPOUND where data or metadata from the file system is used.

When it is important to the client to make sure that only valid successor images are accepted, it must make sure that it does not read data or metadata from the file system without updating its sense of the current state of the image, to avoid the possibility that the fs_status which the client holds will be one for an earlier image, and so accept a new file system instance which is later than that but still earlier than updated data read by the client.

In order to do this reliably, it must do a GETATTR of fs_status that follows any interrogation of data or metadata within the file system in question. Often this is most conveniently done by appending such a GETATTR after all other operations that reference a given file system. When errors occur between reading file system data and performing such a GETATTR, care must be exercised to make sure that the data in question is not used before obtaining the proper fs_status value. In this connection, when an OPEN is done within such a versioned file system and the associated GETATTR of fs_status is not successfully completed, the open file in question must not be accessed until that fs_status is fetched.

The procedure above will ensure that before using any data from the file system the client has in hand a newly-fetched current version of the file system image. Multiple values for multiple requests in flight can be resolved by assembling them into the required partial order (and the elements should form a total order within it) and using the last. The client may then, when switching among file system instances, decline to use an instance which is not of type STATUS4_VERSIONED or whose version field is earlier than the last one obtained from the predecessor file system instance.
11. Directory Delegations

11.1. Introduction to Directory Delegations

Directory caching for the NFSv4.1 protocol is similar to previous versions. Clients typically cache directory information for a duration determined by the client. At the end of a predefined timeout, the client will query the server to see if the directory has been updated. By caching attributes, clients reduce the number of GETATTR calls made to the server to validate attributes. Furthermore, frequently accessed files and directories, such as the current working directory, have their attributes cached on the client so that some NFS operations can be performed without having to make an RPC call. By caching name and inode information about most recently looked up entries in the Directory Name Lookup Cache (DNLC), clients do not need to send LOOKUP calls to the server every time these files are accessed.

This caching approach works reasonably well at reducing network traffic in many environments. However, it does not address environments where there are numerous queries for files that do not exist. In these cases of "misses", the client must make RPC calls to the server in order to provide reasonable application semantics and promptly detect the creation of new directory entries. Examples of high miss activity are compilation in software development environments. The current behavior of NFS limits its potential scalability and wide-area sharing effectiveness in these types of environments. Other distributed stateful file system architectures such as AFS and DFS have proven that adding state around directory contents can greatly reduce network traffic in high miss environments.

Delegation of directory contents is a RECOMMENDED feature of NFSv4.1. Directory delegations provide similar traffic reduction benefits as with file delegations. By allowing clients to cache directory contents (in a read-only fashion) while being notified of changes, the client can avoid making frequent requests to interrogate the contents of slowly-changing directories, reducing network traffic and improving client performance.

Directory delegations allow improved namespace cache consistency to be achieved through delegations and synchronous recalls alone without asking for notifications. In addition, if time-based consistency is sufficient, asynchronous notifications can provide performance benefits for the client, and possibly the server, under some common operating conditions such as slowly-changing and/or very large directories.
11.2. Directory Delegation Design

NFSv4.1 introduces the GET_DIR_DELEGATION (Section 17.39) operation to allow the client to ask for a directory delegation. The delegation covers directory attributes and all entries in the directory. If either of these change the delegation will be recalled synchronously. The operation causing the recall will have to wait before the recall is complete. Any changes to directory entry attributes will not cause the delegation to be recalled.

In addition to asking for delegations, a client can also ask for notifications for certain events. These events include changes to directory attributes and/or its contents. If a client asks for notification for a certain event, the server will notify the client when that event occurs. This will not result in the delegation being recalled for that client. The notifications are asynchronous and provide a way of avoiding recalls in situations where a directory is changing enough that the pure recall model may not be effective while trying to allow the client to get substantial benefit. In the absence of notifications, once the delegation is recalled the client has to refresh its directory cache which might not be very efficient for very large directories.

The delegation is read only and the client may not make changes to the directory other than by performing NFSv4 operations that modify the directory or the associated file attributes so that the server has knowledge of these changes. In order to keep the client namespace synchronized with the server, the server will notify the client holding the delegation of the changes made as a result. This is to avoid any subsequent GETATTR or READDIR calls to the server. If a single client is holding the delegation and that client makes any changes to the directory, the delegation will not be recalled. Multiple clients may hold a delegation on the same directory, but if any such client modifies the directory, the server MUST recall the delegation from the other clients.

Delegations can be recalled by the server at any time. Normally, the server will recall the delegation when the directory changes in a way that is not covered by the notification, or when the directory changes and notifications have not been requested.

Also if the server notices that handing out a delegation for a directory is causing too many notifications or recalls to be sent out, it may decide not to hand out a delegation for that directory or recall existing delegations. If another client removes the directory for which a delegation has been granted, the server will recall the delegation.
11.3. Attributes in Support of Directory Notifications

See Section 5.12 for a description of the attributes associated with directory notifications.

11.4. Delegation Recall

The server will recall the directory delegation by sending a callback to the client. It will use the same callback procedure as used for recalling file delegations. The server will recall the delegation when the directory changes in a way that is not covered by the notification. However, the server will not recall the delegation if attributes of an entry within the directory change. Also, if the server notices that handing out a delegation for a directory is causing too many notifications to be sent out, it may decide not to hand out a delegation for that directory. If another client tries to remove the directory for which a delegation has been granted, the server will recall the delegation.

The server will recall the delegation by sending a CB_RECALL callback to the client. If the recall is done because of a directory changing event, the request making that change will need to wait while the client returns the delegation.

11.5. Directory Delegation Recovery

Crash recovery for state on regular files has two main goals, avoiding the necessity of breaking application guarantees with respect to locked files and delivery of updates cached at the client. Neither of these applies to directories protected by read delegations and notifications. Thus, the client is required to establish a new delegation on a server or client reboot. [[Comment.15: we have special reclaim types allow clients to recovery delegations through client reboot. Do we really want EXCHANGE_ID/CREATE_SESSION to destroy directory delegation state?]]

12. Parallel NFS (pNFS)

12.1. Introduction

PNFS is a set of OPTIONAL features of NFSv4.1 which allow direct client access to the storage devices containing the file data. When file data for a single NFSv4 server is stored on multiple and/or higher throughput storage devices (by comparison to the server's throughput capability), the result can be significantly better file access performance. The relationship among multiple clients, a single server, and multiple storage devices for pNFS (server and
clients have access to all storage devices) is shown in this diagram:

![Diagram of NFSv4 + pNFS](image)

Figure 62

In this structure, the responsibility for coordination of file access by multiple clients is shared among the server, clients, and storage devices. This is in contrast to NFSv4 without pNFS in which this is primarily the server’s responsibility, some of which can be delegated to clients under strictly specified conditions.

PNFS takes the form of OPTIONAL operations that manage data location information called a layout. The layout is managed in a similar fashion as NFSv4 data delegations (e.g., they are recallable and revocable). However, they are distinct abstractions and are manipulated with new operations. When a client holds a layout, it has rights to access the data directly using the location information in the layout.

This document specifies the use of NFSv4.1 as a storage protocol. PNFS allows other storage protocols, and these protocols are deliberately not specified here. These might include:

- Block/volume protocols such as iSCSI ([29]), and FCP ([30]). The block/volume protocol support can be independent of the addressing structure of the block/volume protocol used, allowing more than one protocol to access the same file data and enabling extensibility to other block/volume protocols.

- Object protocols such as OSD over iSCSI or Fibre Channel [31].
With some storage protocols, the storage devices cannot perform fine-grained access checks to ensure that clients are only performing accesses within the bounds permitted to them by the pNFS operations with the server (e.g., the checks may only be possible at file system granularity rather than file granularity). In situations where this added responsibility placed on clients creates unacceptable security risks, pNFS configurations in which storage devices cannot perform fine-grained access checks SHOULD NOT be used. All pNFS server implementations MUST support NFSv4.1 access to any file accessible via pNFS in order to provide an interoperable means of file access in such situations. See Section 12.9 on Security for further discussion.

There are issues about how layouts interact with the existing NFSv4 abstractions of data delegations and record locking. Delegation issues are discussed in Section 12.5.4. Byte range locking issues are discussed in Section 12.2.10 and Section 12.5.1.

12.2. PNFS Definitions

PNFS partitions the NFSv4.1 file system protocol into two parts, the metadata path and the data path. The metadata path is implemented by a metadata server that supports pNFS and the operations described in this document (Section 17). The data path is implemented by a storage device that supports the storage protocol. A subset (defined in Section 13.7) of NFSv4.1 is one such storage protocol. This leads to new terms used to describe the protocol extension and some clarifications of existing terms.

12.2.1. Metadata

This is information about a file, such as its name, owner, where it stored, and so forth. Metadata also includes lower-level information like block addresses and indirect block pointers.

12.2.2. Metadata Server

A pNFS metadata server is an NFSv4.1 server which supports pNFS operations and features. When supporting pNFS the metadata server might hold only the metadata associated with a file, while the data can be stored on the storage devices. However, data may also be written through the metadata server which in turn ensures data is written to the storage devices.
12.2.3. Client

A pNFS client is a NFSv4.1 client as defined by this document, which supports pNFS operations and features, and supports least one storage protocol for performing I/O directly to storage devices.

12.2.4. Storage Device

A storage device controls a regular file’s data, but leaves other metadata management up to the metadata server. A storage device could be another NFSv4.1 server, an object storage device (OSD), a block device accessed over a SAN (e.g., either FiberChannel or iSCSI SAN), or some other entity.

12.2.5. Data Server

A data server is a storage device that is implemented by a server of higher level storage access protocol, such as NFSv4.1.

12.2.6. Storage Protocol or Data Protocol

A storage protocol or data protocol is the used between the pNFS client and the storage device to access the file data. Three layout types have been described: file protocols (i.e., NFSv4.1), object protocols (e.g., OSD), and block/volume protocols (e.g., based on SCSI-block commands). These protocols are in turn realizable over a variety of transport stacks.

Depending the storage protocol, block-level metadata may or may not be managed by the metadata server, but is instead managed by object storage devices or other servers acting as a storage device.

12.2.7. Control Protocol

The control protocol is used by the exported file system between the metadata server and storage devices. Specification of such protocols is outside the scope of this document. Such control protocols would be used to control such activities as the allocation and deallocation of storage and the management of state required by the data servers to perform client access control.

While pNFS allows for any control protocol, in practice the control protocol is closely related to the storage protocol. For example, if the data servers are NFSv4.1 servers, then the protocol between the metadata server and the data servers is likely to involve NFSv4.1 operations. Similarly, when object storage devices are used, the pNFS metadata server will likely use iSCSI/OSD commands to manipulate storage.
Regardless, this document does not mandate any particular control protocol. Instead, it just describes the requirements on the control protocol for maintaining attributes like modify time, the change attribute, and the end-of-file (EOF) position.

12.2.8. Layout

A layout defines how a file’s data is organized on one or more storage devices. There are many possible layout types. They vary in the storage protocol used to access the data, and in the aggregation scheme that lays out the file data on the underlying storage devices. A layout is more precisely identified by the following tuple: <Client, filehandle, layout type>; where filehandle refers to the filehandle of the file on the metadata server. Layouts describe a file, not an octet-range of a file; Section 12.2.11 describes layout segments which do pertain to a range.

12.2.9. Layout Types

A layout describes the mapping of a file’s data to the storage devices that hold the data. A layout is said to belong to a specific layout type (data type layouttype4, see Section 3.2.15). The layout type allows for variants to handle different storage protocols, such as block/volume [24], object [23], and file (Section 13) layout types. A metadata server, along with its control protocol, MUST support at least one layout type. A private sub-range of the layout type name space is also defined. Values from the private layout type range can be used for internal testing or experimentation.

As an example, a file layout type could be an array of tuples (e.g., deviceID, file_handle), along with a definition of how the data is stored across the devices (e.g., striping). A block/volume layout might be an array of tuples that store <deviceID, block_number, block count> along with information about block size and the file offset of the first block. An object layout might be an array of tuples <deviceID, objectID> and an additional structure (i.e., the aggregation map) that defines how the logical octet sequence of the file data is serialized into the different objects. Note, the actual layouts are more complex than these simple expository examples.

12.2.10. Layout Iomode

The layout iomode (data type layoutiomode4, see Section 3.2.23) indicates to the metadata server the client’s intent to perform either just READ operations (Section 17.22) or a mixture of I/O possibly containing WRITE (Section 17.32) and READ operations. For certain layout types, it is useful for a client to specify this intent at LAYOUTGET (Section 17.43) time. E.g., for block/volume
based protocols, block allocation could occur when a READ/WRITE
iomode is specified. A special LAYOUTIOMODE4_ANY iomode is defined
and can only be used for LAYOUTRETURN and LAYOUTRECALL, not for
LAYOUTGET. It specifies that layouts pertaining to both READ and
READ/WRITE iomodes are being returned or recalled, respectively.

A storage device may validate I/O with regards to the iomode; this is
dependent upon storage device implementation and layout type. Thus,
if the client’s layout iomode differs from the I/O being performed,
the storage device may reject the client’s I/O with an error
indicating a new layout with the correct I/O mode should be fetched.
E.g., if a client gets a layout with a READ iomode and performs a
WRITE to a storage device, the storage device is allowed to reject
that WRITE.

The iomode does not conflict with OPEN share modes or lock requests;
open mode checks and lock enforcement are always enforced, and are
logically separate from the pNFS layout level. As well, open modes
and locks are the preferred method for restricting user access to
data files. E.g., an OPEN of read, deny-write does not conflict with
a LAYOUTGET containing an iomode of READ/WRITE performed by another
client. Applications that depend on writing into the same file
concurrently may use record locking to serialize their accesses.

12.2.11. Layout Segment

Since a layout that describes an entire file may be very large, there
is a desire to manage layouts in smaller chunks that correspond to
octet-ranges of the file. For example, the entire layout need not be
returned, recalled, or committed. These chunks are called layout
segments and are further identified by the octet-range and iomode
they represent, yielding a layout segment identifier consisting of
<client ID, filehandle, layout type, range, iomode>. The concepts of
a layout and its layout segments allow clients and metadata servers
to aggregate the results of layout operations into a singly
maintained layout.

It is important to define when layout segments overlap and/or
conflict with each other. For two layout segments with overlapping
octet ranges to actually overlap each other, both segments must be of
the same layout type, correspond to the same filehandle, and have the
same iomode. Layout segments conflict, when they overlap and differ
in the content of the layout (i.e., the storage device/file mapping
parameters differ). Note, differing iomodes do not lead to
conflicting layouts. It is permissible for layout segments with
different iomodes, pertaining to the same octet range, to be held by
the same client.
12.2.12. Device IDs

The device ID (data type deviceid4, see Section 3.2.16) names a storage device. In practice, a significant amount of information may be required to fully address a storage device. Instead of embedding all that information in a layout, layouts embed device IDs. The NFSv4.1 operation GETDEVICEINFO (Section 17.40) is used to retrieve the complete address information about the storage device according to its layout type. For example, the address of an NFSv4.1 data server or of an object storage device could be an IP address and port. The address of a block storage device could be a volume label.

The device ID is qualified by the layout type and unique per file system identifier (FSID, see Section 3.2.5). This allows different layout drivers to generate device IDs without the need for co-ordination.

Clients cannot expect the mapping between device ID and storage device address to persist across metadata server restart. See Section 12.7.4 for a description of how recovery works in that situation.

12.3. PNFS Operations

NFSv4.1 has several operations that are needed for pNFS servers, regardless of layout type or storage protocol. These operations are all issued to a metadata server and summarized here.

GETDEVICEINFO. As noted previously (Section 12.2.12), GETDEVICEINFO (Section 17.40) returns the mapping of device ID to storage device address.

GETDEVICELIST (Section 17.41), allows clients to fetch the all the device ID to storage device address mappings of particular file system.

LAYOUTGET (Section 17.43) is used by a client to get a layout segment for a file.

LAYOUTCOMMIT (Section 17.42) is used to inform the metadata server that the client wants to commit data it wrote to the storage device (which as indicated in the layout segment returned by LAYOUTGET).

LAYOUTRETURN (Section 17.44) is used to return a layout segment or all layouts belong to a file system to a metadata server.

The following pNFS-related operations are callback operations a
metadata server might issue to a pNFS client.

CB_LAYOUTRECALL (Section 19.3) recalls a layout segment or all layouts belonging to a file system, or all layouts belong to a client ID.

CB_RECALL_ANY (Section 19.6), tells a client that it needs to return some number of recallable objects, including layouts, to the metadata server.

CB_RECALLABLE_OBJ_AVAIL (Section 19.7) tells a client that a recallable object that it was denied (in case of pNFS, a layout, denied by LAYOUTGET) due to resource exhaustion, is now available.

12.4. PNFS Attributes

A number of attributes specific to pNFS are listed and described in Section 5.13

12.5. Layout Semantics

12.5.1. Guarantees Provided by Layouts

Layouts delegate to the client the ability to access data out of band. The layout guarantees the holder that the layout will be recalled when the state encapsulated by the layout becomes invalid (e.g., through some operation that directly or indirectly modifies the layout) or, possibly, when a conflicting layout is requested, as determined by the layout’s iomode. When a layout is recalled, and then returned by the client, the client retains the ability to access file data with normal NFSv4.1 I/O operations through the metadata server. Only the right to do I/O out-of-band is affected.

Holding a layout does not guarantee that a user of the layout has the rights to access the data represented by the layout. All user access rights MUST be obtained through the appropriate open, lock, and access operations (i.e., those that would be used in the absence of pNFS). However, if a valid layout for a file is not held by the client, the storage device should reject all I/Os to that file’s octet range that originate from that client. In summary, layouts and ordinary file access controls are independent. The act of modifying a file for which a layout is held, does not necessarily conflict with the holding of the layout that describes the file being modified. However, with certain layout types (e.g., block/volume layouts), the layout’s iomode must agree with the type of I/O being performed.

Depending upon the layout type and storage protocol in use, storage device access permissions may be granted by LAYOUTGET and may be
encoded within the type specific layout. If access permissions are
encoded within the layout, the metadata server must recall the layout
when those permissions become invalid for any reason; for example
when a file becomes unwritable or inaccessible to a client. Note,
clients are still required to perform the appropriate access
operations as described above (e.g., open and lock ops). The degree
to which it is possible for the client to circumvent these access
operations must be clearly addressed by the individual layout type
documents, as well as the consequences of doing so. In addition,
these documents must be clear about the requirements and non-
requirements for the checking performed by the server.

If the pNFS metadata server supports mandatory record locks then
record locks must behave as specified by the NFSv4.1 protocol, as
observed by users of files. If a storage device is unable to
restrict access by a pNFS client which does not hold a required
mandatory record lock then the metadata server must not grant layouts
to a client, for that storage device, that permits any access that
conflicts with a mandatory record lock held by another client. In
this scenario, it is also necessary for the metadata server to ensure
that record locks are not granted to a client if any other client
holds a conflicting layout (a layout that overlaps the range, and has
an iomode that conflicts with the lock type); in this case all
conflicting layouts must be recalled and returned before the lock
request can be granted. This requires the metadata server to
understand the capabilities of its storage devices.

12.5.2. Getting a Layout

A client obtains a layout through a new operation, LAYOUTGET. The
metadata server will give out layouts of a particular type (e.g.,
block/volume, object, or file) and aggregation as requested by the
client. The client selects an appropriate layout type which the
server supports and the client is prepared to use. The layout
returned to the client may not line up exactly with the requested
octet range. A field within the LAYOUTGET request, loga_minlength,
specifies the minimum overlap that MUST exist between the requested
layout and the layout returned by the metadata server. The
loga_minlength field should at least one. A metadata server may give
out multiple overlapping, non-conflicting layout segments to the same
client in response to a LAYOUTGET.

There is no implied ordering between getting a layout and performing
a file OPEN. For example, a layout may first be retrieved by placing
a LAYOUTGET operation in the same COMPOUND as the initial file OPEN.
Once the layout has been retrieved, it can be held across multiple
OPEN and CLOSE sequences.
The storage protocol used by the client to access the data on the storage device is determined by the layout’s type. The client needs to select a layout driver that understands how to interpret and use that layout. The method for layout driver selection used by the client is outside the scope of the pNFS extension.

Although the metadata server is in control of the layout for a file, the pNFS client can provide hints to the server when a file is opened or created about the preferred layout type and aggregation schemes. PNFS introduces a `layout_hint` (Section 5.13.4) attribute that the client can set at file creation time to provide a hint to the server for new files. Setting this attribute separately, after the file has been created could make it difficult, or impossible, for the server implementation to comply. This in turn further complicates the exclusive file creation via OPEN, which when done via the EXCLUSIVE4 createmode does not allow the setting of attributes at file creation time. However as noted in Section 17.16.4, if the server supports a persistent reply cache, the EXCLUSIVE4 createmode is not needed. Therefore, a metadata server that supports the `layout_hint` attribute MUST support a persistent session reply cache, and a pNFS client that wants to set `layout_hint` at file creation (OPEN) time MUST NOT use the EXCLUSIVE4 createmode, and instead MUST used GUARDED for an exclusive regular file creation.

12.5.3. Committing a Layout

Due to the nature of the protocol, the file attributes, and data location mapping (e.g., which offsets store data versus store holes, see Section 13.5) information that exists on the metadata server may become inconsistent in relation to the data stored on the storage devices; e.g., when WRITEs occur before a layout has been committed (e.g., between a LAYOUTGET and a LAYOUTCOMMIT). Thus, it is necessary to occasionally re-synchronize this state and make it visible to other clients through the metadata server.

The LAYOUTCOMMIT operation is responsible for committing a modified layout segment to the metadata server. Note: the data should be written and committed to the appropriate storage devices before the LAYOUTCOMMIT occurs. Note, if the data is being written asynchronously (i.e., if using NFSv4.1 as the storage protocol, the field committed in WRITE4resok is UNSTABLE4) through the metadata server, a COMMIT to the metadata server is required to synchronize the data and make it visible on the storage devices (see Section 12.5.5 for more details). The scope of this operation depends on the storage protocol in use. For block/volume-based layouts, it may require updating the block list that comprises the file and committing this layout to stable storage. Whereas, for file-layouts it requires some synchronization of attributes between
the metadata and storage devices (i.e., mainly the size attribute: EOF). It is important to note that the level of synchronization is from the point of view of the client which issued the LAYOUTCOMMIT. The updated state on the metadata server need only reflect the state as of the client’s last operation previous to the LAYOUTCOMMIT, it need not reflect a globally synchronized state (e.g., other clients may be performing, or may have performed I/O since the client’s last operation and the LAYOUTCOMMIT).

The control protocol is free to synchronize the attributes before it receives a LAYOUTCOMMIT, however upon successful completion of a LAYOUTCOMMIT, state that exists on the metadata server that describes the file MUST be in sync with the state existing on the storage devices that comprise that file as of the issuing client’s last operation. Thus, a client that queries the size of a file between a WRITE to a storage device and the LAYOUTCOMMIT may observe a size that does not reflect the actual data written.

12.5.3.1. LAYOUTCOMMIT and mtime/atime/change

The change attribute and the modify/access times may be updated, by the server, at LAYOUTCOMMIT time; since for some layout types, the change attribute and atime/mtime cannot be updated by the appropriate I/O operation performed at a storage device. The arguments to LAYOUTCOMMIT allow the client to provide suggested access and modify time values to the server. Again, depending upon the layout type, these client provided values may or may not be used. The server should sanity check the client provided values before they are used. For example, the server should ensure that time does not flow backwards. According to the NFSv4 specification, The client always has the option to set these attributes through an explicit SETATTR operation.

As mentioned, for some layout protocols the change attribute and mtime/atime may be updated at or after the time the I/O occurred (e.g., if the storage device is able to communicate these attributes to the metadata server). If, upon receiving a LAYOUTCOMMIT, the server implementation is able to determine that the file did not change since the last time the change attribute was updated (e.g., no WRITEs or over-writes occurred), the implementation need not update the change attribute; file-based protocols may have enough state to make this determination or may update the change attribute upon each file modification. This also applies for mtime and atime; if the server implementation is able to determine that the file has not been modified since the last mtime update, the server need not update mtime at LAYOUTCOMMIT time. Once LAYOUTCOMMIT completes, the new change attribute and mtime/atime should be visible if that file was modified since the latest previous LAYOUTCOMMIT or LAYOUTGET.
12.5.3.2. LAYOUTCOMMIT and size

The file’s size may be updated at LAYOUTCOMMIT time as well. The LAYOUTCOMMIT argument contains a field, loca_last_write_offset, that indicates the highest octet offset written but not yet committed via LAYOUTCOMMIT. Note: this argument is switched on a boolean value (field no_newoffset) indicating whether or not a previous write occurred. If no_newoffset is FALSE, no loca_last_write_offset is given. A loca_last_write_offset specifying an offset of 0 means octet 0 was the highest last octet written.

The metadata server may do one of the following:

1. It may update the file’s size based on the last write offset. However, to the extent possible, the metadata server should sanity check any value to which the file’s size is going to be set. E.g., it must not truncate the file based on the client presenting a smaller last write offset than the file’s current size.

2. If it has sufficient other knowledge of file size (e.g., by querying the storage devices through the control protocol), it may ignore the client provided argument and use the query-derived value.

3. It may use the last write offset as a hint, subject to correction when other information is available as above.

The method chosen to update the file’s size will depend on the storage device’s and/or the control protocol’s implementation. For example, if the storage devices are block devices with no knowledge of file size, the metadata server must rely on the client to set the size appropriately. A new size flag and length are also returned in the results of a LAYOUTCOMMIT. This union indicates whether a new size was set, and to what length it was set. If a new size is set as a result of LAYOUTCOMMIT, then the metadata server must reply with the new size. As well, if the size is updated, the metadata server in conjunction with the control protocol SHOULD ensure that the new size is reflected by the storage devices immediately upon return of the LAYOUTCOMMIT operation; e.g., a READ up to the new file size should succeed on the storage devices (assuming no intervening truncations). Again, if the client wants to explicitly zero-extend or truncate a file, SETATTR must be used; it need not be used when simply writing past EOF via WRITE.
12.5.3.3. LAYOUTCOMMIT and layoutupdate

The LAYOUTCOMMIT argument contains a loca_layoutupdate field (Section 17.42.2) of data type layoutupdate4 (Section 3.2.21). This argument is a layout type-specific structure. The structure can be used to pass arbitrary layout type-specific information from the client to the metadata server at LAYOUTCOMMIT time. For example, if using a block/volume layout, the client can indicate to the metadata server which reserved or allocated blocks the client used and did not use. The content of loca_layoutupdate (field lou_body) need not be the same the layout type-specific content returned by LAYOUTGET (Section 17.43.3) in the loc_body field of the lo_content field, of the logr_layout field. The content of loca_layoutupdate is defined by the layout type specification and is opaque to LAYOUTCOMMIT.

12.5.4. Recalling a Layout

Since a layout protects a client’s access to a file via a direct client-storage-device path, a layout need only be recalled when it is semantically unable to serve this function. Typically, this occurs when the layout no longer encapsulates the true location of the file over the octet range it represents. Any operation or action (e.g., server driven restriping or load balancing) that changes the layout will result in a recall of the layout. A layout is recalled by the CB_LAYOUTRECALL callback operation (see Section 19.3). This callback can either recall a layout segment identified by a octet range, all the layouts associated with a file system (FSID), or all layouts. Recalling all layouts or all the layouts associated with a file system also invalidates the client’s device cache for the affected file systems. Multiple layout segments may be returned in a single compound operation. Section 12.5.4.2 discusses sequencing issues surrounding the getting, returning, and recalling of layouts.

The iomode is also specified when recalling a layout or layout segment. Generally, the iomode in the recall request must match the layout, or segment, being returned; e.g., a recall with an iomode of LAYOUTIOMODE4_RW should cause the client to only return LAYOUTIOMODE4_RW layout segments (not LAYOUTIOMODE4_REALAYOUTIOMODE4_READ segments). However, a special LAYOUTIOMODE4_ANY enumeration is defined to enable recalling a layout of any type (i.e., the client must return both read-only and read/write layouts).

A REMOVE operation may cause the metadata server to recall the layout to prevent the client from accessing a non-existent file and to reclaim state stored on the client. Since a REMOVE may be delayed until the last close of the file has occurred, the recall may also be delayed until this time. As well, once the file has been removed,
after the last reference, the client SHOULD no longer be able to perform I/O using the layout (e.g., with file-based layouts an error such as ESTALE could be returned).

Although pNFS does not alter the caching capabilities of clients, or their semantics, it recognizes that some clients may perform more aggressive write-behind caching to optimize the benefits provided by pNFS. However, write-behind caching may impact the latency in returning a layout in response to a CB_LAYOUTRECALL; just as caching impacts DELEGRETURN with regards to data delegations. Client implementations should limit the amount of unwritten data they have outstanding at any one time. Server implementations may fence clients from performing direct I/O to the storage devices if they perceive that the client is taking too long to return a layout once recalled. A server may be able to monitor client progress by watching client I/Os or by observing LAYOUTRETURNs of sub-portions of the recalled layout. The server can also limit the amount of dirty data to be flushed to storage devices by limiting the octet ranges covered in the layouts it gives out.

Once a layout has been returned, the client MUST NOT issue I/Os to the storage devices for the file, octet range, and iomode represented by the returned layout. If a client does issue an I/O to a storage device for which it does not hold a layout, the storage device SHOULD reject the I/O.

12.5.4.1. Recall Callback Robustness

It has been assumed thus far that pNFS client state for a file exactly matches the pNFS server state for that file and client regarding layout ranges and permissions. This assumption leads to the implication that any callback results in a LAYOUTRETURN or set of LAYOUTRETURNs that exactly match the range in the callback, since both client and server agree about the state being maintained. However, it can be useful if this assumption does not always hold. For example:

- It may be useful for clients to be able to discard layout information without calling LAYOUTRETURN. If conflicts that require callbacks are very rare, and a server can use a multi-file callback to recover per-client resources (e.g., via a FSID recall, or a multi-file recall within a single compound), the result may be significantly less client-server pNFS traffic.

- It may be similarly useful for servers to maintain information about what ranges are held by a client on a coarse-grained basis, leading to the server’s layout ranges being beyond those actually held by the client. In the extreme, a server could manage
conflicts on a per-file basis, only issuing whole-file callbacks even though clients may request and be granted sub-file ranges.

- In order to avoid errors, it is vital that a client not assign itself layout permissions beyond what the server has granted and that the server not forget layout permissions that have been granted. On the other hand, if a server believes that a client holds a layout segment that the client does not know about, it’s useful for the client to cleanly indicate completion of the requested recall either by issuing a LAYOUTRETURN for the entire requested range or by returning an NFS4ERR_NOMATCHING_LAYOUT error to the layout recall callback.

Thus, in light of the above, it is useful for a server to be able to issue callbacks for layout ranges it has not granted to a client, and for a client to return ranges it does not hold. A pNFS client must always return layout segments that comprise the full range specified by the recall. Note, the full recalled layout range need not be returned as part of a single operation, but may be returned in segments. This allows the client to stage the flushing of dirty data, layout commits, and returns. Also, it indicates to the metadata server that the client is making progress.

It is possible that write requests may be presented to a storage device no longer allowed to perform them. This behavior is limited by requiring that a client MUST wait for completion of all writes covered by a layout range before returning a layout that covers that range. Since the server has no control as to when the client will return the layout, the server may later decide to unilaterally revoke the client’s access provided by the layout in question. Upon doing so the server must deal with the possibility of lingering writes, outstanding writes still in flight to data servers identified by the revoked layout. Each layout-specification MUST define whether unilateral layout revocation by the metadata server is supported, and if so, the specification must also outline how lingering writes are to be dealt with; e.g., storage devices identified by the revoked layout in question could be fenced off from the appropriate client. If unilateral revocation is not supported, there MUST be no possibility that the client has outstanding write requests when a layout is returned.

In order to ensure client/server convergence on the layout state, the final LAYOUTRETURN operation in a sequence of LAYOUTRETURN operations for a particular recall, MUST specify the entire range being recalled, echoing the recalled layout type, iomode, recall/return type (FILE, FSID, or ALL), and octet range; even if layout segments pertaining to partial ranges were previously returned. In addition, if the client holds no layout segment that overlaps the range being
recalled, the client should return the NFS4ERR_NOMATCHING_LAYOUT error code. This allows the server to update its view of the client’s layout state.

12.5.4.2. Serialization of Layout Operations

As with other stateful operations, pNFS requires the correct sequencing of layout operations. PNFS uses the sessions feature of NFSv4.1 to provide the correct sequencing between regular operations and callbacks. It is the server’s responsibility to avoid inconsistencies regarding the layouts it hands out and the client’s responsibility to properly serialize its layout requests and layout returns.

12.5.4.2.1. Get/Return Serialization

The protocol allows the client to send concurrent LAYOUTGET and LAYOUTRETURN operations to the server. However, the protocol does not provide any means for the server to process the requests in the same order in which they were created, nor does it provide a way for the client to determine the order in which parallel outstanding operations were processed by the server. Thus, when a layout segment retrieved by an outstanding LAYOUTGET operation intersects with a layout segment returned by an outstanding LAYOUTRETURN the order in which the two conflicting operations are processed determines the final state of the overlapping segment. To disambiguate between the two cases the client MUST serialize LAYOUTGET operations and voluntary LAYOUTRETURN operations for the same file.

It is permissible for the client to send in parallel multiple LAYOUTGET operations for the same file or multiple LAYOUTRETURN operations for the same file; but never a mix of both. It is also permissible for the client to combine LAYOUTRETURN and LAYOUTGET operations for the same file in the same COMPOUND request as the server MUST process these in order. If a client does issue such requests, it MUST NOT have more than one outstanding for the same file at the same time and MUST NOT have other LAYOUTGET or LAYOUTRETURN operations outstanding at the same time for that same file.

12.5.4.2.2. Recall/Return Sequencing

One critical issue with operation sequencing concerns callbacks. The protocol must defend against races between the reply to a LAYOUTGET operation and a subsequent CB_LAYOUTRECALL. A client MUST NOT process a CB_LAYOUTRECALL that identifies an outstanding LAYOUTGET operation to which the client has not yet received a reply. Conflicting LAYOUTGET operations are identified in the CB_SEQUENCE
preceding the CB_LAYOUTRECALL.

The callback races section (Section 2.10.4.3) describes the session mechanism for allowing the client to detect such situations in order to not process such a CB_LAYOUTRECALL. The server MUST reference all conflicting LAYOUTGET operations in the CB_SEQUENCE that precedes the CB_LAYOUTRECALL. A zero length array of referenced operations is used by the server to tell the client that the server does not know of any LAYOUTGET operations that conflict with the recall.

12.5.4.2.2.1. Client Side Considerations

Consider a pNFS client that has issued a LAYOUTGET and then receives an overlapping recall callback for the same file. There are two possibilities, which the client would be unable to distinguish without additional information provided by the sessions implementation.

1. The server processed the LAYOUTGET before issuing the recall, so the LAYOUTGET response is in flight, and must be waited for because it may be carrying layout info that will need to be returned to deal with the recall callback.

2. The server issued the callback before receiving the LAYOUTGET. The server will not respond to the LAYOUTGET until the recall callback is processed.

These possibilities could cause deadlock, as the client must wait for the LAYOUTGET response before processing the recall in the first case, but that response will not arrive until after the recall is processed in the second case. Via the CB_SEQUENCE operation, the server provides the client with the \( \{ \text{slotid}, \text{sequenceid} \} \) of any earlier LAYOUTGET operations which remain unconfirmed at the server by the session slot usage rules. This allows the client to disambiguate between the two cases, in case 1, the server will provide the operation reference(s), whereas in case 2 it will not (because there are no dependent client operations). Therefore, the action at the client will only require waiting in the case that the client has not yet seen the server’s earlier responses to the LAYOUTGET operation(s).

The following requirements apply to avoid this deadlock: by adhering to the following requirements:

- A LAYOUTGET MUST be rejected with the error NFS4ERR_RECALLCONFLICT if there’s an overlapping outstanding recall callback to the same client.
When processing a recall, the client MUST wait for a response to all conflicting outstanding LAYOUTGETs that are referenced in the CB_SEQUENCE for the recall before performing any RETURN that could be affected by any such response.

The client SHOULD wait for responses to all operations required to complete a recall before sending any LAYOUTGETs that would conflict with the recall because the server is likely to return errors for them.

Before sending a new LAYOUTGET for a range covered by a layout recall, the client SHOULD wait for responses to any outstanding LAYOUTGET that overlaps any portion of the new LAYOUTGET’s range. This is because it is possible (although unlikely) that the prior operation may have arrived at the server after the recall completed and hence will succeed.

The recall process can be considered as done by the client when the final LAYOUTRETURN operation for the recalled range is issued.

12.5.4.2.2.2. Server Side Considerations

Consider a related situation from the metadata server’s point of view. The metadata server has issued a recall layout callback and receives an overlapping LAYOUTGET for the same file before the LAYOUTRETURN(s) that respond to the recall callback. Again, there are two cases:

1. The client issued the LAYOUTGET before processing the recall callback.

2. The client issued the LAYOUTGET after processing the recall callback, but it arrived before the LAYOUTRETURN that completed that processing.

The metadata server MUST reject the overlapping LAYOUTGET. The client has two ways to avoid this result – it can issue the LAYOUTGET as a subsequent element of a COMPOUND containing the LAYOUTRETURN that completes the recall callback, or it can wait for the response to that LAYOUTRETURN.

There is little the session sequence logic can do to disambiguate between these two cases, because both operations are independent of one another. They are simply asynchronous events which crossed. The situation can even occur if the session is configured to use a single connection for both operations and callbacks.
12.5.5. Metadata Server Write Propagation

Asynchronous writes written through the metadata server may be propagated lazily to the storage devices. For data written asynchronously through the metadata server, a client performing a read at the appropriate storage device is not guaranteed to see the newly written data until a COMMIT occurs at the metadata server. While the write is pending, reads to the storage device can give out either the old data, the new data, or a mixture thereof. After either a synchronous write completes, or a COMMIT is received (for asynchronously written data), the metadata server must ensure that storage devices give out the new data and that the data has been written to stable storage. If the server implements its storage in any way such that it cannot obey these constraints, then it must recall the layouts to prevent reads being done that cannot be handled correctly.

12.6. PNFS Mechanics

This section describes the operations flow taken by a pNFS client to a metadata server and storage device.

When a pNFS client encounters a new FSID, it issues a GETATTR to the NFSv4.1 server for the fs_layout_type (Section 5.13.1) attribute. If the attribute returns at least one layout type, and the layout type(s) returned is(are) among the set supported by the client, the client knows that pNFS is a possibility for the filesystem. If, from the server that returned the new FSID, the client does not have a client ID that came from an EXCHANGE_ID result that returned EXCHGID4_FLAG_USE_PNFS_MDS, it must send an EXCHANGE_ID to the server with the EXCHGID4_FLAG_USE_PNFS_MDS bit set. If the server’s response does not have EXCHGID4_FLAG_USE_PNFS_MDS, then contrary to what the fs_layout_type attribute said, the server does not support pNFS, and the client will not be able use pNFS to that server.

Once the client has a client ID that supports pNFS, it creates a persistent session over the client ID, requesting persistent.

If the client wants to create a file on the file system identified by the FSID that supports pNFS, it issues an OPEN with a create type of GUARDED4 (if it wants an exclusive create), or UNCHECKED4 (if it does not want an exclusive create). Among the various attributes it sets in createattrs, it includes layout_hint and fills it with information pertinent to the layout type it wants to use. The COMPOUND procedure that the OPEN is sent with should include a GETATTR operation (on the filehandle OPEN sets) that retrieves the layout_type attribute. This is so the client can determine what layout type the server will in fact support, and thus what storage protocol the client must use.
If the client wants to open an existing file, then it also includes a GETATTR to determine what layout type the file supports.

The GETATTR in either the file creation or plain file open case can also include the layout_blksize and layout_alignment attributes so that the client can determine optimal offsets and lengths for I/O on the file.

Assuming the client supports the layout type returned by GETATTR, it then issues LAYOUTGET using the filehandle returned by OPEN, specifying the range it wants to do I/O on. The response is a layout segment, which may be a subset of the range the client asked for. It also includes device IDs and a description of how data is organized (or in the case of writing, how data is to be organized) across the devices. The device IDs and data description are encoded in a format that is specific to the layout type, but the client is expected to understand.

When the client wants to issue an I/O, it determines which device ID it needs to send the I/O command to by examining the data description in the layout. It then issues a GETDEVICEINFO to find the device address of the device ID. The client then sends the I/O command to device address, using the storage protocol defined for the layout type.

If the I/O was an input request, then at some point the client may want to commit the access time to the metadata server. It uses the LAYOUTCOMMIT operation. If the I/O was an output request, then at some point the client may want to commit the modification time and the new size of the file if it believes it lengthed the file, to the metadata server and the modified data to the filesystem. Again, it uses LAYOUTCOMMIT.

12.7. Recovery

Recovery is complicated due to the distributed nature of the pNFS protocol. In general, crash recovery for layouts is similar to crash recovery for delegations in the base NFSv4 protocol. However, the client’s ability to perform I/O without contacting the metadata server and the fact that unlike delegations, layouts are not bound to stateids introduces subtleties that must be handled correctly if file system corruption is to be avoided.

12.7.1. Client Recovery

Client recovery for layouts is similar to client recovery for other lock/delegation state. When an pNFS client reboots, it will lose all information about the layouts that it previously owned. There are
two methods by which the server can reclaim these resources and allow otherwise conflicting layouts to be provided to other clients.

The first is through the expiry of the client’s lease. If the client recovery time is longer than the lease period, the client’s lease will expire and the server will know that state may be released. For layouts the server may release the state immediately upon lease expiry or it may allow the layout to persist awaiting possible lease revival, as long as there are no conflicting requests.

On the other hand, the client may restart in less time than it takes for the lease period to expire. In such a case, the client will contact the server through the standard EXCHANGE_ID protocol. The server will find that the client’s co_ownerid matches the co_ownerid of the previous client invocation, but that the verifier is different. The server uses this as a signal to release all layout state associated with the client’s previous invocation. It is possible that all data written by the client to storage devices but not completed via LAYOUTCOMMIT is lost.

12.7.2. Dealing with Lease Expiration on the Client

The mappings between device IDs and device addresses are what allow a pNFS client to safely write data to and read data from a storage device. These mappings are leased (just like with locking state) from the metadata server, and as long as the lease is valid, the client has a right to issue I/O to the storage devices. The lease on device ID to device address mappings is renewed when the metadata server receives a SEQUENCE operation from the pNFS client. The same is not specified to be true for the data server receiving a SEQUENCE operation, and the client MUST NOT assume that a SEQUENCE sent to a data server will renew its lease.

The loss of the lease leads to the loss of the device ID to device address mappings. If a mapping is used for I/O after lease expiration, the consequences could be data corruption. To avoid losing its lease, the client should start its lease timer based on the time that it issued the operation to the metadata server rather than based on the time the response was received. It is also necessary to take propagation delay into account as described in Section 8.12. Thus, the client must be aware of the one-way propagation delay and should issue renewals well in advance of lease expiration.

If a client believes its lease has expired, it MUST NOT issue I/O to the storage device until it has validated its lease. The client can issue a SEQUENCE operation to the metadata server. If the SEQUENCE operation is successful, but sr_status_flag has
SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED, 
SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED, or 
SEQ4_STATUS_ADMIN_STATE_REVOKED set, the client must recover by 
deleting all its records of layouts and device ID to device address 
mappings, then writing any modified but uncommitted data in its 
memory directly to the metadata server with the stable argument to 
WRITE set to FILE_SYNC4, and finally reacquiring any layouts it needs 
via LAYOUTGET.

If sr_status_flags from the metadata server has 
SEQ4_STATUS_RESTART_RECLAIM_NEEDED set (or SEQUENCE returns 
NFS4ERR_STATE_CLIENTID, or SEQUENCE returns NFS4ERR_BAD_SESSION and 
CREATE_SESSION returns NFS4ERR_STATE_CLIENTID) then the metadata 
server has restarted, and the client must recovery using the methods 
described in Section 12.7.4.

If sr_status_flags from the metadata server has 
SEQ4_STATUS_LEASE_MOVED set, then the client recovers by following 
the procedure described in Section 10.6.7.1. After that, the client 
may get an indication that the layout state was not moved with the 
filesystem. The client is then required the client to recover per 
other applicable situations discussed in Paragraph 3 or Paragraph 4 
of this section.

If sr_status_flags reports no loss of state, then the lease the 
client has with the metadata server is valid and renewed, and the 
client can re-commence I/O to the storage devices.

While clients should not issue I/Os to storage devices that may 
extend past the lease expiration time period, this is not always 
possible (e.g., an extended network partition that starts after the 
I/O is send and does not heal till the I/O request is received by the 
data server). Thus the metadata server and/or storage device are 
responsible for protecting the pNFS server from I/Os that are sent 
before the lease expires, but arrive after the lease expires. See 
Section 12.7.3.

12.7.3. Dealing with Loss of Layout State on the Metadata Server

This section describes recovery from the situation where all of 
the following are true: the metadata server has not restarted; a pNFS 
client’s device ID to device address mappings and/or layouts have 
been discarded (usually because the client’s lease expired) and are 
invalid; and an I/O from the pNFS client arrives at the storage 
device. The metadata server and its storage devices may solve this 
by fencing the client (i.e., prevent the execution of I/O operations 
from the client to the storage devices after layout state loss). The 
details of how fencing is done are specific to the layout type. The
solution for NFSv4.1 file-based layouts is described in this document (Section 13.13), and for other layout types in their respective external specification documents.

12.7.4. Recovery from Metadata Server Restart

The pNFS client will discover that the metadata server has restarted (e.g. rebooted) via the methods described in Section 8.6.2 and discussed in a pNFS-specific context in Paragraph 4, of Section 12.7.2. The client MUST stop using and delete device ID to device address mappings it previously received from the metadata server. Having done that, if the client wrote data to the storage device without committing the layout segment(s) via LAYOUTCOMMIT, then client has additional work to do in order to get the client, metadata server and storage device(s) all synchronized on the state of the data.

- If the client has data still modified and unwritten in the client’s memory, the client has only two choices.

  1. The client can obtain a layout segment via LAYOUTGET after the server’s grace period and write the data to the storage devices.

  2. The client can write that data through the metadata server using the WRITE (Section 17.32) operation, and then obtain layout segments as needed.

As noted in Paragraph 2 of Section 8.6.2.1, and in Section 17.43.4, LAYOUTGET and WRITE may not be allowed until the grace period expires. Under some conditions, as described in Section 12.7.5, LAYOUTGET and/or WRITE maybe permitted during the metadata server’s grace period.

- If the client synchronously wrote data to the storage device, but still a copy of that data in its memory, then it has available to it the recovery options listed above in the previous bullet point. If the metadata server is also in its grace period, the client has available to it the options below in the next bullet item.

- The client does not have a copy of the data in its memory and the metadata server is still in its grace period. The client cannot use LAYOUTGET (within or outside the grace period) to reclaim a layout segment because the contents of the response from LAYOUTGET
may not match what it had previously. The range might be
different or it might get the same range but the content of the
layout might be different. Even if the content of the layout
appears to be the same, the device IDs may map to different
device addresses, and even if the device addresses are the same,
the device addresses could have been assigned to a different
storage device. The option of retrieving the data from the
storage device and writing it to the metadata server per the
recovery scenario described above in the previous two bullets is
not available because, again, the mappings of range to device ID,
device ID to device address, device address to physical device are
stale and new mappings via new LAYOUTGET do not solve the problem.

The only recovery option for this scenario is to issue a
LAYOUTCOMMIT in reclaim mode, which the metadata server will
accept as long as it is in its grace period. The use of
LAYOUTCOMMIT in reclaim mode informs the metadata server that the
layout segment has changed. It is critical the metadata server
receive this information before its grace period ends, and thus
before it starts allowing updates to the filesystem.

To issue LAYOUTCOMMIT in reclaim mode, the client sets the
loca_reclaim field of the operation’s arguments (Section 17.42.2)
to TRUE. During the metadata server’s recovery grace period (and
only during the recovery grace period) the metadata server is
prepared to accept LAYOUTCOMMIT requests with the loca_reclaim
field set to TRUE.

When loca_reclaim is TRUE, the client is attempting to commit
changes to the layout segment that occurred prior to the restart
of the metadata server. The metadata server applies some
consistency checks on the loca_layoutupdate field of the arguments
to determine whether the client can commit the data written to the
data server to the filesystem. The loca_layoutupdate field is of
data type layoutupdate4, and contains layout type-specific content
(in the lou_body field of loca_layoutupdate). The layout type-
specific information that loca_layoutupdate might have is
discussed in Section 12.5.3.3. If the metadata server’s
consistency checks on loca_layoutupdate succeed, then the metadata
server MUST commit the data (as described by the loca_offset,
loca_length, and loca_layoutupdate fields of the arguments) that
was written to storage device. If the metadata server’s
consistency checks on loca_layoutupdate fail, the metadata server
rejects the LAYOUTCOMMIT operation, and makes no changes to the
file system. However, any time LAYOUTCOMMIT with loca_reclaim
TRUE fails, the pNFS client has lost all the data in the range
defined by <loca_offset, loca_length>. A client can defend
against this risk by caching all data, whether written
synchronously or asynchronously in its memory and not release the cached data until a successful LAYOUTCOMMIT.

- The client does not have a copy of the data in its memory and the metadata server is no longer in its grace period; i.e. the metadata server returns NFS4ERR_NO_GRACE. As with the scenario in the above bullet item, the failure of LAYOUTCOMMIT means the data in the range \(<loca\_offset, loca\_length>\) lost. The defense against the risk is the same; cache all written data on the client until a successful LAYOUTCOMMIT.

### 12.7.5. Operations During Metadata Server Grace Period

Some of the recovery scenarios thus far noted that some operations, namely WRITE and LAYOUTGET might be permitted during the metadata server's grace period. The metadata server may allow these operations during its grace period, if it can reliably determine that servicing such a request will not conflict with an impending LAYOUTCOMMIT (or, in the case of WRITE, conflicting with an impending OPEN, or a LOCK on a file with mandatory record locking enabled) reclaim request. As mentioned previously, some operations, namely WRITE and LAYOUTGET are likely to be rejected during the metadata server’s grace period, because to provide simple, valid handling during the grace period, the easiest method is to simply reject all non-reclaim pNFS requests and WRITE operations by returning the NFS4ERR_GRACE error. However, depending on the storage protocol (which is specific to the layout type) and metadata server implementation, the metadata server may be able to determine that a particular request is safe. For example, a metadata server may save provisional allocation mappings for each file to stable storage, as well as information about potentially conflicting OPEN share modes and mandatory record locks that might have been in effect at the time of restart, and use this information during the recovery grace period to determine that a WRITE request is safe.

### 12.7.6. Storage Device Recovery

Recovery from storage device restart is mostly dependent upon the layout type in use. However, there are a few general techniques a client can use if it discovers a storage device has crashed while holding modified, uncommitted data that was asynchronously written. First and foremost, it is important to realize that the client is the only one who has the information necessary to recover non-committed data; since, it holds the modified data and most probably nobody else does. Second, the best solution is for the client to err on the side of caution and attempt to re-write the modified data through another path.
The client should immediately write the data to the metadata server, with the stable field in the WRITE4args set to FILE_SYNC4. Once it does this, there is no need to wait for the original storage device.

12.8. Metadata and Storage Device Roles

If the same physical hardware is used to implement both a metadata server and storage device, then the same hardware entity is to be understood to be implementing two distinct roles and it is important that it be clearly understood on behalf of which role the hardware is executing at any given time.

Various sub-cases can be distinguished.

1. The storage device uses NFSv4.1 as the storage protocol. The same physical hardware is used to implement both a metadata and data server. If an EXCHANGE_ID operation issued to the metadata server has EXCHGID4_FLAG_USE_PNFS_MDS set and not EXCHGID4_FLAG_USE_PNFS_DS not set, the role of all sessions derived from the client ID is metadata server-only. If an EXCHANGE_ID operation issued to the data server has EXCHGID4_FLAG_USE_PNFS_DS set and EXCHGID4_FLAG_USE_PNFS_MDS not set, the role of all sessions derived from the client ID is data server only. These assertions are true regardless whether the network addresses of the metadata server and data server are the same or not.

The client will use the same client owner for both the metadata server EXCHANGE_ID and the data server EXCHANGE_ID. Since the client issues one with EXCHGID4_FLAG_USE_PNFS_MDS set, and the other with EXCHGID4_FLAG_USE_PNFS_DS set, the server will need to return unique client IDs, as well as server_owners, which will eliminate ambiguity about dual roles the same physical entity serves.

2. The metadata and data server each return EXCHANGE_ID results with EXCHGID4_FLAG_USE_PNFS_DS and EXCHGID4_FLAG_USE_PNFS_MDS both set, the server_owner and server_scope results are the same, and the client IDs are the same, and if RPCSEC_GSS is used, the server principals are the same. As noted in Section 2.10.3.4.1 the two servers are the same, whether they have the same network address or not. If the pNFS server is ambiguous in its EXCHANGE_ID results as to what role a client ID may be used for, yet still requires the NFSv4.1 request be directed in a manner specific to a role (e.g. a READ request for a particular offset directed to the metadata server role might use a different offset if the READ was intended for the data server role, if the file is using STRIPE4_DENSE packing, see Section 13.5), the pNFS server
may mark the metadata filehandle differently from the data filehandle so that operations addressed to the metadata server can be distinguished from those directed to the data servers. Marking the metadata and data server filehandles differently (and this is RECOMMENDED) is possible because the former are derived from OPEN operations, and the latter are derived from LAYOUTGET operations.

Note, that it may be the case that while the metadata server and the storage device are distinct from one client’s point of view, the roles may be reversed according to another client’s point of view. For example, in the cluster file system model a metadata server to one client, may be a data server to another client. If NFSv4.1 is being used as the storage protocol, then pNFS servers need to mark filehandles according to their specific roles.

If a current filehandle is set that is inconsistent with the role to which it is directed, then the error NFS4ERR_BADHANDLE should result. For example, if a request is directed at the data server, because the first current handle is from a layout, any attempt to set the current filehandle to be a value not from a layout should be rejected. Similarly, if the first current file handle was for a value not from a layout, a subsequent attempt to set the current filehandle to a value obtained from a layout should be rejected.

3. The storage device does not use NFSv4.1 as the storage protocol, and the same physical hardware is used to implement both a metadata and storage device. Whether distinct network addresses are used to access metadata server and storage device is immaterial, because, it is always clear to the pNFS client and server, from upper layer protocol being used (NFSv4.1 or non-NFSv4.1) what role the request to the common server network address is directed to.

12.9. Security Considerations

PNFS has a metadata path and a data path (i.e., storage protocol). The metadata path includes the pNFS-specific operations (listed in Section 12.3); all existing NFSv4.1 conventional (non-pNFS) security mechanisms and features apply to the metadata path. The combination of components in a pNFS system (see Figure 62) is required to preserve the security properties of NFSv4.1 with respect to an entity accessing storage device from a client, including security countermeasures to defend against threats that NFSv4.1 provides defenses for in environments where these threats are considered significant.
In some cases, the security countermeasures for connections to storage devices may take the form of physical isolation or a recommendation not to use pNFS in an environment. For example, it may be impractical to provide confidentiality protection for some storage protocols to protect against eavesdropping; in environments where eavesdropping on such protocols is of sufficient concern to require countermeasures, physical isolation of the communication channel (e.g., via direct connection from client(s) to storage device(s)) and/or a decision to forego use of pNFS (e.g., and fall back to conventional NFSv4.1) may be appropriate courses of action.

Where communication with storage devices is subject to the same threats as client to metadata server communication, the protocols used for that communication need to provide security mechanisms comparable to those available via RPSEC_GSS for NFSv4.1. Many situations in which pNFS is likely to be used will not be subject to the overall threat profile for which NFSv4.1 is required to provide countermeasures.

PNFS implementations MUST NOT remove NFSv4’s access controls. The combination of clients, storage devices, and the metadata server are responsible for ensuring that all client to storage device file data access respects NFSv4.1’s ACLs and file open modes. This entails performing both of these checks on every access in the client, the storage device, or both (as applicable; when the storage device is an NFSv4.1 server, the storage device is ultimately responsible for controlling access). If a pNFS configuration performs these checks only in the client, the risk of a misbehaving client obtaining unauthorized access is an important consideration in determining when it is appropriate to use such a pNFS configuration. Such configurations SHOULD NOT be used when client-only access checks do not provide sufficient assurance that NFSv4.1 access control is being applied correctly.

13. PNFS: NFSv4.1 File Layout Type

This section describes the semantics and format of NFSv4.1 file-based layouts for pNFS. NFSv4.1 file-based layouts uses the LAYOUT4_NFSV4_1_FILES layout type. The LAYOUT4_NFSV4_1_FILES type defines striping data across multiple NFSv4.1 data servers.

13.1. Session Considerations

Sessions are a mandatory feature of NFSv4.1, and this extends to both the metadata server and file-based (NFSv4.1-based) data servers. If data is served by both the metadata server and an NFSv4.1-based data server, the metadata and data server MUST have separate client IDs.
(unless the EXCHANGE_ID results indicate the server will allow the client ID to support both metadata and data pNFS operations).

When a creating a client ID to access a pNFS metadata server, the pNFS metadata client sends an EXCHANGE_ID operation that has EXCHGID4_FLAG_USE_PNFS_MDS set (EXCHGID4_FLAG_USE_NON_PNFS and EXCHGID4_FLAG_USE_PNFS_DS MAY be set as well). If the server’s EXCHANGE_ID results have EXCHGID4_FLAG_USE_PNFS_MDS set, then the client may use the client ID to create sessions that will exchange pNFS metadata operations.

If pNFS metadata client gets a layout that refers it to an NFSv4.1 data server, it needs a client ID on that data server. If it does not yet have a client ID from the server that had the EXCHGID4_FLAG_USE_PNFS_DS flag set in the EXCHANGE_ID results, then the client must send an EXCHANGE_ID to the data server, using the same co_ownerid as it sent to the metadata server, with the EXCHGID4_FLAG_USE_PNFS_DS flag set in arguments. If the server’s EXCHANGE_ID results have EXCHGID4_FLAG_USE_PNFS_DS set, then the client may use the client ID to create sessions that will exchange pNFS data operations.

The client ID returned by a metadata server has no required association to the client ID returned by a data server that the metadata server’s layouts referred the client to, although a server implementation is free construct such an association (e.g. via a private data server/metadata server protocol and client ID table). Similarly the EXCHANGE_ID/CREATE_SESSION sequenceid state used by the pNFS metadata client and server has no association with the EXCHANGE_ID/CREATE_SESSION sequenceid state used by the data client/server (and the pNFS server and the pNFS client MUST NOT make this association). By decoupling the client IDs of metadata and data servers from each other, implementation of the session on pNFS servers is potentially simpler.

In a non-pNFS server or in a metadata server, the sessionid in the SEQUENCE operation implies the client ID, which in turn might be used by the server to map the stateid to the right client/server pair. However, when a data server is presented with a READ or WRITE operation with a stateid, because the stateid is associated with client ID on a metadata server, and because the sessionid in the preceding SEQUENCE operation is tied to the potentially unrelated data server client ID, the data server has no obvious way to determine the metadata server from the COMPOUND procedure, and thus has no way to validate the stateid. One recommended approach is for pNFS servers to encode metadata server routing and/or identity information in the data server filehandles as returned in the layout. If the metadata server identity or location changes, requiring the
data server filehandles to become invalid (stale), the metadata server must first recall the layouts.

Invalidating data server filehandles does not render the pNFS data cache invalid. If the metadata server file handle of a file is persistent, the client can map the metadata server filehandle to cached data, and when granted data server filehandles, map the data server filehandles to their metadata server filehandle.

13.2. File Layout Definitions

The following definitions apply to the LAYOUT4_NFSV4_1_FILES layout type, and may be applicable to other layout types.

Unit. A unit is a set of data written to a data server.

Pattern. A pattern is a method of distributing fix sized units across a set of data servers. A pattern is iterated one or more times. A pattern has one or more units. Each unit in each iteration of a pattern MUST be the same size.

Stripe. A stripe is a set of data distributed across a set of data servers in a pattern before that pattern repeats.

Stripe Width. A stripe width is the size of stripe in octets.

Hereafter, this document will refer to a unit that is written in a pattern as a "stripe unit".

A pattern may have more stripe units than data servers. If so, some data servers will have more than one stripe unit per stripe. A data server that has multiple stripe units per stripe MAY store each unit in a different data file.

13.3. File Layout Data Types

The high level NFSv4.1 layout types are nfsv4_1_file_layout_ds_addr4, nfsv4_1_file_layouthint4, and nfsv4_1_file_layout4.

When LAYOUTGET returns a LAYOUT4_NFSV4_1_FILES layout (indicated in the loc_type field of the lo_content field), the loc_body field of the lo_content field contains a value of data type nfsv4_1_file_layout4. Among other content, nfsv4_1_file_layout4 has storage device IDs (within the nfl_ds_fh_list array) of data type deviceid4.

The GETDEVICEINFO operation maps a device ID to a storage device address (type device_addr4). When GETDEVICEINFO returns a device
address with a layout type of LAYOUT4_NFSV4_1_FILES (the 
da_layout_type field), the da_addr_body field contains a value of 
data type nfsv4_1_file_layout_ds_addr4.

The SETATTR operation supports a layout hint attribute 
(Section 5.13.4). When the client sets a layout hint (data type 
layouthint4) with a layout type of LAYOUT4_NFSV4_1_FILES (the 
loh_type field), the loh_body field contains a value of data type 
nfsv4_1_file_layouthint4.

The top level and lower level NFSv4.1 layout data types have the 
following XDR descriptions.
enum file_layout_ds_type4 {
    FILEDS4_SIMPLE = 1,
    FILEDS4_COMPLEX = 2
};

%/* Encoded in the da_addr_body field of type device_addr4: */
union nfsv4_1_file_layout_ds_addr4
    switch (file_layout_ds_type4 nflda_type) {
    case FILEDS4_SIMPLE:
       /netaddr4 nflda_simp_ds_list<>;
    case FILEDS4_COMPLEX:
       /deviceid4 nflda_comp_ds_list<>;
    default:
        void;
    }

enum stripetype4 {
    STRIPE4_SPARSE = 1,
    STRIPE4_DENSE = 2
};

%/* Encoded in the loh_body field of type layouthint4: */
struct nfsv4_1_file_layouthint4 {
    stripetype4 nflh_stripe_type;
    length4 nflh_stripe_unit_size;
    uint32_t nflh_stripe_width;
};

struct nfsv4_1_file_layout_ds_fh4 {
    deviceid4 nfldf_ds_id;
    uint32_t nfldf_ds_index;
    nfs_fh4 nfldf_fh;
};

%/* Encoded in the loc_body field of type layout_content4: */
struct nfsv4_1_file_layout4 {
    stripetype4 nfl_stripe_type;
    bool nfl_commit_through_mds;
    length4 nfl_stripe_unit_size;
    length4 nfl_file_size;
    uint32_t nfl_stripe_indices<>;
    nfsv4_1_file_layout_ds_fh4 nfl_ds_fh_list<>;
};

%/*
% * Encoded in the lou_body field of type layoutupdate4:
% * Nothing. lou_body is a zero length array of octets.
% */
The nfsv4_1_file_layout_ds_addr4 data server address is composed of a FILEDS4_SIMPLE or a FILEDS4_COMPLEX data server address. A FILEDS4_SIMPLE data server address is composed of an array of network addresses (data type netaddr4). All data servers in a FILEDS4_SIMPLE list (field nflda_simp_ds_list) must be equivalent and are used for data server multipathing; see Section 13.6 for more details on equivalent data servers. FILEDS4_SIMPLE data servers always refer to actual data servers. On the other hand, a FILEDS4_COMPLEX data server address is constructed of list of device IDs (field nflda_comp_ds_list). Each device ID in nflda_comp_ds_list corresponds to the device ID of a data server address of type FILEDS4_SIMPLE. A FILEDS4_COMPLEX data server list MUST NOT contain device IDs of other FILEDS4_COMPLEX data servers; only device IDs of FILEDS4_SIMPLE data servers are to be referenced. This enables multiple equivalent data servers to be identified through a single device ID and provides a space efficient mechanism by which to identify multiple data servers within a layout. FILEDS4_COMPLEX and FILEDS4_SIMPLE data servers share the same device ID space and should be cached similarly by the client.

The nfsv4_1_file_layout4 data type specifies an ordered array of <device ID, filehandle> tuples, as well as the stripe unit size, type of stripe layout (discussed later in this section and in Section 13.4), and the file’s current size as of LAYOUTGET (Section 17.43) time.

The nfl_ds_fh_list array within the nfsv4_1_file_layout4 data type contains a list of nfsv4_1_file_layout_devfh4 structures. Each of these structures describes one or more FILEDS4_SIMPLE or FILEDS4_COMPLEX data servers that contribute to a stripe of the file. The nfl_stripe_indices array contains a list of indices into the nfl_ds_fh_list array; an index of zero specifies the first entry in nfl_ds_fh_list. Each successive index selects a nfl_ds_fh_list entry which are to be used next in sequence for that stripe. This allows an arbitrary sequencing through the possible data servers to be encoded compactly. The value of every element in nfl_stripe_indices must be less than the number of elements in the nfl_ds_fh_list array.

When the nfl_stripe_indices array is of zero length, the elements of the nfl_ds_fh_list array are simply used in order, so that the portion of the stripe held by the corresponding entry is determined by its position within the data server list.

If the nfl_stripe_indices array is of non-zero length, there is no requirement that the nfl_stripe_indices and nfl_ds_fh_list arrays have the same number of entries. If the nfl_stripe_indices array has fewer entries than the nfl_ds_fh_list array, this simply means not all entries of nfl_ds_fh_list are in the striping pattern.
Even if nfl_stripe_indices has the same number of entries as the nfl_ds_fh_list array, this does not necessarily mean all entries of nfl_ds_fh_list are used, because nothing prevents an index value from appearing in multiple entries of nfl_stripe_indices.

If the nfl_stripe_indices array has more entries than the nfl_ds_fh_list array, then this simply means index values in nfl_stripe_indices are appearing more than once.

Each nfl_ds_fh_list entry contains a device ID, data server index, and a filehandle. The device ID (field nfldf_ds_id), identifies the data server. The GETDEVICEINFO operation is used to map nfldf_ds_id to a data server address, which will be either a FILEDS4_COMPLEX or FILEDS4_SIMPLE data server address. When the device ID maps to a FILEDS4_COMPLEX data server address server, the data server index (field nfldf_ds_index) indicates the starting element of the to use from the list of device IDs (nflda_comp_ds_list) of the FILEDS4_COMPLEX address. (As discussed in Section 13.4 the nfldf_ds_index field plays a critical role in the flattening of a FILEDS4_COMPLEX device.) If the nfldf_ds_id field maps to a FILEDS4_SIMPLE device, the nfldf_ds_index field has no meaning and should be zero. The filehandle, nfldf_fh, identifies the file on the data server identified by the device ID.

The generic layout hint structure is described in Section 3.2.22. The client uses the layout hint in the layout_hint (Section 5.13.4) attribute to specify the type of layout to be used for a newly created file. The LAYOUT4_NFSV4_1_FILES layout type-specific content for the layout hint is composed of the preferred stripe packing type (field nflh_stripe_type, discussed in Section 13.5), the size of the stripe unit (field nflh_stripe_unit_size), and the width of the stripe (field nflh_stripe_width).

13.4. Interpreting the File Layout

The client is expected to construct a flat list of <data server, file handle> pairs over which the file is striped. A flat data server list contains no FILEDS4_COMPLEX data servers, and is constructed by concatenating each data server encountered while traversing nfl_stripe_indices (or nfl_ds_fh_list in the case of a zero sized nfl_stripe_indices array), while expanding each FILEDS4_COMPLEX data server address. The client must expand the FILEDS4_COMPLEX data server address’s device ID list by starting at the device ID entry of the nflda_comp_ds_list array indexed by nfldf_ds_index, ending with the device ID prior to nfldf_ds_index (or ending with the last entry of the nflda_comp_ds_list array if nfldf_ds_index is zero. All devices IDs in the nflda_comp_ds_list must be consumed; this may require wrapping around the end of the array if nfldf_ds_index is
non-zero. The stripe width is determined by the stripe unit size multiplied by the number of data server entries within the flattened stripe.

Consider the following example:

Given a set of data servers with the following device IDs:

1->{simple}; 2->{complex, ds_list=<3, 4>}; 3->{simple}; 4->{simple}; 5->{simple}; 6->{complex, ds_list=<1, 5>};

Device IDs 1, 3, 4 and 5 identify FILEDS4_SIMPLE data servers. Device ID 2 is a FILEDS4_COMPLEX data server constructed of FILEDS4_SIMPLE data servers 3 and 4. Device ID 6 is a FILEDS4_COMPLEX data server constructed of FILEDS4_SIMPLE data servers 4, 1, and 5.

Within an instance of nfsv4_1_file_layout4, imagine a nfl_ds_fh_list constructed of <device ID, device index, FH> tuples:

\[
ds_fh_list = [<6, 1, 0x17>, <1, 0, 0x12>, <5, 0, 0x22>, <2, 0, 0x13>, <3, 0, 0x14>, <4, 0, 0x15>]
\]

And a nfl_stripe_indices array containing the following indices:

\[
nfl_stripe_indices = [5, 2, 4, 0, 1, 3]
\]

Using nfl_stripe_indices as indices into the nfl_ds_fh_list, we get the following re-ordered list of nfsv4_1_file_layout_devfh4 values:

\[
[4, 0, 0x15>, <5, 0, 0x22>, <2, 0, 0x13>, <6, 3, 0x17>, <1, 0, 0x12>, <5, 0, 0x22>]
\]

Converting the FILEDS4_COMPLEX devices to FILEDS4_SIMPLE devices gives us the following list of 9 FILEDS4_SIMPLE <device ID, FH> tuples.

\[
[<4, 0x15>, <5, 0x22>, <3, 0x13>, <4, 0x13>, <1, 0x17>, <5, 0x17>, <4, 0x14>, <1, 0x12>, <5, 0x22>]
\]

The above list of tuples fully describes the striping pattern. We observe several things. First, the tuples are not 3-tuples; they do not have an index value because FILEDS4_SIMPLE devices do not use the index. Second, each tuple in the sequence represents a destination for each stripe unit in the pattern. Third, device 2 is a FILEDS4_COMPLEX device that gets replaced with devices 3 and 4. Fourth, device 6 is a FILEDS4_COMPLEX device that gets replaced with
devices 1, 5, 4 (and not in the order 4, 1, 5, because the
nfl_ds_fh_list entry for device 6 has a non-zero index value 1, so we
start with second simple device that device 6 maps to and wrap around
to the first simple device after processing the third simple device
that device 6 maps to). Fifth, when converting from FILEDS4_COMPLEX
to FILEDS4_SIMPLE, the filehandle in the FILEDS4_SIMPLE entries that
replace a FILEDS4_COMPLEX entry is from the replaced FILEDS4_COMPLEX
entry. As a result the striping pattern can have the same device ID
appear multiple times, and with different filehandles.

The flattened data server list specifies the pattern over which the
devices must be striped and over which data is written (in increments
of the stripe unit size). It also specifies the filehandle to be
used for each stripe unit of the pattern. A data server that has
more than one stripe unit of a pattern to store each unit may store
those stripes in different files, but to do so, will need unique
filehandles in the data server list, as the previous example showed.
While data servers may be repeated multiple times within the
flattened data server list, if a STRIPE4_DENSE stripe type is used
(see Section 13.5), the same filehandle MUST NOT be used on the same
data server for different stripe units of the same file.

A data file stored on a data server MUST map to a single file as
defined by the metadata server; i.e., data from two files as viewed
by the metadata server MUST NOT be stored within the same data file
on any data server.

13.5. Sparse and Dense Stripe Unit Packing

The nfl_stripe_type field specifies how the data is packed within the
data file on a data server. It allows for two different data
packings: STRIPE4_SPARSE and STRIPE4_DENSE. The stripe type
determines the calculation that must be made to map the client
visible file offset to the offset within the data file located on the
data server.

STRIPE4_SPARSE merely means that the logical offsets of the file as
viewed by a client issuing READs and WRITEs directly to the metadata
server are the same offsets each data server uses when storing a
stripe unit. The effect then, for striping patterns consisting of at
least two stripe units, is for each data server file to be sparse or
holey. So for example, suppose a pattern with 3 stripe units, the
stripe unit size is a block of 4 kilobytes, there are 3 data servers
in the pattern, then the file in data server 1 will have blocks 0, 3,
6, 9, ... filled, data server 2’s file will have blocks 1, 4, 7, 10,
... filled, and data server 3’s file will have blocks 2, 5, 8, 11,
... filled. The unfilled blocks of each file will be holes, hence
the files in each data server are sparse. Logical blocks 0, 3, 6,
... of the file would exist as physical blocks 0, 3, 6 on data server 1, logical blocks 1, 4, 7, ... would exist as physical blocks 1, 4, 7 on data server 2, and logical blocks 2, 5, 8, ... would exist as physical blocks 2, 5, 8 on data server 3.

The STRIPE4_SPARSE stripe type has holes for the octet ranges not exported by that data server, thereby allowing pNFS clients to use the real offset into the data server’s file, regardless of the data server’s position within the pattern. However, if a client attempts I/O to one of the holes, then an error MUST be returned by the data server. Using the above example, if data server 2 received a READ or WRITE request for block 4, the data server would return NFS4ERR_PNFS_IO_HOLE. Thus data servers need to understand the striping pattern in order to support STRIPE4_SPARSE layouts.

STRIPE4_DENSE means that the data server files have no holes. STRIPE4_DENSE might be selected because the data server does not (efficiently) support holey files, e.g. the data server’s file system allocates storage in the gaps, making STRIPE4_SPARSE a waste of space. If the STRIPE4_DENSE stripe type is indicated in the layout, the data files must be packed. Using the example striping pattern and stripe unit size that was used for the STRIPE4_SPARSE example, the STRIPE4_DENSE example would have all data servers’ data files blocks, 0, 1, 2, 3, 4, ... filled. Logical blocks 0, 3, 6, ... of the file would live on blocks 0, 1, 2, ... of the file of data server 1, logical blocks 1, 4, 7, ... of the file would live on blocks 0, 1, 2, ... of the file of data server 2, and logical blocks 2, 5, 8, ... of the file would live on blocks 0, 1, 2, ... of the file of data server 3.

Since the STRIPE4_DENSE layout does not leave holes on the data servers, the pNFS client is allowed to write to any offset of any data file of any data server in the stripe. Thus the the data servers need not know the file’s striping pattern.

The calculation to determine the octet offset within the data file for dense data server layouts is:

```plaintext

stripe_width = stripe_unit_size * N;

where N = number of <data server, filehandle pairs>
in flattened nfl_ds_fh_list

data_file_offset = floor(file_offset / stripe_width)

* stripe_unit_size

+ file_offset % stripe_unit_size
```

Regardless of the data server layout, the calculation to determine the index into the device array is the same:
data_server_idx = floor(file_offset / stripe_unit_size) mod N

Section 13.12 describe the semantics for dealing with reads to holes within the striped file. This is of particular concern, since each individual component stripe file (i.e., the component of the striped file that lives on a particular data server) may be of different length. Thus, clients may experience 'short' reads when reading off the end of one of these component files.

13.6. Data Server Multipathing

The NFSv4.1 file layout supports multipathing to "equivalent" (defined later in this section) data servers. Data server-level multipathing is primarily of use in the case of a data server failure; it allows the client to switch to another data server that is exporting the same data stripe unit, without having to contact the metadata server for a new layout.

To support data server multipathing, there is an array of data server network addresses (nflda_simp_ds_list) within the FILEDS4_SIMPLE case of the nfsv4_1_file_layout_ds_addr4 switched union. This array represents an ordered list of data server (each identified by a network address) where the first element has the highest priority. Each data server in the list MUST be equivalent to every other data server in the list and each data server MUST be attempted in the order specified.

Two data servers are equivalent if they export the same system image (e.g., the stateids and filehandles that they use are the same) and provide the same consistency guarantees. Two equivalent data servers must also have sufficient connections to the storage, such that writing to one data server is equivalent to writing to another; this also applies to reading. Also, if multiple copies of the same data exist, reading from one must provide access to all existing copies. As such, it is unlikely that multipathing will provide additional benefit in the case of an I/O error.

[[Comment.16: [NOTE: the error cases in which a client is expected to attempt an equivalent data server should be specified.]]]

13.7. Operations Issued to NFSv4.1 Data Servers

Clients MUST use the filehandle described within the layout when accessing data on NFSv4.1 data servers. When using the layout’s filehandle, the client MUST only issue the NULL procedure and the COMPOUND procedure’s BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CREATE_SESSION, COMMIT, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, READ, WRITE, PUTFH, SECINFO_NO_NAME, SET_SSV, and
SEQUENCE operations to the NFSv4.1 data server associated with that data server filehandle. If a client issues an operation to the data server other than those specified above, using the filehandle and data server listed in the file’s layout, that data server MUST return an error to the client (unless the pNFS server has chosen to not disambiguate the data server filehandle from the metadata server filehandle, and/or the pNFS server has chosen to not disambiguate the metadata server client ID from the data server client ID). The client MUST follow the instruction implied by the layout (i.e., which filehandles to use on which data servers). As described in Section 12.5.1, a client MUST NOT issue I/Os to data servers for which it does not hold a valid layout. The data servers MAY reject such requests.

GETATTR and SETATTR MUST be directed to the metadata server. In the case of a SETATTR of the size attribute, the control protocol is responsible for propagating size updates/truncations to the data servers. In the case of extending WRITEs to the data servers, the new size must be visible on the metadata server once a LAYOUTCOMMIT has completed (see Section 12.5.3.2). Section 13.12, describes the mechanism by which the client is to handle data server files that do not reflect the metadata server’s size.

13.8. COMMIT Through Metadata Server

The nfl_commit_through_mds field in the file layout (data type nfsv4_1_file_layout4) gives the metadata server the preferred way of performing COMMIT. If this field is TRUE, the client SHOULD send COMMIT to the metadata server instead of sending it to the same data server to which the associated WRITEs were sent. In order to maintain the current NFSv4.1 commit and recovery model, all the data servers MUST return a common writeverf verifier in all WRITE responses for a given file layout. The value of the writeverf verifier MUST be changed at the metadata server or any data server that is referenced in the layout, whenever there is a server event that can possibly lead to loss of uncommitted data. The scope of the verifier can be for a file or for the entire pNFS server. It might be more difficult for the server to maintain the verifier at the file level but the benefit is that only events that impact a given file will require recovery action.

The single COMMIT to the metadata server will return a verifier and the client should compare it to all the verifiers from the WRITEs and fail the COMMIT if there is any mismatched verifiers. If COMMIT to the metadata server fails, the client should reissue WRITEs for all the modified data in the file. The client should treat modified data with a mismatched verifier as a WRITE failure and try to recover by reissuing the WRITEs to the original data server or using another...
path to that data if the layout has not been recalled. Another option the client has is getting a new layout or just rewrite the data through the metadata server. If the flag nfl_commit_through_mds is FALSE, the client should not send COMMIT to the metadata server. Although it is valid to send COMMIT to the metadata server it should be used only to commit data that was written through the metadata server. See Section 12.7.6 for recovery options.

13.9. Global Stateid Requirements

Note, there are no stateids embedded within the layout returned by the metadata server to the pNFS client. The client uses a stateid returned previously by the metadata server (including results from OPEN -- a delegation stateid is acceptable as well as a non-delegation stateid -- lock operations, WANT_DELEGATION, and also from the CB_PUSH_DELEG callback operation) or a special stateid to perform I/O on the data servers, as in regular NFSv4.1. Special stateid usage for I/O is subject to the NFSv4.1 protocol specification. The stateid used for I/O MUST have the same effect and be subject to the same validation on data server as it would if the I/O was being performed on the metadata server itself in the absence of pNFS. This has the implication that stateids are globally valid on both the metadata and data servers. This requires the metadata server to propagate changes in lock and open state to the data servers, so that the data servers can validate I/O accesses. This is discussed further in Section 13.11. Depending on when stateids are propagated, the existence of a valid stateid on the data server may act as proof of a valid layout.

13.10. The Layout Iomode

The layout iomode need not be used by the metadata server when servicing NFSv4.1 file-based layouts, although in some circumstances it may be useful to use. For example, if the server implementation supports reading from read-only replicas or mirrors, it would be useful for the server to return a layout enabling the client to do so. As such, the client SHOULD set the iomode based on its intent to read or write the data. The client may default to an iomode of LAYOUTIOMODE4_RW. The iomode need not be checked by the data servers when clients perform I/O. However, the data servers SHOULD still validate that the client holds a valid layout and return an error if the client does not.

13.11. Data Server State Propagation

Since the metadata server, which handles lock and open-mode state changes, as well as ACLs, may not be co-located with the data servers where I/O access are validated, as such, the server implementation
MUST take care of propagating changes of this state to the data servers. Once the propagation to the data servers is complete, the full effect of those changes must be in effect at the data servers. However, some state changes need not be propagated immediately, although all changes SHOULD be propagated promptly. These state propagations have an impact on the design of the control protocol, even though the control protocol is outside of the scope of this specification. Immediate propagation refers to the synchronous propagation of state from the metadata server to the data server(s); the propagation must be complete before returning to the client.

13.11.1. Lock State Propagation

If the pNFS server supports mandatory locking, any mandatory locks on a file MUST be made effective at the data servers before the request that establishes them returns to the caller. Thus, mandatory lock state MUST be synchronously propagated to the data servers. On the other hand, since advisory lock state is not used for checking I/O accesses at the data servers, there is no semantic reason for propagating advisory lock state to the data servers. However, since all lock, unlock, open downgrades and upgrades MAY affect the "seqid" stored within the stateid (see Section 8.1.3.1), the stateid changes may cause difficulty if this state is not propagated. Thus, when a client uses a stateid on a data server for I/O with a newer "seqid" number than the one the data server has, the data server may need to query the metadata server and get any pending updates to that stateid. This allows stateid sequence number changes to be propagated lazily, on-demand.

Since updates to advisory locks neither confer nor remove privileges, these changes need not be propagated immediately, and may not need to be propagated promptly. The updates to advisory locks need only be propagated when the data server needs to resolve a question about a stateid. In fact, if record locking is not mandatory (i.e., is advisory) the clients are advised not to use the lock-based stateids for I/O at all. The stateids returned by open are sufficient and eliminate overhead for this kind of state propagation.

13.11.2. Open-mode Validation

Open-mode validation MUST be performed against the open mode(s) held by the data servers. However, the server implementation may not always require the immediate propagation of changes. Reduction in access because of CLOSEs or DOWNGRADES does not have to be propagated immediately, but SHOULD be propagated promptly; whereas changes due to revocation MUST be propagated immediately. On the other hand, changes that expand access (e.g., new OPEN’s and upgrades) do not have to be propagated immediately but the data server SHOULD NOT
reject a request because of open mode issues without making sure that the upgrade is not in flight.

### 13.11.3. File Attributes

Since the SETATTR operation has the ability to modify state that is visible on both the metadata and data servers (e.g., the size), care must be taken to ensure that the resultant state across the set of data servers is consistent; especially when truncating or growing the file.

As described earlier, the LAYOUTCOMMIT operation is used to ensure that the metadata is synchronized with changes made to the data servers. For the NFSv4.1-based data storage protocol, it is necessary to re-synchronize state such as the size attribute, and the setting of mtime/change/atime. See Section 12.5.3 for a full description of the semantics regarding LAYOUTCOMMIT and attribute synchronization. It should be noted, that by using an NFSv4.1-based layout type, it is possible to synchronize this state before LAYOUTCOMMIT occurs. For example, the control protocol can be used to query the attributes present on the data servers.

Any changes to file attributes that control authorization or access as reflected by ACCESS calls or READs and WRITEs on the metadata server, MUST be propagated to the data servers for enforcement on READ and WRITE I/O calls. If the changes made on the metadata server result in more restrictive access permissions for any user, those changes MUST be propagated to the data servers synchronously.

The OPEN operation (Section 17.16.5) does not impose any requirement that I/O operations on an open file have the same credentials as the OPEN itself, and so requires the server’s READ and WRITE operations to perform appropriate access checking. Changes to ACLs also require new access checking by READ and WRITE on the server. The propagation of access right changes due to changes in ACLs may be asynchronous only if the server implementation is able to determine that the updated ACL is not more restrictive for any user specified in the old ACL. Due to the relative infrequency of ACL updates, it is suggested that all changes be propagated synchronously.

### 13.12. Data Server Component File Size

A potential problem exists when a component data file on a particular data server is grown past EOF; the problem exists for both dense and sparse layouts. Imagine the following scenario: a client creates a new file (size == 0) and writes to octet 131072; the client then seeks to the beginning of the file and reads octet 100. The client should receive 0s back as a result of the READ. However, if the READ
falls on a data server different than that that received client’s original WRITE, the data server servicing the READ may still believe that the file’s size is at 0 and return no data with the EOF flag set. The data server can only return 0s if it knows that the file’s size has been extended. This would require the immediate propagation of the file’s size to all data servers, which is potentially very costly. Therefore, the client that has initiated the extension of the file’s size MUST be prepared to deal with these EOF conditions; the EOF’ed or short READs will be treated as a hole in the file and the NFS client will substitute 0s for the data when the offset is less than the client’s view of the file size.

The NFSv4.1 protocol only provides close to open file data cache semantics; meaning that when the file is closed all modified data is written to the server. When a subsequent OPEN of the file is done, the change attribute is inspected for a difference from a cached value for the change attribute. For the case above, this means that a LAYOUTCOMMIT will be done at close (along with the data WRITEs) and will update the file’s size and change attribute. Access from another client after that point will result in the appropriate size being returned.

13.13. Recovery Considerations

As described in Section 12.7, the layout type-specific storage protocol is responsible for handling the effects of I/Os started before lease expiration, extending through lease expiration. The NFSv4.1 file layout type prevents all I/Os from being executed after lease expiration, without relying on a precise client lease timer and without requiring data servers to maintain lease timers.

It works as follows. As described in Section 13.1, in COMPOUND procedure requests to the data server, the data filehandle provided by the PUTFH operation and the stateid in the READ or WRITE operation are used to validate that the client has a valid layout for the I/O being performed, if it does not, the I/O is rejected. Before the metadata server takes any action to invalidate a layout given out by a previous instance, it must make sure that all layouts from that previous instance are invalidated at the data servers.

This means that a metadata server may not restripe a file until it has contacted all of the data servers to invalidate the layouts from the previous instance nor may it give out mandatory locks that conflict with layouts from the previous instance without either doing a specific invalidation (as it would have to do anyway) or doing a global data server invalidation.

The NFSv4.1 file layout type MUST adhere to the security considerations outlined in Section 12.9. NFSv4.1 data servers must make all of the required access checks on each READ or WRITE I/O as determined by the NFSv4.1 protocol. If the metadata server would deny READ or WRITE operation on a given file due its ACL, mode attribute, open mode, open deny mode, mandatory lock state, or any other attributes and state, the data server MUST also deny the READ or WRITE operation. This impacts the control protocol and the propagation of state from the metadata server to the data servers; see Section 13.11 for more details.

The methods for authentication, integrity, and privacy for file layout-based data servers are the same as that used for metadata servers. Metadata and data servers use ONC RPC security flavors to authenticate, and SECINFO and SECINFO_NO_NAME to negotiate the security mechanism and services to be used.

For a given file object, a metadata server MAY require different security parameters (secinfo4 value) than the data server. For a given file object with multiple data servers, the secinfo4 value SHOULD be the same across all data servers.

If an NFSv4.1 implementation supports pNFS and supports NFSv4.1 file layouts, then the implementation MUST support the SECINFO_NO_NAME operation, on both the metadata and data servers.

14. Internationalization

The primary issue in which NFS version 4 needs to deal with internationalization, or I18N, is with respect to file names and other strings as used within the protocol. The choice of string representation must allow reasonable name/string access to clients which use various languages. The UTF-8 encoding of the UCS as defined by ISO10646 [10] allows for this type of access and follows the policy described in "IETF Policy on Character Sets and Languages", RFC2277 [11].

RFC3454 [12], otherwise know as "stringprep", documents a framework for using Unicode/UTF-8 in networking protocols, so as "to increase the likelihood that string input and string comparison work in ways that make sense for typical users throughout the world." A protocol must define a profile of stringprep "in order to fully specify the processing options." The remainder of this Internationalization section defines the NFS version 4 stringprep profiles. Much of terminology used for the remainder of this section comes from...
There are three UTF-8 string types defined for NFS version 4: utf8str_cs, utf8str_cis, and utf8str_mixed. Separate profiles are defined for each. Each profile defines the following, as required by stringprep:

- The intended applicability of the profile
- The character repertoire that is the input and output to stringprep (which is Unicode 3.2 for referenced version of stringprep)
- The mapping tables from stringprep used (as described in section 3 of stringprep)
- Any additional mapping tables specific to the profile
- The Unicode normalization used, if any (as described in section 4 of stringprep)
- The tables from stringprep listing of characters that are prohibited as output (as described in section 5 of stringprep)
- The bidirectional string testing used, if any (as described in section 6 of stringprep)
- Any additional characters that are prohibited as output specific to the profile

Stringprep discusses Unicode characters, whereas NFS version 4 renders UTF-8 characters. Since there is a one-to-one mapping from UTF-8 to Unicode, when the remainder of this document refers to Unicode, the reader should assume UTF-8.

Much of the text for the profiles comes from RFC3491 [13].

14.1. Stringprep profile for the utf8str_cs type

Every use of the utf8str_cs type definition in the NFS version 4 protocol specification follows the profile named nfs4_cs_prep.

14.1.1. Intended applicability of the nfs4_cs_prep profile

The utf8str_cs type is a case sensitive string of UTF-8 characters. Its primary use in NFS Version 4 is for naming components and pathnames. Components and pathnames are stored on the server’s file system. Two valid distinct UTF-8 strings might be the same after
processing via the utf8str_cs profile. If the strings are two names inside a directory, the NFS version 4 server will need to either:

- disallow the creation of a second name if it’s post processed form
  collides with that of an existing name, or
- allow the creation of the second name, but arrange so that after
  post processing, the second name is different than the post
  processed form of the first name.

14.1.2. Character repertoire of nfs4_cs_prep

The nfs4_cs_prep profile uses Unicode 3.2, as defined in stringprep’s
Appendix A.1

14.1.3. Mapping used by nfs4_cs_prep

The nfs4_cs_prep profile specifies mapping using the following tables
from stringprep:

Table B.1

Table B.2 is normally not part of the nfs4_cs_prep profile as it is
primarily for dealing with case-insensitive comparisons. However, if
the NFS version 4 file server supports the case_insensitive file
system attribute, and if case_insensitive is true, the NFS version 4
server MUST use Table B.2 (in addition to Table B1) when processing
utf8str_cs strings, and the NFS version 4 client MUST assume Table
B.2 (in addition to Table B.1) are being used.

If the case_preserving attribute is present and set to false, then
the NFS version 4 server MUST use table B.2 to map case when
processing utf8str_cs strings. Whether the server maps from lower to
upper case or the upper to lower case is an implementation
dependency.

14.1.4. Normalization used by nfs4_cs_prep

The nfs4_cs_prep profile does not specify a normalization form. A
later revision of this specification may specify a particular
normalization form. Therefore, the server and client can expect that
they may receive unnormalized characters within protocol requests and
responses. If the operating environment requires normalization, then
the implementation must normalize utf8str_cs strings within the
protocol before presenting the information to an application (at the
client) or local file system (at the server).
14.1.5. Prohibited output for nfs4_cs_prep

The nfs4_cs_prep profile specifies prohibiting using the following tables from stringprep:

Table C.3
Table C.4
Table C.5
Table C.6
Table C.7
Table C.8
Table C.9

14.1.6. Bidirectional output for nfs4_cs_prep

The nfs4_cs_prep profile does not specify any checking of bidirectional strings.

14.2. Stringprep profile for the utf8str_cis type

Every use of the utf8str_cis type definition in the NFS version 4 protocol specification follows the profile named nfs4_cis_prep.

14.2.1. Intended applicability of the nfs4_cis_prep profile

The utf8str_cis type is a case insensitive string of UTF-8 characters. Its primary use in NFS Version 4 is for naming NFS servers.

14.2.2. Character repertoire of nfs4_cis_prep

The nfs4_cis_prep profile uses Unicode 3.2, as defined in stringprep’s Appendix A.1

14.2.3. Mapping used by nfs4_cis_prep

The nfs4_cis_prep profile specifies mapping using the following tables from stringprep:

Table B.1
14.2.4. Normalization used by nfs4_cis_prep

The nfs4_cis_prep profile specifies using Unicode normalization form KC, as described in stringprep.

14.2.5. Prohibited output for nfs4_cis_prep

The nfs4_cis_prep profile specifies prohibiting using the following tables from stringprep:

Table C.1.2
Table C.2.2
Table C.3
Table C.4
Table C.5
Table C.6
Table C.7
Table C.8
Table C.9

14.2.6. Bidirectional output for nfs4_cis_prep

The nfs4_cis_prep profile specifies checking bidirectional strings as described in stringprep’s section 6.

14.3. Stringprep profile for the utf8str_mixed type

Every use of the utf8str_mixed type definition in the NFS version 4 protocol specification follows the profile named nfs4_mixed_prep.

14.3.1. Intended applicability of the nfs4_mixed_prep profile

The utf8str_mixed type is a string of UTF-8 characters, with a prefix that is case sensitive, a separator equal to ‘@’, and a suffix that is fully qualified domain name. Its primary use in NFS Version 4 is for naming principals identified in an Access Control Entry.
14.3.2. Character repertoire of nfs4_mixed_prep

The nfs4_mixed_prep profile uses Unicode 3.2, as defined in stringprep's Appendix A.1

14.3.3. Mapping used by nfs4_cis_prep

For the prefix and the separator of a utf8str_mixed string, the nfs4_mixed_prep profile specifies mapping using the following table from stringprep:

Table B.1

For the suffix of a utf8str_mixed string, the nfs4_mixed_prep profile specifies mapping using the following tables from stringprep:

Table B.1
Table B.2

14.3.4. Normalization used by nfs4_mixed_prep

The nfs4_mixed_prep profile specifies using Unicode normalization form KC, as described in stringprep.

14.3.5. Prohibited output for nfs4_mixed_prep

The nfs4_mixed_prep profile specifies prohibiting using the following tables from stringprep:

Table C.1.2
Table C.2.2
Table C.3
Table C.4
Table C.5
Table C.6
Table C.7
Table C.8
Table C.9
14.3.6. Bidirectional output for nfs4_mixed_prep

The nfs4_mixed_prep profile specifies checking bidirectional strings as described in stringprep’s section 6.

14.4. UTF-8 Related Errors

Where the client sends an invalid UTF-8 string, the server should return an NFS4ERR_INVAL (Table 8) error. This includes cases in which inappropriate prefixes are detected and where the count includes trailing bytes that do not constitute a full UCS character.

Where the client supplied string is valid UTF-8 but contains characters that are not supported by the server as a value for that string (e.g. names containing characters that have more than two octets on a file system that supports Unicode characters only), the server should return an NFS4ERR_BADCHAR (Table 8) 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 server should return the error NFS4ERR_BADNAME (Table 8). This includes situations in which the server file system imposes a normalization constraint on name strings, but will also include such situations as file system prohibitions of "." and ".." as file names for certain operations, and other such constraints.

15. Error Values

NFS error numbers are assigned to failed operations within a compound request. A compound request contains a number of NFS operations that have their results encoded in sequence in a compound reply. The results of successful operations will consist of an NFS4_OK status followed by the encoded results of the operation. If an NFS operation fails, an error status will be entered in the reply and the compound request will be terminated.

15.1. Error Definitions

<table>
<thead>
<tr>
<th>Error</th>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFS4_OK</td>
<td>0</td>
<td>Indicates the operation completed successfully.</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>NFS4ERR_ACCESS</td>
<td>13</td>
<td>Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with NFS4ERR_PERM, which restricts itself to owner or privileged user permission failures.</td>
</tr>
<tr>
<td>NFS4ERR_ATTRNOTSUPP</td>
<td>10032</td>
<td>An attribute specified is not supported by the server. Does not apply to the GETATTR operation.</td>
</tr>
<tr>
<td>NFS4ERR_ADMIN_REVOKED</td>
<td>10047</td>
<td>Due to administrator intervention, the lockowner’s record locks, share reservations, and delegations have been revoked by the server.</td>
</tr>
<tr>
<td>NFS4ERR_BACK_CHAN_BUSY</td>
<td>10057</td>
<td>The session cannot be destroyed because the server has callback requests outstanding.</td>
</tr>
<tr>
<td>NFS4ERR_BADCHAR</td>
<td>10040</td>
<td>A UTF-8 string contains a character which is not supported by the server in the context in which it is being used.</td>
</tr>
<tr>
<td>NFS4ERR_BAD_COOKIE</td>
<td>10003</td>
<td>READDIR cookie is stale.</td>
</tr>
<tr>
<td>NFS4ERR_BADHANDLE</td>
<td>10001</td>
<td>Illegal NFS filehandle. The filehandle failed internal consistency checks.</td>
</tr>
<tr>
<td>NFS4ERR_BADIOMODE</td>
<td>10049</td>
<td>Layout iomode is invalid.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NFS4ERR_BADLAYOUT</td>
<td>10050</td>
<td>Layout specified is invalid.</td>
</tr>
<tr>
<td>NFS4ERR_BADNAME</td>
<td>10041</td>
<td>A name string in a request consists 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.</td>
</tr>
<tr>
<td>NFS4ERR_BADOWNER</td>
<td>10039</td>
<td>An owner, owner_group, or ACL attribute value can not be translated to local representation.</td>
</tr>
<tr>
<td>NFS4ERR_BAD_SESSION_DIGEST</td>
<td>10051</td>
<td>The digest used in a SET_SSV or BIND_CONN_TO_SESSION request is not valid.</td>
</tr>
<tr>
<td>NFS4ERR_BADTYPE</td>
<td>10007</td>
<td>An attempt was made to create an object of a type not supported by the server.</td>
</tr>
<tr>
<td>NFS4ERR_BAD_RANGE</td>
<td>10042</td>
<td>The range for a LOCK, LOCKT, or LOCKU operation is not appropriate to the allowable range of offsets for the server.</td>
</tr>
<tr>
<td>NFS4ERR_BAD_SEQID</td>
<td>10026</td>
<td>The sequence number in a locking request is neither the next expected number or the last number processed. This error does not apply to and should never be generated in NFSv4.1.</td>
</tr>
<tr>
<td>NFS4ERR_BADSESSION</td>
<td>10052</td>
<td>TDB</td>
</tr>
<tr>
<td>NFS4ERR_BADSLOT</td>
<td>10053</td>
<td>TDB</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| 10025             | **NFS4ERR_BAD_STATEID**  
A stateid generated by the current server instance, but which does not designate any locking state (either current or superseded) for a current lockowner-file pair, was used. |
| 10036             | **NFS4ERR_BADXDR**  
The server encountered an XDR decoding error while processing an operation. |
| 10017             | **NFS4ERR_CLID_INUSE**  
The EXCHANGE_ID operation has found that a client ID is already in use by another client. |
| 10074             | **NFS4ERR_CLIENTID_BUSY**  
The DESTROY_CLIENTID operation has found there are has sessions and/or stateids bound to the client ID. |
| 10054             | **NFS4ERR_COMPLETE_ALREADY**  
A RECLAIM_COMPLETE operation was done by a client which had already performed one. |
| 10055             | **NFS4ERR_CONN_NOTBOUNDTO_SESSION**  
The connection is not bound to the specified session. |
| 10073             | **NFS4ERR_CONN_BINDING_NOT_ENFORCED**  
Client is trying use enforced connection binding, but it disabled enforcement when the session was created. |
| 10045             | **NFS4ERR_DEADLOCK**  
The server has been able to determine a file locking deadlock condition for a blocking lock request. |
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFS4ERR_DELAY</td>
<td>10008</td>
<td>The server initiated the request, but was not able to complete it in a timely fashion. The client should wait and then try the request with a new RPC transaction ID. For example, this error should be returned from a server that supports hierarchical storage and receives a request to process a file that has been migrated. In this case, the server should start the immigration process and respond to the client with this error. This error may also occur when a necessary delegation recall makes processing a request in a timely fashion impossible.</td>
</tr>
<tr>
<td>NFS4ERR_DELEG_ALREADY_WANTED</td>
<td>10056</td>
<td>The client has already registered that it wants a delegation.</td>
</tr>
<tr>
<td>NFS4ERR_DENIED</td>
<td>10010</td>
<td>An attempt to lock a file is denied. Since this may be a temporary condition, the client is encouraged to retry the lock request until the lock is accepted.</td>
</tr>
<tr>
<td>NFS4ERR_DQUOT</td>
<td>69</td>
<td>Resource (quota) hard limit exceeded. The user’s resource limit on the server has been exceeded.</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NFS4ERR_EXIST</td>
<td>17</td>
<td>File exists. The file specified already exists.</td>
</tr>
<tr>
<td>NFS4ERR_EXPIRED</td>
<td>10011</td>
<td>A lease has expired that is being used in the current operation.</td>
</tr>
<tr>
<td>NFS4ERR_FBIG</td>
<td>27</td>
<td>File too large. The operation would have caused a file to grow beyond the server’s limit.</td>
</tr>
<tr>
<td>NFS4ERR_FHEXPIRED</td>
<td>10014</td>
<td>The filehandle provided is volatile and has expired at the server.</td>
</tr>
<tr>
<td>NFS4ERR_FILE_OPEN</td>
<td>10046</td>
<td>The operation can not be successfully processed because a file involved in the operation is currently open.</td>
</tr>
<tr>
<td>NFS4ERR_GRACE</td>
<td>10013</td>
<td>The server is in its recovery or grace period which should match the lease period of the server.</td>
</tr>
<tr>
<td>NFS4ERR_INVAL</td>
<td>22</td>
<td>Invalid argument or unsupported argument for an operation. Two examples are attempting a READLINK on an object other than a symbolic link or specifying a value for an enum field that is not defined in the protocol (e.g. nfs_ftype4).</td>
</tr>
<tr>
<td>NFS4ERR_IO</td>
<td>5</td>
<td>I/O error. A hard error (for example, a disk error) occurred while processing the requested operation.</td>
</tr>
</tbody>
</table>
| Code     | Description                                                                 
|----------|-----------------------------------------------------------------------------
<p>| NFSERR_ISDIR | 21 | Is a directory. The caller specified a directory in a non-directory operation. |
| NFSERR_LAYOUTTRYLATER | 10058 | Layouts are temporarily unavailable for the file, client should retry later. |
| NFSERR_LAYOUTUNAVAILABLE | 10059 | Layouts are not available for the file or its containing file system. |
| NFSERR_LEASE_MOVED | 10031 | A lease being renewed is associated with a file system that has been migrated to a new server. |
| NFSERR_LOCKED | 10012 | A read or write operation was attempted on a locked file. |
| NFSERR_LOCK_NOTSUPP | 10043 | Server does not support atomic upgrade or downgrade of locks. |
| NFSERR_LOCK_RANGE | 10028 | A lock request is operating on a sub-range of a current lock for the lock owner and the server does not support this type of request. |
| NFSERR_LOCKS_HELD | 10037 | A CLOSE was attempted and file locks would exist after the CLOSE. |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Code Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10021</td>
<td>NFS4ERR_MINOR_VERS_MISMATCH</td>
<td>The server has received a request that specifies an unsupported minor version. The server must return a COMPOUND4res with a zero length operations result array.</td>
</tr>
<tr>
<td>10063</td>
<td>NFS4ERR_SEQ_MISORDERED</td>
<td>The requester sent a SEQUENCE or CB_SEQUENCE operation with an invalid sequenceid.</td>
</tr>
<tr>
<td>10064</td>
<td>NFS4ERR_SEQUENCE_POS</td>
<td>The requester sent a COMPOUND or CB_COMPOUND with a SEQUENCE or CB_SEQUENCE operation that was not the first operation.</td>
</tr>
<tr>
<td>31</td>
<td>NFS4ERR_MLINK</td>
<td>Too many hard links.</td>
</tr>
<tr>
<td>10019</td>
<td>NFS4ERR_MOVED</td>
<td>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 &quot;fs_locations&quot; attribute for the current filehandle. For further discussion, refer to the section &quot;Multi-server Name Space&quot;.</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>NFS4ERR_NAME_TOO_LONG</td>
<td>63</td>
<td>The filename in an operation was too long.</td>
</tr>
<tr>
<td>NFS4ERR_NOENT</td>
<td>2</td>
<td>No such file or directory. The file or directory name specified does not exist.</td>
</tr>
<tr>
<td>NFS4ERR_NO_FILE_HANDLE</td>
<td>10020</td>
<td>The logical current filehandle value (or, in the case of RESTOREFH, the saved filehandle value) has not been set properly. This may be a result of a malformed COMPOUND operation (i.e. no PUTFH or PUTROOTFH before an operation that requires the current filehandle be set).</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NFS4ERR_NO_GRACE</td>
<td>10033</td>
<td>A reclaim of client state was attempted in circumstances in which the server cannot guarantee that conflicting state has not been provided to another client. This can occur because the reclaim has been done outside of the grace period of the server, after the client has done a RECLAIM_COMPLETE operation, or because previous operations have created a situation in which the server is not able to determine that a reclaim-interfering edge condition does not exist.</td>
</tr>
<tr>
<td>NFS4ERR_NOMATCHING_LAYOUT</td>
<td>10060</td>
<td>Client has no matching layout (segment) to return.</td>
</tr>
<tr>
<td>NFS4ERR_NOSPC</td>
<td>28</td>
<td>No space left on device. The operation would have caused the server’s file system to exceed its limit.</td>
</tr>
<tr>
<td>NFS4ERR_NOTDIR</td>
<td>20</td>
<td>Not a directory. The caller specified a non-directory in a directory operation.</td>
</tr>
<tr>
<td>NFS4ERR_NOTEMPTY</td>
<td>66</td>
<td>An attempt was made to remove a directory that was not empty.</td>
</tr>
<tr>
<td>NFS4ERR_NOTSUPP</td>
<td>10004</td>
<td>Operation is not supported.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>NFS4ERR_NOT_SAME</td>
<td>10027</td>
<td>This error is returned by the VERIFY operation to signify that the attributes compared were not the same as provided in the client’s request.</td>
</tr>
<tr>
<td>NFS4ERR_NXIO</td>
<td>6</td>
<td>I/O error. No such device or address.</td>
</tr>
<tr>
<td>NFS4ERR_OLD_STATEID</td>
<td>10024</td>
<td>A stateid which designates the locking state for a lockowner-file at an earlier time was used. This error does not apply to and should never be generated in NFSv4.1.</td>
</tr>
<tr>
<td>NFS4ERR_OPENMODE</td>
<td>10038</td>
<td>The client attempted a READ, WRITE, LOCK or SETATTR operation not sanctioned by the stateid passed (e.g. writing to a file opened only for read).</td>
</tr>
<tr>
<td>NFS4ERR_OP_ILLEGAL</td>
<td>10044</td>
<td>An illegal operation value has been specified in the argop field of a COMPOUND or CB_COMPOUND procedure.</td>
</tr>
<tr>
<td>NFS4ERR_OP_NOT_IN_SESSION</td>
<td>10070</td>
<td>The COMPOUND or CB_COMPOUND contains an operation that requires a SEQUENCE or CB_SEQUENCE operation to precede it in order to establish a session.</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NFS4ERR_PERM</td>
<td>1</td>
<td>Not owner. The operation was not allowed because the caller is either not a privileged user (root) or not the owner of the target of the operation.</td>
</tr>
<tr>
<td>NFS4ERR_PNFS_IO_HOLE</td>
<td>10075</td>
<td>The pNFS client has attempted to read from or write to a illegal hole of a file of a data server that is using the STRIPE4_SPARSE stripe type. See Section 13.5.</td>
</tr>
<tr>
<td>NFS4ERR_RECALLCONFLICT</td>
<td>10061</td>
<td>Layout is unavailable due to a conflicting LAYOUTRECALL that is in progress.</td>
</tr>
<tr>
<td>NFS4ERR_RECLAIM_BAD</td>
<td>10034</td>
<td>The reclaim provided by the client does not match any of the server’s state consistency checks and is bad.</td>
</tr>
<tr>
<td>NFS4ERR_RECLAIM_CONFLICT</td>
<td>10035</td>
<td>The reclaim provided by the client has encountered a conflict and can not be provided. Potentially indicates a misbehaving client.</td>
</tr>
<tr>
<td>NFS4ERR_REP_TOO_BIG</td>
<td>10066</td>
<td>The reply to a COMPOUND or CB_COMPOUND would exceed the channel’s negotiated maximum response size.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE</td>
<td>10067 The reply to a COMPOUND or CB_COMPOUND would exceed the channel’s negotiated maximum size for replies cached in the reply cache.</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_REQ_TOO_BIG</td>
<td>10065 The COMPOUND or CB_COMPOUND request exceeds the channel’s negotiated maximum size for requests.</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_RESTOREFH</td>
<td>10030 The RESTOREFH operation does not have a saved filehandle (identified by SAVEFH) to operate upon.</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_RETRY_UNCACHED_REP</td>
<td>10068 The requester has attempted a retry of COMPOUND or CB_COMPOUND which it previously requested not be placed in the reply cache.</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_ROFS</td>
<td>30 Read-only file system. A modifying operation was attempted on a read-only file system.</td>
<td></td>
</tr>
<tr>
<td>NFS4ERRSAME</td>
<td>10009 This error is returned by the NVERIFY operation to signify that the attributes compared were the same as provided in the client’s request.</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>NFS4ERR_SERVERFAULT</td>
<td>10006</td>
<td>An error occurred on the server which does not map to any of the legal NFS version 4 protocol error values. The client should translate this into an appropriate error. UNIX clients may choose to translate this to EIO.</td>
</tr>
<tr>
<td>NFS4ERR_SHARE_DENIED</td>
<td>10015</td>
<td>An attempt to OPEN a file with a share reservation has failed because of a share conflict.</td>
</tr>
<tr>
<td>NFS4ERR_STALE</td>
<td>70</td>
<td>Invalid filehandle. The filehandle given in the arguments was invalid. The file referred to by that filehandle no longer exists or access to it has been revoked.</td>
</tr>
<tr>
<td>NFS4ERR_STALE_CLIENTID</td>
<td>10022</td>
<td>A client ID not recognized by the server was used in a locking or CREATE_SESSION request.</td>
</tr>
<tr>
<td>NFS4ERR_STALE_STATEID</td>
<td>10023</td>
<td>A stateid generated by an earlier server instance was used.</td>
</tr>
<tr>
<td>NFS4ERR_SYMLINK</td>
<td>10029</td>
<td>The current filehandle provided for a LOOKUP is not a directory but a symbolic link. Also used if the final component of the OPEN path is a symbolic link.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NFS4ERR_TOOSMALL</td>
<td>10005</td>
<td>The encoded response to a READDIR request exceeds the size limit set by the initial request.</td>
</tr>
<tr>
<td>NFS4ERR_TOO_MANY_OPS</td>
<td>10070</td>
<td>The COMPOUND or CB_COMPOUND request has too many operations.</td>
</tr>
<tr>
<td>NFS4ERR UNKNOWN_LAYOUTTYPE</td>
<td>10062</td>
<td>Layout type is unknown.</td>
</tr>
<tr>
<td>NFS4ERRUnsafe_COMPOUND</td>
<td>10069</td>
<td>The client has sent a COMPOUND request with an unsafe mix of operations.</td>
</tr>
<tr>
<td>NFS4ERR_WRONGSEC</td>
<td>10016</td>
<td>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 retry the operation.</td>
</tr>
<tr>
<td>NFS4ERR_XDEV</td>
<td>18</td>
<td>Attempt to do an operation between different fsids.</td>
</tr>
</tbody>
</table>

Table 8

15.2. Operations and their valid errors

Mappings of valid error returns for each protocol operation
<table>
<thead>
<tr>
<th>Operation</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>NFS4ERR_ACCESS, NFS4ERR_BADHANDLE, NFS4ERR_BADXDR, NFS4ERR_DELAY,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_NOFILEHANDLE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REQ_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_TOO_MANY_OPS, NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERRUnsafe_COMPOUND,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE</td>
</tr>
<tr>
<td>BIND_CONN_TO_SESSION</td>
<td>NFS4ERR_BADSESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_CONN_BINDING_NOT_ENFORCED</td>
</tr>
<tr>
<td>CLOSE</td>
<td>NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADHANDLE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BAD_STATEID, NFS4ERR_BADXDR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_DELAY, NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_INVAL, NFS4ERR_ISDIR, NFS4ERR_LEASEMOVED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_LOCKS_HELD, NFS4ERR_MOVED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_NOFILEHANDLE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERRUnsafe_COMPOUND,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE</td>
</tr>
<tr>
<td>COMMIT</td>
<td>NFS4ERR_ACCESS, NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERRUnsafe_COMPOUND,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE</td>
</tr>
<tr>
<td>CREATE</td>
<td>NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADNAME, NFS4ERR_BADOWNER,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADTYPE, NFS4ERR_BADXDR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_DELAY, NFS4ERR_DQUOT,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_EXIST, NFS4ERR_FHEXPIRED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_NAME_TOO_LONG, NFS4ERR_NOFILEHANDLE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_NOSPC, NFS4ERR_NOTDIR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_PERM,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_REQ_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_UNSAME_COMPOUND, NFS4ERR_ROFS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE</td>
</tr>
<tr>
<td>EXCHANGE_ID</td>
<td></td>
</tr>
<tr>
<td>CREATE_SESSION</td>
<td>NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_STALE_CLIENTID</td>
</tr>
<tr>
<td>DELEGPURGE</td>
<td>NFS4ERR_BADXDR, NFS4ERR_NOTSUPP,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_LEASEMOVED, NFS4ERR_MOVED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_REQ_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_UNSAME_COMPOUND,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_STALE_CLIENTID</td>
</tr>
<tr>
<td>DELEGRETURN</td>
<td>NFS4ERR_ADMIN_REVOKED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BAD_STATEID, NFS4ERR_BADXDR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_EXPIRED, NFS4ERR_INVAL,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_LEASEMOVED, NFS4ERR_MOVED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTSUPP,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_REQ_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_UNSAME_COMPOUND,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_STALE_CLIENTID</td>
</tr>
<tr>
<td>DESTROY_CLIENTID</td>
<td>NFS4ERR_CLIENTID_BUSY,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_STALE_CLIENTID</td>
</tr>
<tr>
<td>DESTROY_SESSION</td>
<td>NFS4ERR_BACK_CHAN_BUSY,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADSESSION, NFS4ERR_STALE_CLIENTID</td>
</tr>
</tbody>
</table>
| GET_DIR_DELEGATION          | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,  
|                           | NFS4ERR_BADXDR, NFS4ERR_FHEXPIRED, 
|                           | NFS4ERR_INVAL, NFS4ERR_MOVED, 
|                           | NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,  
|                           | NFS4ERR_OP_NOT_IN_SESSION,  
|                           | NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS, 
|                           | NFS4ERR_REP_TOO_BIG, 
|                           | NFS4ERR_REP_TOO_BIG_TO_CACHE, 
|                           | NFS4ERR_UNSAFE_COMPOUND,  
|                           | NFS4ERR_SERVERFAULT, NFS4ERR_STALE, 
|                           | NFS4ERR_WRONGSEC, NFS4ERR_EIO, 
|                           | NFS4ERR_NOTSUPP  
| GETATTR                   | NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,  
|                           | NFS4ERR_BADXDR, NFS4ERR_DELAY, 
|                           | NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, 
|                           | NFS4ERR_IO, NFS4ERR_MOVED, 
|                           | NFS4ERR_NOFILEHANDLE, 
|                           | NFS4ERR_OP_NOT_IN_SESSION, 
|                           | NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS, 
|                           | NFS4ERR_REP_TOO_BIG, 
|                           | NFS4ERR_REP_TOO_BIG_TO_CACHE, 
|                           | NFS4ERR_UNSAFE_COMPOUND,  
|                           | NFS4ERR_SERVERFAULT, NFS4ERR_STALE 
| GETDEVICEINFO             | NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,  
|                           | NFS4ERR_TOOSMALL, 
|                           | NFS4ERR_UNKNOWN_LAYOUTTYPE 
| GETDEVICELIST             | NFS4ERR_BAD_COOKIE, NFS4ERR_FHEXPIRED,  
|                           | NFS4ERR_INVAL, NFS4ERR_TOOSMALL,  
|                           | NFS4ERR_UNKNOWN_LAYOUTTYPE 
| GETFH                     | NFS4ERR_BADHANDLE, NFS4ERR_FHEXPIRED,  
|                           | NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, 
|                           | NFS4ERR_OP_NOT_IN_SESSION, 
|                           | NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS, 
|                           | NFS4ERR_REP_TOO_BIG, 
|                           | NFS4ERR_REP_TOO_BIG_TO_CACHE, 
|                           | NFS4ERR_UNSAFE_COMPOUND,  
|                           | NFS4ERR_SERVERFAULT, NFS4ERR_STALE 
| ILLEGAL                   | NFS4ERR_OP_ILLEGAL 
| LAYOUTCOMMIT              | NFS4ERR_BADLAYOUT, NFS4ERR_BADIO MODE,  
|                           | NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,  
|                           | NFS4ERR_NOFILEHANDLE, NFS4ERR_NO_GRACE, 
|                           | NFS4ERR_RECLAIM_BAD, NFS4ERR_STALE,  
|                           | NFS4ERR_STALE_CLIENTID, 
|                           | NFS4ERR_UNKNOWN_LAYOUTTYPE |
| LAYOUTGET                          | NFS4ERR_BADLAYOUT, NFS4ERR_BADIOMODE,   |
|                                  | NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,       |
|                                  | NFS4ERR_INVAL, NFS4ERR_LAYOUTUNAVAILABLE,|
|                                  | NFS4ERRayouttrylater, NFS4ERR_LOCKED,   |
|                                  | NFS4ERR_NofileHANDLE, NFS4ERR_NOTSUPP,  |
|                                  | NFS4ERR_RECALLCONFLICT, NFS4ERR_STALE,  |
|                                  | NFS4ERR_STALE_CLIENTID, NFS4ERR_TOOSMALL,|
|                                  | NFS4ERR_UNKNOWN_LAYOUTTYPE              |
| LAYOUTRETURN                      | NFS4ERR_BADLAYOUT, NFS4ERR_BADIOMODE,   |
|                                  | NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,       |
|                                  | NFS4ERR_NofileHANDLE, NFS4ERR_NO_GRACE, |
|                                  | NFS4ERR_STALE, NFS4ERR_STALE_CLIENTID,  |
|                                  | NFS4ERR_UNKNOWN_LAYOUTTYPE              |
| 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_OP_NOT_IN_SESSION,              |
|                                  | NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS,|
|                                  | NFS4ERR_REP_TOO_BIG,                    |
|                                  | NFS4ERR_REP_TOO_BIG_TO_CACHE,           |
|                                  | NFS4ERR_UNSAFE_COMPOUND, NFS4ERR_ROFS,  |
|                                  | NFS4ERR_SERVERFAULT, NFS4ERR_STALE,     |
|                                  | NFS4ERR_WRONGSEC, NFS4ERR_XDEV          |
LOCK

<table>
<thead>
<tr>
<th>Error Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,</td>
</tr>
<tr>
<td>NFS4ERR_BADHANDLE, NFS4ERR_BAD_RANGE,</td>
</tr>
<tr>
<td>NFS4ERR_BAD_STATEID, NFS4ERR_BADXDR,</td>
</tr>
<tr>
<td>NFS4ERR_DEADLOCK, NFS4ERR_DELAY,</td>
</tr>
<tr>
<td>NFS4ERR_DENIED, NFS4ERR_EXPIRED,</td>
</tr>
<tr>
<td>NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,</td>
</tr>
<tr>
<td>NFS4ERR_INVAL, NFS4ERR_ISDIR,</td>
</tr>
<tr>
<td>NFS4ERR_LEASE_MOVED, NFS4ERR_LOCK_NOTSUPP,</td>
</tr>
<tr>
<td>NFS4ERR_LOCK_RANGE, NFS4ERRMOVED,</td>
</tr>
<tr>
<td>NFS4ERR_NOFILEHANDLE, NFS4ERR_NO_GRACE,</td>
</tr>
<tr>
<td>NFS4ERR_OPENMODE,</td>
</tr>
<tr>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td>NFS4ERR_RECLAIM_BAD,</td>
</tr>
<tr>
<td>NFS4ERR_RECLAIMCONFLICT,</td>
</tr>
<tr>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td>NFS4ERR_REQ_TOO_BIG,</td>
</tr>
<tr>
<td>NFS4ERR_REQ_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td>NFS4ERR_UNSAFE_COMPOUND,</td>
</tr>
<tr>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE,</td>
</tr>
<tr>
<td>NFS4ERR_STALE_CLIENTID,</td>
</tr>
<tr>
<td>NFS4ERR_STALE_STATEID</td>
</tr>
</tbody>
</table>

LOCKT

<table>
<thead>
<tr>
<th>Error Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFS4ERR_ACCESS, NFS4ERR_BADHANDLE,</td>
</tr>
<tr>
<td>NFS4ERR_BAD_RANGE, NFS4ERR_BADXDR,</td>
</tr>
<tr>
<td>NFS4ERR_DELAY, NFS4ERR_DENIED,</td>
</tr>
<tr>
<td>NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,</td>
</tr>
<tr>
<td>NFS4ERR_INVAL, NFS4ERR_ISDIR,</td>
</tr>
<tr>
<td>NFS4ERR_LEASE_MOVED, NFS4ERR_LOCK_RANGE,</td>
</tr>
<tr>
<td>NFS4ERRMOVED, NFS4ERR_NOFILEHANDLE,</td>
</tr>
<tr>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td>NFS4ERR_REQ_TOO_BIG,</td>
</tr>
<tr>
<td>NFS4ERR_REQ_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td>NFS4ERR_UNSAFE_COMPOUND,</td>
</tr>
<tr>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE,</td>
</tr>
<tr>
<td>NFS4ERR_STALE_CLIENTID</td>
</tr>
<tr>
<td>LOCKU</td>
</tr>
<tr>
<td>LOOKUP</td>
</tr>
<tr>
<td>LOOKUPP</td>
</tr>
<tr>
<td>NVERIFY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>OPEN</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>OPEN_DOWNGRADE</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>OPENATTR</td>
</tr>
<tr>
<td>PUTFH</td>
</tr>
<tr>
<td>PUTPUBFH</td>
</tr>
<tr>
<td>PUTROOTFH</td>
</tr>
<tr>
<td>READ</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>READ</td>
</tr>
<tr>
<td>READDIR</td>
</tr>
<tr>
<td>READLINK</td>
</tr>
<tr>
<td>RECLAIM_COMPLETE</td>
</tr>
</tbody>
</table>

<p>| ERRORS |
|========|
| NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, |
| NFS4ERR_BADHANDLE, NFS4ERR_BAD_STATEID, |
| NFS4ERR_BADXDR, NFS4ERR_DELAY, |
| NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED, |
| NFS4ERR_GRACE, NFS4ERR_IO, NFS4ERR_INVAL, |
| NFS4ERR_ISDIR, NFS4ERR_LEASEMOVED, |
| NFS4ERR_LOCKED, NFS4ERR_MOVED, |
| NFS4ERR_NOFILEHANDLE, NFS4ERR_NXIO, |
| NFS4ERR_OP_NOT_IN_SESSION, |
| NFS4ERR_OPENMODE, NFS4ERR_PNFS_IO_HOLE, |
| NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS, |
| NFS4ERR_REP_TOO_BIG, |
| NFS4ERR_REP_TOO_BIG_TO_CACHE, |
| NFS4ERR_UNSAFE_COMPOUND, |
| NFS4ERR_SERVERFAULT, NFS4ERR_STALE, |
| NFS4ERR_STALE_STATEID |
| NFS4ERR_ACCESS, NFS4ERR_BADHANDLE, |
| NFS4ERR_BAD_COOKIE, NFS4ERR_BADXDR, |
| NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, |
| NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, |
| NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR, |
| NFS4ERR_NOTSAME, |
| NFS4ERR_OP_NOT_IN_SESSION, |
| NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS, |
| NFS4ERR_REP_TOO_BIG, |
| NFS4ERR_REP_TOO_BIG_TO_CACHE, |
| NFS4ERR_UNSAFE_COMPOUND, |
| NFS4ERR_SERVERFAULT, NFS4ERR_STALE, |
| NFS4ERR_TOSMALL |
| NFS4ERR_ACCESS, NFS4ERR_BADHANDLE, |
| NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, |
| NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR, |
| NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, |
| NFS4ERR_NOTSUPP, |
| NFS4ERR_OP_NOT_IN_SESSION, |
| NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS, |
| NFS4ERR_REP_TOO_BIG, |
| NFS4ERR_REP_TOO_BIG_TO_CACHE, |
| NFS4ERR_UNSAFE_COMPOUND, |
| NFS4ERR_SERVERFAULT, NFS4ERR_STALE |
| NFS4ERR_COMPLETE_ALREADY |</p>
<table>
<thead>
<tr>
<th>RELEASE_LOCKOWNER</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NF4ERR_ADMIN_REVOKED, NF4ERR_BADXDR,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_EXPIRED, NF4ERRLEASEMOVED,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_LOCKS_HELD,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_OP_NOT_IN_SESSION,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REQ_TOO_BIG, NF4ERR_TOO_MANY_OPS,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG_TO_CACHE,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_UNSAFE_COMPOUND,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_SERVERFAULT,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_STALE_CLIENTID</td>
<td></td>
</tr>
<tr>
<td>REMOVE</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_ACCESS, NF4ERR_BADCHAR,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_BADHANDLE, NF4ERR_BADNAME,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_BADXDR, NF4ERR_DELAY,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_FHEXPIRED, NF4ERR_FILE_OPEN,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_INVAL, NF4ERR_IO, NF4ERRMOVED,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_NAMEETOOLONG, NF4ERR_NOENT,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_NOPFILEHANDLE, NF4ERR_NODIR,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_NOTEMPTY,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_OP_NOT_IN_SESSION,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REQ_TOO_BIG, NF4ERR_TOO_MANY_OPS,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG_TO_CACHE,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_UNSAFE_COMPOUND, NF4ERR_ROFS,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_SERVERFAULT, NF4ERR_STALE</td>
<td></td>
</tr>
<tr>
<td>RENAME</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_ACCESS, NF4ERR_BADCHAR,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_BADHANDLE, NF4ERR_BADNAME,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_BADXDR, NF4ERR_DELAY,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_DQUOT, NF4ERR_EXIST,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_FHEXPIRED, NF4ERR_FILE_OPEN,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_INVAL, NF4ERR_IO, NF4ERRMOVED,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_NAMEETOOLONG, NF4ERR_NOENT,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_NOPFILEHANDLE, NF4ERR_NOSPC,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_NODIR, NF4ERR_NOTEMPTY,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_OP_NOT_IN_SESSION,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REQ_TOO_BIG, NF4ERR_TOO_MANY_OPS,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG_TO_CACHE,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_SERVERFAULT, NF4ERR_STALE,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_WRONGSEC, NF4ERR_XDEV</td>
<td></td>
</tr>
<tr>
<td>RESTOREFH</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_BADHANDLE, NF4ERR_FHEXPIRED,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_MOVED, NF4ERR_OP_NOT_IN_SESSION,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REQ_TOO_BIG, NF4ERR_TOO_MANY_OPS,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_REP_TOO_BIG_TO_CACHE,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_SERVERFAULT, NF4ERR_STALE,</td>
<td></td>
</tr>
<tr>
<td>NF4ERR_WRONGSEC, NF4ERR_XDEV</td>
<td></td>
</tr>
<tr>
<td>SAVEFH</td>
<td>NFSerr_BADHANDLE, NFSerr_FHexpired, NFSerr_MOVED, NFSerr_NOFILEHANDLE, NFSerr_OP_NOT_IN_SESSION, NFSerr_REQ_TOO_BIG, NFSerr_REQ_TOO_MANY_OPS, NFSerr_REP_TOO_BIG, NFSerr_REP_TOO_BIG_TO_CACHE, NFSerrUnsafe_COMPOUND, NFSerr_SERVERFAULT, NFSerr_STALE</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SECINFO</td>
<td>NFSerr_ACCESS, NFSerr_BadChar, NFSerr_BADHANDLE, NFSerr_BADNAME, NFSerr_BADXDR, NFSerr_FHexpired, NFSerrINVAL, NFSerr_MOVED, NFSerr_NAMETOOLONG, NFSerr_NONENT, NFSerr_NOFILEHANDLE, NFSerr_NOTDIR, NFSerr_OP_NOT_IN_SESSION, NFSerr_REQ_TOO_BIG, NFSerr_REQ_TOO_MANY_OPS, NFSerr_REP_TOO_BIG, NFSerr_REP_TOO_BIG_TO_CACHE, NFSerrUnsafe_COMPOUND, NFSerr_SERVERFAULT, NFSerr_STALE</td>
</tr>
<tr>
<td>SECINFO_NO_NAME</td>
<td>NFSerr_ACCESS, NFSerr_BADHANDLE, NFSerr_BADNAME, NFSerr_BadXDR, NFSerr_FHexpired, NFSerrINVAL, NFSerr_MOVED, NFSerr_NAMETOOLONG, NFSerr_NONENT, NFSerr_NOFILEHANDLE, NFSerr_NOTDIR, NFSerr_OP_NOT_IN_SESSION, NFSerr_REQ_TOO_BIG, NFSerr_REQ_TOO_MANY_OPS, NFSerr_REP_TOO_BIG, NFSerr_REP_TOO_BIG_TO_CACHE, NFSerrUnsafe_COMPOUND, NFSerr_SERVERFAULT, NFSerr_STALE</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>NFSerr_BADSESSION, NFSerr_Badslot, NFSerr_CONN_NOT_BOUND_TO_SESSION, NFSerr_SEQ_Misordered, NFSerr_SEQUENCE_POS, NFSerr_REQ_TOO_BIG, NFSerr_TOO_MANY_OPS, NFSerr_REP_TOO_BIG, NFSerr_REP_TOO_BIG_TO_CACHE</td>
</tr>
<tr>
<td>SET_SSV</td>
<td>NFSerr_Bad_Session_Digest, NFSerr_CONN_BINDING_NOT_ENFORCED</td>
</tr>
<tr>
<td>SETATTR</td>
<td>NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADHANDLE, NFS4ERR_BADOWNER,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BAD_STATEID, NFS4ERR_BADXDR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_DELAY, NFS4ERR_DQUOT,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_EXPIRED, NFS4ERR_FBIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_LOCKED, NFS4ERR_MOVED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OPENMODE, NFS4ERR_PERM,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_UNSAFE_COMPOUND, NFS4ERR_ROFS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_STALE_STATEID</td>
</tr>
<tr>
<td>EXCHANGE_ID</td>
<td>NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_INVAL, NFS4ERR_SERVERFAULT</td>
</tr>
<tr>
<td>CREATE_SESSION</td>
<td>NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_DELAY, NFS4ERR_SERVERFAULT,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_STALE_CLIENTID</td>
</tr>
<tr>
<td>VERIFY</td>
<td>NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADCHAR, NFS4ERR_BADHANDLE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADXDR, NFS4ERR_DELAY,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_NOT_SAME,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_UNSAFE_COMPOUND,</td>
</tr>
<tr>
<td>WANT_DELEGATION</td>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE</td>
</tr>
<tr>
<td>WRITE</td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_BADHANDLE, NFS4ERR_BAD_STATEID,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_BADXDR, NFS4ERR_DELAY,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_DQUOT, NFS4ERR_EXPIRED,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_FBIG, NFS4ERR_FHEXPIRED,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_ISDIR, NFS4ERR_LEASEMOVED,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_LOCKED, NFS4ERR_MOVED,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_NFILEHANDLE, NFS4ERR_NOSPC,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_NXIO, NFS4ERR_OPNOTINSESSION,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_OPENMODE, NFS4ERR_PNFSIOHOLE,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_REQTOOBIG, NFS4ERR_TOOMANYOPS,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_REPTOOBIG,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_REPTOOBIGTOCACHE,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_UNSAFE_COMPOUND, NFS4ERR_ROFS,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_SERVERFAULT, NFS4ERR_STALE,</td>
<td></td>
</tr>
<tr>
<td>NFS4ERR_STALESTATEID</td>
<td></td>
</tr>
</tbody>
</table>

Table 9
### 15.3. Callback operations and their valid errors

Mappings of valid error returns for each protocol callback operation

<table>
<thead>
<tr>
<th>Callback Operation</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB_GETATTR</td>
<td>NFS4ERR_BADHANDLE NFS4ERR_BADXDR</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_UNSAFE_COMPOUND,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT</td>
</tr>
<tr>
<td>CB_ILLEGAL</td>
<td>NFS4ERR_OP_ILLEGAL</td>
</tr>
<tr>
<td>CB_LAYOUTRECALL</td>
<td>NFS4ERR_NOMATCHING_LAYOUT</td>
</tr>
<tr>
<td>CB_NOTIFY</td>
<td>NFS4ERR_BAD_STATEID NFS4ERR_INVAL</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADXDR NFS4ERR_SERVERFAULT</td>
</tr>
<tr>
<td>CB_PUSH_DELEG</td>
<td></td>
</tr>
<tr>
<td>CB_RECALL</td>
<td>NFS4ERR_BADHANDLE NFS4ERR_BAD_STATEID</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_BADXDR</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SERVERFAULT</td>
</tr>
<tr>
<td>CB_RECALL_ANY</td>
<td>NFS4ERR_OP_NOT_IN_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_INVAL</td>
</tr>
<tr>
<td>CB_RECALLABLE_OBJ_AVAIL</td>
<td></td>
</tr>
<tr>
<td>CB_RECALL_CREDIT</td>
<td></td>
</tr>
<tr>
<td>CB_SEQUENCE</td>
<td>NFS4ERR_BADSESSION, NFS4ERR_BADSLOT,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_CONN_NOT_BOUND_TO_SESSION,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SEQ_MISORDERED,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_SEQUENCE_POS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REQ_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_TOO_MANY_OPS,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG,</td>
</tr>
<tr>
<td></td>
<td>NFS4ERR_REP_TOO_BIG_TO_CACHE,</td>
</tr>
</tbody>
</table>

Table 10
### 15.4. Errors and the operations that use them

<table>
<thead>
<tr>
<th>Error</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFS4ERR_ACCESS</td>
<td>ACCESS, COMMIT, CREATE, GETATTR, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, READ, READDIR, READLINK, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_ADMIN_REVOKED</td>
<td>CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, OPEN_DOWNGRADE, READ, RELEASE_LOCKOWNER, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_ATTRNOTSUPP</td>
<td>CREATE, NVERIFY, OPEN, SETATTR, VERIFY</td>
</tr>
<tr>
<td>NFS4ERR_BACK_CHAN_BUSY</td>
<td>DESTROY_SESSION</td>
</tr>
<tr>
<td>NFS4ERR_BADCHAR</td>
<td>CREATE, LINK, LOOKUP, NVERIFY, OPEN, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY</td>
</tr>
<tr>
<td>NFS4ERR_BADHANDLE</td>
<td>ACCESS, CB_GETATTR, CB_RECALL, CLOSE, COMMIT, CREATE, GETATTR, GETFH, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_BADIOMODE</td>
<td>LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN</td>
</tr>
<tr>
<td>NFS4ERR_BADLAYOUT</td>
<td>LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN</td>
</tr>
<tr>
<td>NFS4ERR_BADNAME</td>
<td>CREATE, LINK, LOOKUP, OPEN, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME</td>
</tr>
<tr>
<td>NFS4ERR_BADOWNER</td>
<td>CREATE, OPEN, SETATTR</td>
</tr>
<tr>
<td>Error Code</td>
<td>Relevant Operations</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>NFS4ERR_BADSESSION</td>
<td>BIND_CONN_TO_SESSION, CB_SEQUENCE, DESTROY_SESSION, SEQUENCE</td>
</tr>
<tr>
<td>NFS4ERR_BAD_SLOT</td>
<td>CB_SEQUENCE, SEQUENCE</td>
</tr>
<tr>
<td>NFS4ERR_BAD_TYPE</td>
<td>CREATE</td>
</tr>
<tr>
<td>NFS4ERR_BADXDR</td>
<td>ACCESS, CB_GETATTR, CB_NOTIFY, CB_RECALL, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGPURGE, DELEGRETURN, EXCHANGE_ID, GETATTR, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, RELEASE_LOCKOWNER, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_BAD_COOKIE</td>
<td>GETDEVICELIST, READDIR</td>
</tr>
<tr>
<td>NFS4ERR_BAD_RANGE</td>
<td>CB_SEQUENCE, SET_SSV</td>
</tr>
<tr>
<td>NFS4ERR_BAD_SESSION_DIGEST</td>
<td>BIND_CONN_TO_SESSION, SET_SSV</td>
</tr>
<tr>
<td>NFS4ERR_BAD_STATEID</td>
<td>CB_NOTIFY, CB_RECALL, CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN_DOWNGRADE, READ, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_CLID_INUSE</td>
<td>CREATE_SESSION, EXCHANGE_ID</td>
</tr>
<tr>
<td>NFS4ERR_CLIENTID_BUSY</td>
<td>DESTROY_CLIENTID</td>
</tr>
<tr>
<td>NFS4ERR_COMPLETE_ALREADY</td>
<td>RETCLAIM_COMPLETE</td>
</tr>
<tr>
<td>NFS4ERR_CONN_BINDING_NOT_ENFORCED</td>
<td>BIND_CONN_TO_SESSION, SET_SSV</td>
</tr>
<tr>
<td>NFS4ERR_CONN_NOT_BOUND_TO_SESSION</td>
<td>CB_SEQUENCE, SEQUENCE</td>
</tr>
<tr>
<td>NFS4ERR_DEADLOCK</td>
<td>ACCESS, CLOSE, CREATE, CREATE_SESSION, GETATTR, LINK, LOCK, LOCKT, NVERIFY, Open, OPENATTR, READ, READDIR, READLINK, REMOVE, RENAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_DENIED</td>
<td>LOCK, LOCKT</td>
</tr>
<tr>
<td>NFS4ERR_DQUOT</td>
<td>CREATE, LINK, OPEN, OPENATTR, RENAME, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_EIO</td>
<td>GET_DIR_DELEGATION</td>
</tr>
<tr>
<td>NFS4ERR_EXIST</td>
<td>CREATE, LINK, OPEN, RENAME</td>
</tr>
<tr>
<td>NFS4ERR_EXPIRED</td>
<td>CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN, OPEN_DOWNGRADE, READ, RELEASE_LOCKOWNER, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_FBIG</td>
<td>SETATTR, WRITE</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>NFS4ERR_FHEXPIRED</td>
<td>ACCESS, CLOSE, COMMIT, CREATE, GETATTR, GETDEVICEINFO, GETDEVICELIST, GETFH, GET_DIR_DElegation, LAYOUTcommit, LAYOUTget, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_no_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_FILE_OPEN</td>
<td>LINK, REMOVE, RENAME</td>
</tr>
<tr>
<td>NFS4ERR_GRACE</td>
<td>LAYOUTGET, LOCK, LOCKT, LOCKU, OPEN, READ, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_INVAl</td>
<td>ACCESS, CB_NOTIFY, CB_RECALL_ANY, CLOSE, COMMIT, CREATE, DELEGRETURN, EXCHANGE_ID, GETATTR, GETDEVICEINFO, GETDEVICELIST, GET_DIR_DElegation, LAYOUTcommit, LAYOUTget, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, NVERIFY, OPEN, OPEN_DOWNGRADE, READ, READDIR, READLINK, REMOVE, RENAME, SECINFO, SECINFO_no_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_IO</td>
<td>ACCESS, COMMIT, CREATE, GETATTR, LINK, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, READ, READDIR, READLINK, REMOVE, RENAME, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_ISDIR</td>
<td>CLOSE, COMMIT, LINK, LOCK, LOCKT, LOCKU, OPEN, READ, READLINK, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_LAYOUTTRYLATER</td>
<td>LAYOUTGET</td>
</tr>
<tr>
<td>NFS4ERR_LAYOUTUNAVAILABLE</td>
<td>LAYOUTGET</td>
</tr>
<tr>
<td>Code</td>
<td>Operations</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>NFS4ERRLEASEMOVED</td>
<td>CLOSE, DELEGPURGE, DELEGRETURN, LOCK, LOCKT, LOCKU, OPEN, READ, RELEASELOCKOWNER, WRITE</td>
</tr>
<tr>
<td>NFS4ERRLOCKED</td>
<td>LAYOUTGET, READ, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERRLOCKSHELD</td>
<td>CLOSE, RELEASELOCKOWNER</td>
</tr>
<tr>
<td>NFS4ERRLOCKNOTSUPP</td>
<td>LOCK</td>
</tr>
<tr>
<td>NFS4ERRLOCKRANGE</td>
<td>LOCK, LOCKT, LOCKU</td>
</tr>
<tr>
<td>NFS4ERRMLINK</td>
<td>LINK</td>
</tr>
<tr>
<td>NFS4ERRMOVED</td>
<td>ACCESS, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, GETFH, GET_DIRDELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERRNAMETOOLONG</td>
<td>CREATE, LINK, LOOKUP, OPEN, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME</td>
</tr>
<tr>
<td>NFS4ERRNOENT</td>
<td>LINK, LOOKUP, LOOKUPP, OPEN, OPENATTR, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME</td>
</tr>
<tr>
<td>NFS4ERRNOFILEHANDLE</td>
<td>ACCESS, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, GETFH, GET_DIRDELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, READ, READDIR, READLINK, REMOVE, RENAME, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERRNOMATCHING_LAYOUT</td>
<td>CB_LAYOUTRECALL</td>
</tr>
<tr>
<td>NFS4ERRNOSPC</td>
<td>CREATE, LINK, OPEN, OPENATTR, RENAME, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERRNOTDIR</td>
<td>CREATE, GET_DIRDELEGATION, LINK, LOOKUP, LOOKUPP, OPEN, REaddir, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME</td>
</tr>
<tr>
<td>NFS4ERRNOTEMPTY</td>
<td>REMOVE, RENAME</td>
</tr>
<tr>
<td>Error Code</td>
<td>Operations</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>NFS4ERR_NOTSUPP</td>
<td>DELEGPURGE, DELEGRETURN, GET_DIR_DELEGATION, LAYOUTGET, LINK, OPENATTR, READLINK</td>
</tr>
<tr>
<td>NFS4ERR_NOT_SAME</td>
<td>READDIR, VERIFY LAYOUTCOMMIT, LAYOUTRETURN, LOCK, OPEN</td>
</tr>
<tr>
<td>NFS4ERR_NO_GRACE</td>
<td>READ, WRITE LOCK, READ, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_NXIO</td>
<td>CB_ILLEGAL, ILLEGAL</td>
</tr>
<tr>
<td>NFS4ERR_OPENMODE</td>
<td>ACCESS, CB_GETATTR, CB_RECALL, CB_RECALL_ANY, CLOSE, COMMIT, CREATE, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_OP_ILLEGAL</td>
<td>CREATE, OPEN, SETATTR</td>
</tr>
<tr>
<td>NFS4ERR_OP_NOT_IN_SESSION</td>
<td>READ, WRITE LAYOUTGET</td>
</tr>
<tr>
<td>NFS4ERR_PERM</td>
<td>LAYOUTCOMMIT, LOCK, OPEN ACCESS, CB_GETATTR, CB_RECALL, CB_RECALL_ANY, CB_SEQUENCE, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, GETFH, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOCKUP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RELEASE_LOCKOWNER, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_PNFS_IO_HOLE</td>
<td>READ, WRITE LAYOUTGET</td>
</tr>
<tr>
<td>NFS4ERR_RECALLCONFLICT</td>
<td>LAYOUTCOMMIT, LOCK, OPEN ACCESS, CB_GETATTR, CB_RECALL, CB_RECALL_ANY, CB_SEQUENCE, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, GETFH, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOCKUP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RELEASE_LOCKOWNER, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SEQUENCE, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_RECLAIM_BAD</td>
<td>CB_ILLEGAL, ILLEGAL</td>
</tr>
<tr>
<td>NFS4ERR_RECLAIM_CONFLICT</td>
<td>ACCESS, CB_GETATTR, CB_RECALL, CB_RECALL_ANY, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, GETFH, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOCKUP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RELEASE_LOCKOWNER, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SEQUENCE, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_REP_TOO_BIG</td>
<td>CB_ILLEGAL, ILLEGAL</td>
</tr>
<tr>
<td>Error Code</td>
<td>Supported Operations</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td><strong>NFS4ERR_REP_TOO_BIG_TO_CACHE</strong></td>
<td>ACCESS, CB_GETATTR,</td>
</tr>
<tr>
<td></td>
<td>CB_RECALL, CB_RECALL_ANY,</td>
</tr>
<tr>
<td></td>
<td>CB_SEQUENCE, CLOSE, COMMIT,</td>
</tr>
<tr>
<td></td>
<td>CREATE, DELEGPURGE,</td>
</tr>
<tr>
<td></td>
<td>DELEGRETURN, GETATTR, GETFH,</td>
</tr>
<tr>
<td></td>
<td>GET_DIR_DELEGATION, LINK,</td>
</tr>
<tr>
<td></td>
<td>LOCK, LOCKT, LOCKU, LOOKUP,</td>
</tr>
<tr>
<td></td>
<td>LOOKUPP, NVERIFY, OPEN,</td>
</tr>
<tr>
<td></td>
<td>OPENATTR, OPEN_DOWNGRADE,</td>
</tr>
<tr>
<td></td>
<td>PUTFH, PUTPUBFH, PUTROOTFH,</td>
</tr>
<tr>
<td></td>
<td>READ, READDIR, READLINK,</td>
</tr>
<tr>
<td></td>
<td>RELEASE_LOCKOWNER, REMOVE,</td>
</tr>
<tr>
<td></td>
<td>RENAME, RESTOREFH, SAVEFH,</td>
</tr>
<tr>
<td></td>
<td>SECINFO, SECINFO_NO_NAME,</td>
</tr>
<tr>
<td></td>
<td>SEQUENCE, SETATTR, VERIFY,</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
</tr>
<tr>
<td><strong>NFS4ERR_REQ_TOO_BIG</strong></td>
<td>ACCESS, CB_GETATTR,</td>
</tr>
<tr>
<td></td>
<td>CB_RECALL, CB_RECALL_ANY,</td>
</tr>
<tr>
<td></td>
<td>CB_SEQUENCE, CLOSE, COMMIT,</td>
</tr>
<tr>
<td></td>
<td>CREATE, DELEGPURGE,</td>
</tr>
<tr>
<td></td>
<td>DELEGRETURN, GETATTR, GETFH,</td>
</tr>
<tr>
<td></td>
<td>GET_DIR_DELEGATION, LINK,</td>
</tr>
<tr>
<td></td>
<td>LOCK, LOCKT, LOCKU, LOOKUP,</td>
</tr>
<tr>
<td></td>
<td>LOOKUPP, NVERIFY, OPEN,</td>
</tr>
<tr>
<td></td>
<td>OPENATTR, OPEN_DOWNGRADE,</td>
</tr>
<tr>
<td></td>
<td>PUTFH, PUTPUBFH, PUTROOTFH,</td>
</tr>
<tr>
<td></td>
<td>READ, READDIR, READLINK,</td>
</tr>
<tr>
<td></td>
<td>RELEASE_LOCKOWNER, REMOVE,</td>
</tr>
<tr>
<td></td>
<td>RENAME, RESTOREFH, SAVEFH,</td>
</tr>
<tr>
<td></td>
<td>SECINFO, SECINFO_NO_NAME,</td>
</tr>
<tr>
<td></td>
<td>SEQUENCE, SETATTR, VERIFY,</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
</tr>
<tr>
<td><strong>NFS4ERR_RESTOREFH</strong></td>
<td>RESTOREFH</td>
</tr>
<tr>
<td><strong>NFS4ERR_ROFS</strong></td>
<td>COMMIT, CREATE, LINK, OPEN,</td>
</tr>
<tr>
<td></td>
<td>OPENATTR, REMOVE, RENAME,</td>
</tr>
<tr>
<td></td>
<td>SETATTR, WRITE</td>
</tr>
<tr>
<td><strong>NFS4ERR_SAME</strong></td>
<td>NVERIFY</td>
</tr>
<tr>
<td><strong>NFS4ERR_SEQUENCE_POS</strong></td>
<td>CB_SEQUENCE, SEQUENCE</td>
</tr>
<tr>
<td><strong>NFS4ERR_SEQ_MISORDERED</strong></td>
<td>CB_SEQUENCE, SEQUENCE</td>
</tr>
<tr>
<td>Error Code</td>
<td>Operations</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>NFS4ERR_SERVERFAULT</td>
<td>ACCESS, CB_GETATTR, CB_NOTIFY, CB_RECALL, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGRETURN, DELEGRETURN, EXCHANGE_ID, GETATTR, GETFH, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTFUBFH, PUTROOTFH, READ, READDIR, READLINK, RELEASE_LOCKOWNER, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_SHARE_DENIED</td>
<td>OPEN</td>
</tr>
<tr>
<td>NFS4ERR_STALE</td>
<td>OPEN, REMOVE, RENAME, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_STALE_CLIENTID</td>
<td>CREATE_SESSION, DELEGRETURN, DESTROY_CLIENTID, DESTROY_SESSION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LOCK, LOCKT, LOCKU, LOCKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_STALE_STATEID</td>
<td>CLOSE, DELEGRETURN, LOCK, LOCKU, OPEN_DOWNGRADE, READ, SETATTR, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_SYMLINK</td>
<td>LOOKUP, OPEN</td>
</tr>
<tr>
<td>NFS4ERR_TOOSMALL</td>
<td>GETDEVICEINFO, GETDEVICELIST, LAYOUTGET, READDIR</td>
</tr>
</tbody>
</table>
### Table 11

<table>
<thead>
<tr>
<th>Code</th>
<th>Procedural Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFS4ERR_TOO_MANY_OPS</td>
<td>ACCESS, CB_GETATTR, CB_RECALL, CB_RECALL_ANY, CB_SEQUENCE, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, GETFH, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RELEASE_LOCKOWNER, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SEQUENCE, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_UNKNOWN_LAYOUTTYPE</td>
<td>GETDEVICEINFO, GETDEVICELIST, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN</td>
</tr>
<tr>
<td>NFS4ERR_UNSAFE_COMPOUND</td>
<td>ACCESS, CB_GETATTR, CLOSE, COMMIT, CREATE, DELEGPURGE, DELEGRETURN, GETATTR, GETFH, GET_DIR_DELEGATION, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RELEASE_LOCKOWNER, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE</td>
</tr>
<tr>
<td>NFS4ERR_WRONGSEC</td>
<td>GET_DIR_DELEGATION, LINK, LOOKUP, LOOKUPP, OPEN, PUTFH, PUTPUBFH, PUTROOTFH, RENAME, RESTOREFH</td>
</tr>
<tr>
<td>NFS4ERR_XDEV</td>
<td>LINK, RENAME</td>
</tr>
</tbody>
</table>

#### 16. NFS version 4.1 Procedures

#### 16.1. Procedure 0: NULL - No Operation
16.1.1. SYNOPSIS

16.1.2. ARGUMENTS

void;

16.1.3. RESULTS

void;

16.1.4. 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.

16.1.5. ERRORS

None.

16.2. Procedure 1: COMPOUND - Compound Operations

16.2.1. SYNOPSIS

compoundargs -> compoundres

16.2.2. ARGUMENTS

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

struct COMPOUND4args {
    utf8str_cs        tag;
    uint32_t          minorversion;
    nfs_argop4        argarray<>;
};
16.2.3. RESULTS

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

    struct COMPOUND4res {
        nfsstat4        status;
        utf8str_cs      tag;
        nfs_resop4      resarray<>;
    };

16.2.4. 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. See Section 2.10.4.4 for a more detailed discussion.

The server will generally choose between two methods of decoding the client’s request. The first would be the traditional one pass XDR decode. If there is an XDR decoding error in this case, the 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 the individual operations; the most interesting is the decode of attributes. In this case, the server may encounter an XDR decode error during the second pass. In this case, the server would return the error NFS4ERR_BADXDR to signify the decode error.

The COMPOUND arguments contain a "minorversion" field. For NFSv4.1, the value for this field is 1. If the server receives a COMPOUND procedure with a minorversion field value that it does not support, the server MUST return an error of NFS4ERR_MINOR_VERS_MISMATCH and a zero length resultdata array.
Contained within the COMPOUND results is a "status" field. If the results array length is non-zero, this status must be equivalent to the status of the last operation that was executed within the COMPOUND procedure. Therefore, if an operation incurred an error then the "status" value will be the same error value as is being returned for the operation that failed.

Note that operations, 0 (zero) and 1 (one) are not defined for the COMPOUND procedure. Operation 2 is not defined but reserved for future definition and use with minor versioning. If the server receives a operation array that contains operation 2 and the minorversion field has a value of 0 (zero), an error of NFS4ERR_OP_ILLEGAL, as described in the next paragraph, is returned to the client. If an operation array contains an operation 2 and the minorversion field is non-zero and the server does not support the minor version, the server returns an error of NFS4ERR_MINOR_VERS_MISMATCH. Therefore, the NFS4ERR_MINOR_VERS_MISMATCH error takes precedence over all other errors.

It is possible that the server receives a request that contains an operation that is less than the first legal operation (OP_ACCESS) or greater than the last legal operation (OP_RELEASE_LOCKOWNER). In this case, the server’s response will encode the opcode OP_ILLEGAL rather than the illegal opcode of the request. The status field in the ILLEGAL return results will set to NFS4ERR_OP_ILLEGAL. The COMPOUND procedure’s return results will also be NFS4ERR_OP_ILLEGAL.

The definition of the "tag" in the request is left to the implementor. It may be used to summarize the content of the compound request for the benefit of packet sniffers and engineers debugging implementations. However, the value of "tag" in the response SHOULD be the same value as provided in the request. This applies to the tag field of the CB_COMPOUND procedure as well.

16.2.4.1. Current File Handle and Stateid

The COMPOUND procedure offers a simple environment for the execution of the operations specified by the client. The first two relate to the file handle while the second two relate to the current stateid.

16.2.4.1.1. Current File Handle

The current and saved file handle are used throughout the protocol. Most operations implicitly use the current file handle as a argument and many set the current file handle as part of the results. The combination of client specified sequences of operations and current and saved file handle arguments and results allows for greater
protocol flexibility. The best or easiest example of current file handle 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 75

In this example, the PUTFH operation explicitly sets the current file handle value while the result of each LOOKUP operation sets the current file handle value to the resultant file system object. Also, the client is able to insert GETATTR operations using the current file handle as an argument.

Along with the current file handle, there is a saved file handle. While the current file handle is set as the result of operations like LOOKUP, the saved file handle must be set directly with the use of the SAVEFH operation. The SAVEFH operations copies the current file handle value to the saved value. The saved file handle value is used in combination with the current file handle value for the LINK and RENAME operations. The RESTOREFH operation will copy the saved file handle value to the current file handle value; as a result, the saved file handle value may be used a sort of "scratch" area for the client’s series of operations.

16.2.4.1.2. Current Stateid

With NFSv4.1, additions of a current stateid and a saved stateid have been made to the COMPOUND processing environment; this allows for the passing of stateids between operations. There are no changes to the syntax of the protocol, only changes to the semantics of a few operations.

A "current stateid" is the stateid that is associated with the current file handle. The current stateid may only be changed by an operation that modifies the current file handle or returns a stateid. If an operation returns a stateid it MUST set the current stateid to the returned value. If an operation sets the current file handle but does not return a stateid, the current stateid MUST be set to the all-zeros special stateid. As an example, PUTFH will change the current server state from \{ocfh, osid\} to \{cfh, 0\} while LOCK will
change the current state from \{cfh, osid\} to \{cfh, nsid\}. The \texttt{SAVEFH}
and \texttt{RESTOREFH} operations will save and restore both the file handle
and the stateid as a set.

Any operation which takes as an argument a stateid that is not the
special all-zeros stateid MUST set the current stateid to the all-
zeros value before evaluating the operation. If the argument is the
special all-zeros stateid, the operation is evaluated using the
current stateid.

The following example is the common case of a simple \texttt{READ} operation
with a supplied stateid showing that the \texttt{PUTFH} initializes the
current stateid to zero. The subsequent \texttt{READ} with stateid \texttt{sid1}
replaces the current stateid before evaluating the operation.

\begin{verbatim}
  PUTFH fh1                         - -> \{fh1, 0\}
  READ sid1,0,1024       \{fh1, sid1\} -> \{fh1, sid1\}
\end{verbatim}

\textit{Figure 76}

This next example performs an \texttt{OPEN} with the client provided stateid
\texttt{sid1} and as a result generates stateid \texttt{sid2}. The next operation
specifies the \texttt{READ} with the special all-zero stateid but the current
stateid set by the previous operation is actually used when the
operation is evaluated, allowing correct interaction with any
existing, potentially conflicting, locks.

\begin{verbatim}
  PUTFH fh1                         - -> \{fh1, 0\}
  OPEN R,sid1,"compA"    \{fh1, sid1\} -> \{fh2, sid2\}
  READ 0,0,1024          \{fh2, sid2\} -> \{fh2, sid2\}
  CLOSE 0                \{fh2, sid2\} -> \{fh2, sid3\}
\end{verbatim}

\textit{Figure 77}

The final example is similar to the second in how it passes the
stateid \texttt{sid2} generated by the \texttt{LOCK} operation to the next \texttt{READ}
operation. This allows the client to explicitly surround a single
I/O operation with a lock and its appropriate stateid to guarantee
correctness with other client locks.

\begin{verbatim}
  PUTFH fh1                         - -> \{fh1, 0\}
  LOCK W,0,1024,sid1     \{fh1, sid1\} -> \{fh1, sid2\}
  READ 0,0,1024          \{fh1, sid2\} -> \{fh1, sid2\}
  LOCKU W,0,1024,0       \{fh1, sid2\} -> \{fh1, sid3\}
\end{verbatim}

\textit{Figure 78}
16.2.5.  IMPLEMENTATION

16.2.6.  ERRORS

   All errors defined in the protocol

17.  NFS version 4.1 Operations

17.1.  Operation 3: ACCESS - Check Access Rights

17.1.1.  SYNOPSIS

   (cfh), accessreq -> supported, accessrights

17.1.2.  ARGUMENTS

   /*
   * ACCESS: Check access permission
   */
   const ACCESS4_READ     = 0x00000001;
   const ACCESS4_LOOKUP   = 0x00000002;
   const ACCESS4_MODIFY   = 0x00000004;
   const ACCESS4_EXTEND   = 0x00000008;
   const ACCESS4_DELETE   = 0x00000010;
   const ACCESS4_EXECUTE  = 0x00000020;

   struct ACCESS4args {
      /* CURRENT_FH: object */
      uint32_t access;
   };

17.1.3.  RESULTS

   struct ACCESS4resok {
      uint32_t supported;
      uint32_t access;
   };

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

17.1.4. 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 was originally sent in the arguments. For example, if the client sends an ACCESS operation with only the ACCESS4_READ value set and the server supports this value, the server will return only ACCESS4_READ even if it could have reliably checked other values.

The results of this operation are necessarily advisory in nature. A return status of NFS4_OK and the appropriate bit set in the bit mask does not imply that such access will be allowed to the file system object in the future. This is because access rights can be revoked by the server at any time.

The following access permissions may be requested:

ACCESS4_READ  Read data from file or read a directory.
ACCESS4_LOOKUP Look up a name in a directory (no meaning for non-directory objects).
ACCESS4_MODIFY Rewrite existing file data or modify existing directory entries.
ACCESS4_EXTEND Write new data or add directory entries.
ACCESS4_DELETE Delete an existing directory entry.
ACCESS4_EXECUTE Execute file (no meaning for a directory).

On success, the current filehandle retains its value.

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

In the NFS version 2 protocol, the only reliable way to determine whether an operation was allowed was to try it and see if it succeeded or failed. Using the ACCESS operation in the NFS version 4 protocol, the client can ask the server to indicate whether or not one or more classes of operations are permitted. The ACCESS operation is provided to allow clients to check before doing a series of operations which will result in an access failure. The OPEN operation provides a point where the server can verify access to the file object and method to return that information to the client. The ACCESS operation is still useful for directory operations or for use in the case the UNIX API "access" is used on the client.

The information returned by the server in response to an ACCESS call is not permanent. It was correct at the exact time that the server performed the checks, but not necessarily afterwards. The server can revoke access permission at any time.

The client should use the effective credentials of the user to build the authentication information in the ACCESS request used to determine access rights. It is the effective user and group credentials that are used in subsequent read and write operations.

Many implementations do not directly support the ACCESS4_DELETE permission. Operating systems like UNIX will ignore the ACCESS4_DELETE bit if set on an access request on a non-directory object. In these systems, delete permission on a file is determined by the access permissions on the directory in which the file resides, instead of being determined by the permissions of the file itself. Therefore, the mask returned enumerating which access rights can be determined will have the ACCESS4_DELETE value set to 0. This indicates to the client that the server was unable to check that particular access right. The ACCESS4_DELETE bit in the access mask returned will then be ignored by the client.

17.2. Operation 4: CLOSE - Close File

17.2.1. SYNOPSIS

(cfh), seqid, open_stateid -> open_stateid
17.2.2. ARGUMENTS

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

17.2.3. RESULTS

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

17.2.4. 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 that associated with the supplied stateid. State associated with other OPENs is not affected.

If record locks are held, the client SHOULD release all locks before issuing a CLOSE. The server MAY free all outstanding locks on CLOSE but some servers may not support the CLOSE of a file that still has record locks held. The server MUST return failure if any locks would exist after the CLOSE.

The seqid value argument must have the value zero. If any other value is specified the server MUST return the error NFS4ERR_INVAL.

On success, the current filehandle retains its value.

17.2.5. 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.

17.3. Operation 5: COMMIT - Commit Cached Data

17.3.1. SYNOPSIS

(cfh), offset, count -> verifier

17.3.2. ARGUMENTS

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

17.3.3. RESULTS

struct COMMIT4resok {
  verifier4       writeverf;
};

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

17.3.4. 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.

17.3.5. IMPLEMENTATION

The COMMIT operation is similar in operation and semantics to the POSIX fsync(2) system call that synchronizes a file’s state with the disk (file data and metadata is flushed to disk or stable storage). COMMIT performs the same operation for a client, flushing any unsynchronized data and metadata on the server to the server’s disk or stable storage for the specified file. Like fsync(2), it may be that there is some modified data or no modified data to synchronize. The data may have been synchronized by the server’s normal periodic buffer synchronization activity. COMMIT should return NFS4_OK, unless there has been an unexpected error.

COMMIT differs from fsync(2) in that it is possible for the client to flush a range of the file (most likely triggered by a buffer-reclamation scheme on the client before file has been completely written).

The server implementation of COMMIT is reasonably simple. If the server receives a full file COMMIT request, that is starting at offset 0 and count 0, it should do the equivalent of fsync()’ing the file. Otherwise, it should arrange to have the cached data in the range specified by offset and count to be flushed to stable storage. In both cases, any metadata associated with the file must be flushed to stable storage before returning. It is not an error for there to be nothing to flush on the server. This means that the data and metadata that needed to be flushed have already been flushed or lost during the last server failure.

The client implementation of COMMIT is a little more complex. There are two reasons for wanting to commit a client buffer to stable storage. The first is that the client wants to reuse a buffer. In this case, the offset and count of the buffer are sent to the server in the COMMIT request. The server then flushes any cached data based on the offset and count, and flushes any metadata associated with the
file. It then returns the status of the flush and the write verifier. The other reason for the client to generate a COMMIT is for a full file flush, such as may be done at close. In this case, the client would gather all of the buffers for this file that contain uncommitted data, do the COMMIT operation with an offset of 0 and count of 0, and then free all of those buffers. Any other dirty buffers would be sent to the server in the normal fashion.

After a buffer is written by the client with the stable parameter set to UNSTABLE4, the buffer must be considered as modified by the client until the buffer has either been flushed via a COMMIT operation or written via a WRITE operation with stable parameter set to FILE_SYNC4 or DATA_SYNC4. This is done to prevent the buffer from being freed and reused before the data can be flushed to stable storage on the server.

When a response is returned from either a WRITE or a COMMIT operation and it contains a write verifier that is different than previously returned by the server, the client will need to retransmit all of the buffers containing uncommitted cached data to the server. How this is to be done is up to the implementor. If there is only one buffer of interest, then it should probably be sent back over in a WRITE request with the appropriate stable parameter. If there is more than one buffer, it might be worthwhile retransmitting all of the buffers in WRITE requests with the stable parameter set to UNSTABLE4 and then retransmitting the COMMIT operation to flush all of the data on the server to stable storage. The timing of these retransmissions is left to the implementor.

The above description applies to page-cache-based systems as well as buffer-cache-based systems. In those systems, the virtual memory system will need to be modified instead of the buffer cache.

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

17.4.1. SYNOPSIS

(cfh), name, type, attrs -> (cfh), change_info, attrs_set
17.4.2. ARGUMENTS

/*
 * CREATE: Create a non-regular file
 */
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;
};

17.4.3. RESULTS

struct CREATE4resok {
    change_info4   cinfo;
    bitmap4        attrset;  /* attributes set */
};

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

17.4.4. DESCRIPTION

The CREATE operation creates a non-regular file object in a directory with a given name. The OPEN operation MUST be used to create a regular file.
The objname specifies the name for the new object. The objtype determines the type of object to be created: directory, symlink, etc.

If an object of the same name already exists in the directory, the server will return the error NFS4ERR_EXIST.

For the directory where the new file object was created, the server returns change_info4 information in cinfo. With the atomic field of the change_info4 struct, the server will indicate if the before and after change attributes were obtained atomically with respect to the file object creation.

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

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

The createattrs specifies the initial set of attributes for the object. The set of attributes may include any writable attribute valid for the object type. When the operation is successful, the server will return to the client an attribute mask signifying which attributes were successfully set for the object.

If createattrs includes neither the owner attribute nor an ACL with an ACE for the owner, and if the server’s file system both supports and requires an owner attribute (or an owner ACE) then the server MUST derive the owner (or the owner ACE). This would typically be from the principal indicated in the RPC credentials of the call, but the server’s operating environment or file system semantics may dictate other methods of derivation. Similarly, if createattrs includes neither the group attribute nor a group ACE, and if the server’s file system both supports and requires the notion of a group attribute (or group ACE), the server MUST derive the group attribute (or the corresponding owner ACE) for the file. This could be from the RPC call’s credentials, such as the group principal if the credentials include it (such as with AUTH_SYS), from the group identifier associated with the principal in the credentials (for e.g., POSIX systems have a passwd database 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.
17.4.5. 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.

17.5. Operation 7: DELEGPURGE - Purge Delegations Awaiting Recovery

17.5.1. SYNOPIS

client ID ->

17.5.2. ARGUMENTS

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

17.5.3. RESULTS

struct DELEGPURGE4res {
    nfsstat4        status;
};

17.5.4. 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 should be used by clients that record delegation information on stable storage on the client. In this case, DELEGPURGE should be issued immediately after doing delegation recovery on all delegations 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. The set of delegations known to the server and the client may be different. The reason for this is that a client may fail after making a request which resulted in delegation but before it received the results and committed them to the client’s stable storage.
The server MAY support DELEGPURGE, but if it does not, it MUST NOT support CLAIM_DELEGATE_PREV.

17.6. Operation 8: DELEGRETURN - Return Delegation

17.6.1. SYNOPSIS

(cfh), stateid ->

17.6.2. ARGUMENTS

/*
 * DELEGRETURN: Return a delegation
 */
struct DELEGRETURN4args {
    /* CURRENT_FH: delegated file */
    stateid4   deleg_stateid;
};

17.6.3. RESULTS

struct DELEGRETURN4res {
    nfsstat4   status;
};

17.6.4. 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.

17.7. Operation 9: GETATTR - Get Attributes

17.7.1. SYNOPSIS

(cfh), attrbits -> attrbits, attrvals
17.7.2. ARGUMENTS

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

17.7.3. RESULTS

struct GETATTR4resok {
    fattr4 obj_attributes;
};

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

17.7.4. 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 which it was able to return, which will include all attributes requested by the client which are attributes supported by the server for the target file system. This bitmap is 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 for the target file system. If the server does not support a particular attribute on the target file system 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 5), even if previously, when the file
system was present, more attributes were supported.

All servers must support the mandatory attributes as specified in
File Attributes (Section 10.3.1), for all file systems, with the
exception of absent file systems.

On success, the current filehandle retains its value.

17.7.5. IMPLEMENTATION

17.8. Operation 10: GETFH - Get Current Filehandle

17.8.1. SYNOPSIS

  (cfh) -> filehandle

17.8.2. ARGUMENTS

  /* CURRENT_FH: */
  void;

17.8.3. RESULTS

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

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

17.8.4. DESCRIPTION

  This operation returns the current filehandle value.

  On success, the current filehandle retains its value.
17.8.5. 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
```

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

17.9.1. SYNOPTIS

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

17.9.2. ARGUMENTS

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

17.9.3. RESULTS

```
struct LINK4resok {
  change_info4 cinfo;
};
```

union LINK4res switch (nfsstat4 status) {
  case NFS4_OK:
    LINK4resok resok4;
  default:
    void;
};
17.9.4. 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.

17.9.5. IMPLEMENTATION

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

The statement "file and the target directory must reside within the same file system on the server" means that the fsid fields in the attributes for the objects are the same. If they reside on different file systems, the error, NFS4ERR_XDEV, is returned. On some servers, the filenames, "." and "..", are illegal as newname.

In the case that newname is already linked to the file represented by the saved filehandle, the server will return NFS4ERR_EXIST.

Note that symbolic links are created with the CREATE operation.

17.10. Operation 12: LOCK - Create Lock

17.10.1. SYNOPSIS

(cfh) locktype, reclaim, offset, length, locker -> stateid
17.10.2. ARGUMENTS

/*
 * For LOCK, transition from open_stateid and lock_owner
 * to a lock stateid.
 */
struct open_to_lock_owner4 {
  seqid4          open_seqid;
  stateid4        open_stateid;
  seqid4          lock_seqid;
  lock_owner4     lock_owner;
};

/*
 * For LOCK, existing lock stateid continues to request new
 * file lock for the same lock_owner and open_stateid.
 */
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;
};
17.10.3. RESULTS

```c
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;
};
```

17.10.4. DESCRIPTION

The LOCK operation requests a record lock for the octet 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 octets 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 octet offsets that fit within 32 bits. If the client specifies a range that includes an octet beyond the last octet offset of the 32-bit range, but does not include the last octet offset of the 32-bit and all of the octet 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.
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 record locking state associated with the current open file and lock owner. In the case in which it has, the argument is just a stateid for the set of locks associated with that open file and lock owner, together with a lock_seqid value which must be zero. In the case where no such state has been established, or the client does not have the stateid available, the argument contains the stateid of the open file with which this lock is to be associated, together with the lock_owner which which the lock is to be associated. The open_to_lock_owner case covers the very first lock done by a lock_owner for a given open file and offers a method to use the established state of the open_stateid to transition to the use of a lock stateid.

The client field of the lock owner, and all seqid values in the arguments have zero as the only valid value. When any of these are specified as other than zero, the server MUST return an NFS4ERR_INVAL. The client ID with which all owners and stateids are associated is the client ID associated with the session on which the request was issued. The client ID appearing in a LOCK4denied structure is the actual client associated with the conflicting lock, whether this is the client ID associated with the current session, or a different one.

On success, the current filehandle retains its value.

17.10.5. 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. The File Locking section 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 lockowner 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 lockowner, 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 octets 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.

17.11. Operation 13: LOCKT - Test For Lock

17.11.1. SYNOPSIS

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

17.11.2. ARGUMENTS

struct LOCKT4args {
  /* CURRENT_FH: file */
  nfs_lock_type4 locktype;
  offset4 offset;
  length4 length;
  lock_owner4 owner;
};

17.11.3. RESULTS

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

17.11.4. 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. The owner field in the results includes the client ID of the owner of conflicting lock, whether this
is the client ID associated with the current session or a different client ID. 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.

The client ID field of the owner should be specified as zero. The client ID used for ownership comparisons is that associated with the session on which the request is issued. If the client ID field is other than zero, the server MUST return the error NFS4ERR_INVAL.

On success, the current filehandle retains its value.

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

LOCKT uses a lock_owner4 rather than 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 lockowner. 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 lockowner 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.

17.12. Operation 14: LOCKU - Unlock File

17.12.1. SYNOPSIS

(cfh) type, seqid, stateid, offset, length -> stateid
17.12.2.  ARGUMENTS

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

17.12.3.  RESULTS

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

17.12.4.  DESCRIPTION

The LOCKU operation unlocks the record 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.

The seqid parameter should be specified as zero. If any other value is specified, the server must return an NFS4ERR_INVAL error.

On success, the current filehandle retains its value.

17.12.5.  IMPLEMENTATION

If the area to be unlocked does not correspond exactly to a lock actually held by the lockowner 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 lockowner. In all of these cases, allowed by POSIX locking 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.

17.13. Operation 15: LOOKUP - Lookup Filename

17.13.1. SYNOPSIS

(cfh), component -> (cfh)

17.13.2. ARGUMENTS

/*
 * LOOKUP: Lookup filename
 */
struct LOOKUP4args {
    /* CURRENT_FH: directory */
    component4 objname;
};

17.13.3. RESULTS

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

17.13.4. 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 a zero length string or if any component does not obey the UTF-8 definition, the error NFS4ERR_INVAL will be returned.
17.13.5. 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):

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

NFS version 4 servers depart from the semantics of previous NFS versions in allowing LOOKUP requests to cross mountpoints on the server. The client can detect a mountpoint crossing by comparing the fsid attribute of the directory with the fsid attribute of the directory looked up. If the fsids are different then the new directory is a server mountpoint. UNIX clients that detect a mountpoint crossing will need to mount the server’s 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 "..". NFS version 4 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.

17.14.1. SYNOPSIS

(cfh) -> (cfh)

17.14.2. ARGUMENTS

/* CURRENT_FH: object */
void;

17.14.3. RESULTS

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

17.14.4. 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.

As for LOOKUP, LOOKUPP will also cross mountpoints.

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

If the requester’s security flavor does not match that configured for the parent directory, then the server SHOULD return NFS4ERR_WRONGSEC (a future minor revision of NFSv4 may upgrade this to MUST) in the LOOKUPP response. However, if the server does so, it MUST support the new SECINFO_NO_NAME operation, so that the client can gracefully determine the correct security flavor. See the discussion of the SECINFO_NO_NAME operation for a description.

If the current filehandle is a named attribute directory that is associated with a file system object via OPENATTR (i.e., not a subdirectory of a named attribute directory) LOOKUPP SHOULD return the filehandle of the associated file system object.
17.14.5. IMPLEMENTATION

17.15. Operation 17: NVERIFY - Verify Difference in Attributes

17.15.1. SYNOPSIS

(cfh), fattr -> -

17.15.2. ARGUMENTS

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

17.15.3. RESULTS

struct NVERIFY4res {
   nfsstat4 status;
};

17.15.4. 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 NFS4ERRSAME must be returned.

On success, the current filehandle retains its value.

17.15.5. 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.

17.16. Operation 18: OPEN - Open a Regular File

17.16.1. SYNOPSIS

<cfh>, share_access, share_deny, owner, openhow, claim -> (cfh), stateid, cinfo, rflags, attrset, delegation

17.16.2. ARGUMENTS

/*
 * Various definitions for OPEN
 */
enum createmode4 {
    UNCHECKED4 = 0,
    GUARDED4   = 1,
    EXCLUSIVE4 = 2
};

union createhow4 switch (createmode4 mode) {
    case UNCHECKED4:
    case GUARDED4:
        fattr4         createattrs;
    case EXCLUSIVE4:
        verifier4      createverf;
};

enum opentype4 {
    OPEN4_NOCREATE = 0,
    OPEN4_CREATE   = 1
};

union openflag4 switch (opentype4 opentype) {
    case OPEN4_CREATE:
        createhow4     how;
    default:
        void;
};

/* Next definitions used for OPEN delegation */
enum limit_by4 {
    NFS_LIMIT_SIZE = 1,
    NFS_LIMIT_BLOCKS = 2,
    /* others as needed */
};

struct nfs_modified_limit4 {
    uint32_t num_blocks;
    uint32_t bytes_per_block;
};

union nfs_space_limit4 switch (limit_by4 limitby) {
    /* limit specified as file size */
    case NFS_LIMIT_SIZE:
        uint64_t filesize;
    /* limit specified by number of blocks */
    case NFS_LIMIT_BLOCKS:
        nfs_modified_limit4 mod_blocks;
};

/* Share Access and Deny constants for open argument */
const OPEN4_SHARE_ACCESS_READ   = 0x00000001;
const OPEN4_SHARE_ACCESS_WRITE  = 0x00000002;
const OPEN4_SHARE_ACCESS_BOTH   = 0x00000003;
const OPEN4_SHARE_DENY_NONE     = 0x00000000;
const OPEN4_SHARE_DENY_READ     = 0x00000001;
const OPEN4_SHARE_DENY_WRITE    = 0x00000002;
const OPEN4_SHARE_DENY_BOTH     = 0x00000003;

/* new flags for share_access field of OPEN4args */
const OPEN4_SHARE_ACCESS_WANT_DELEG_MASK        = 0xFF00;
const OPEN4_SHARE_ACCESS_WANT_NO_PREFERENCE     = 0x0000;
const OPEN4_SHARE_ACCESS_WANT_READ_DELEG        = 0x0100;
const OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG       = 0x0200;
const OPEN4_SHARE_ACCESS_WANT_ANY_DELEG         = 0x0300;
const OPEN4_SHARE_ACCESS_WANT_NO_DELEG          = 0x0400;
const OPEN4_SHARE_ACCESS_WANT_CANCEL            = 0x0500;
const OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL = 0x10000;
const OPEN4_SHARE_ACCESS_WANT_PUSH_DELEG_WHEN_UNCONTENDED = 0x20000;

enum open_delegation_type4 {
    OPEN_DELEGATE_NONE      = 0,
    OPEN_DELEGATE_READ      = 1,
    OPEN_DELEGATE_WRITE     = 2,
OPEN_DELEGATE_NONE_EXT = 3 /* new to v4.1 */
};

enum open_claim_type4 {
    CLAIM_NULL = 0,
    CLAIM_PREVIOUS = 1,
    CLAIM_DELEGATE_CUR = 2,
    CLAIM_DELEGATE_PREV = 3,
    CLAIM_FH = 4, /* new to v4.1 */
    CLAIM_DELEG_CUR_FH = 5, /* new to v4.1 */
    CLAIM_DELEG_PREV_FH = 6 /* new to v4.1 */
};

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;
};

17.16.3. RESULTS

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) */
    nfsace4 permissions;    /* Defines users who don’t
                               * need an ACCESS call to
                               * open for read */
};
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. */

enum why_no_delegation4 { /* new to v4.1 */
  WND4_NOT_WANTED = 0,
  WND4CONTENTION = 1,
  WND4RESOURCE = 2,
  WND4NOT_SUPP_FTYPE = 3,
  WND4_WRITE_DELEG_NOT_SUPP_FTYPE = 4,
  WND4NOT_SUPP_UPGRADE = 5,
  WND4NOT_SUPP_DOWNGRADE = 6,
  WND4CANCELED = 7,
  WND4IS_DIR = 8
};

union open_none_delegation4 /* new to v4.1 */
switch (why_no_delegation4 ond_why) {
  case WND4CONTENTION:
    bool ond_server_will_push_deleg;
  case WND4RESOURCE:
    bool ond_server_will_signal_avail;
  default:
    void;
}

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;
  case OPEN_DELEGATE_NONE_EXT: /* new to v4.1 */
    open_none_delegation4 od_whynone;
}
typedef enum {
    OPEN4_RESULT_CONFIRM = 0x00000002,
    OPEN4_RESULT_LOCKTYPE_POSIX = 0x00000004,
    OPEN4_RESULT_PRESERVE_UNLINKED = 0x00000008,
    OPEN4_RESULT_MAY_NOTIFY_LOCK = 0x00000020,
} OPEN4_RESULT;

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;
        break;
    default:
        void;
        break;
};

17.16.4. DESCRIPTION

The OPEN operation creates and/or opens a regular file in a directory with the provided name. If the file does not exist at the server and creation is desired, specification of the method of creation is provided by the openhow parameter. The client has the choice of three creation methods: UNCHECKED, GUARDED, or EXCLUSIVE.

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.

UNCHECKED means that the file should be created if a file of that name does not exist and encountering an existing regular file of that name is not an error. For this type of create, createattrs specifies the initial set of attributes for the file. The set of attributes
may include any writable attribute valid for regular files. When an UNCHECKED 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 GUARDED is specified, the server checks for the presence of a duplicate object by name before performing the create. If a duplicate exists, an error of NFS4ERR_EXIST is returned as the status. If the object does not exist, the request is performed as described for UNCHECKED. For each of these cases (UNCHECKED and GUARDED) where the operation is successful, the server will return to the client an attribute mask signifying which attributes were successfully set for the object.

EXCLUSIVE specifies that the server is to follow exclusive creation semantics, using the verifier to ensure exclusive creation of the target. The server should check for the presence of a duplicate object by name. If the object does not exist, the server creates the object and stores the verifier with the object. If the object does exist and the stored verifier matches the client provided verifier, the server uses the existing object as the newly created object. If the stored verifier does not match, then an error of NFS4ERR_EXIST is returned. No attributes may be provided in this case, since the server may use an attribute of the target object to store the verifier. 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 each OPEN, the client must provide a value for the owner field for the OPEN argument. The client ID associated with the owner is not derived from the client field of the owner parameter but is instead the client ID associated with the session on which the request is issued. If the client ID field of the owner parameter is not zero, the server MUST return an NFS4ERR_INVAL error.
additional discussion of SHARE semantics see Section 8.8.

The seqid value is not used in NFSv4.1. If the value passed is not zero, the server MUST return an NFS4ERR_INVAL error.

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 seven claim types as follows:

<table>
<thead>
<tr>
<th>open type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAIM_NULL CLAIM_FH</td>
<td>For the client, this is a new OPEN request and there is no previous state associate with the file for the client. With CLAIM_NULL the file is identified by the current filehandle and the specified component name. With CLAIM_FH (new to v4.1) the file is identified by just the current filehandle.</td>
</tr>
<tr>
<td>CLAIM_PREVIOUS</td>
<td>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.</td>
</tr>
<tr>
<td>CLAIM_DELEGATE_CUR</td>
<td>The client is claiming a delegation for OPEN as granted by the server. Generally this is done as part of recalling a delegation. With CLAIM_DELEGATE_CUR, the file is identified by the current filehandle and the specified component name. With CLAIM_DELEG_PREV_FH (new to v4.1), the file is identified by just the current filehandle.</td>
</tr>
<tr>
<td>CLAIM_DELEG_PREV_FH</td>
<td>The client is claiming a delegation granted to a previous client instance; used after the client reboots. The server MAY support CLAIM_DELEG_PREV or CLAIM_DELEG_PREV_FH. If it does support either open type, CREATE_SESSION MUST NOT remove the client’s delegation state, and the server MUST support the DELEGPURGE operation.</td>
</tr>
</tbody>
</table>
For OPEN requests whose claim type is other than CLAIM_PREVIOUS (i.e. requests other than those devoted to reclaiming opens after a server reboot) that reach the server during its grace or lease expiration period, the server returns an error of NFS4ERR_GRACE.

For any OPEN request, the server may return an open delegation, which allows further opens and closes to be handled locally on the client as described in the section Open Delegation. Note that delegation is up to the server to decide. The client should never assume that delegation will or will not be granted in a particular instance. It should always be prepared for either case. A partial exception is the reclaim (CLAIM_PREVIOUS) case, in which a delegation type is claimed. In this case, delegation will always be granted, although the server may specify an immediate recall in the delegation structure.

The rflags returned by a successful OPEN allow the server to return information governing how the open file is to be handled.

- OPEN4_RESULT_CONFIRM is deprecated and MUST not be returned by an NFSv4.1 server.
- OPEN4_RESULT_LOCKTYPE_POSIX indicates the server’s file locking behavior supports the complete set of Posix locking techniques. From this the client can choose to manage file locking state in a way to handle a mis-match of file locking management.
- OPEN4_RESULT_PRESERVE_UNLINKED indicates the server will preserve the open file if the client (or any other client) removes the file as long as it is open. Furthermore, the server promises to preserve the file through the grace period after server reboot, thereby giving the client the opportunity to reclaim his open.
- OPEN4_RESULT_MAY_NOTIFY_LOCK indicates that the server may attempt CB_NOTIFY_LOCK callbacks for locks on this file. This flag is a hint only, and may be safely ignored by the client.

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 the section "UTF-8 Related Errors" for further discussion.

When an OPEN is done and the specified lockowner already has the resulting filehandle open, the result is to "OR" together the new share and deny status together with the existing status. In this case, only a single CLOSE need be done, even though multiple 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 lockowner.

If the underlying file system at the server is only accessible in a read-only mode and the OPEN request has specified ACCESS_WRITE or ACCESS_BOTH, the server will return NFS4ERR_ROFS to indicate a read-only file system.

As with the CREATE operation, the server MUST derive the owner, owner ACE, group, or group ACE if any of the four attributes are required and supported by the server’s file system. For an OPEN with the EXCLUSIVE4 createmode, the server has no choice, since such OPEN calls do not include the createattrs field. Conversely, if createattrs is specified, and includes owner or group (or corresponding ACEs) that the principal in the RPC call’s credentials does not have authorization to create files for, then the server may return NFS4ERR_PERM.

In the case of a OPEN which specifies a size of zero (e.g. truncation) and the file has named attributes, the named attributes are left as is. They are not removed.

NFSv4.1 gives more precise control to clients over acquisition of delegations via the following new flags for the share_access field of OPEN4args:

- OPEN4_SHARE_ACCESS_WANT_READ_DELEG
- OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG
- OPEN4_SHARE_ACCESS_WANT_ANY_DELEG
- OPEN4_SHARE_ACCESS_WANT_NO_DELEG
- OPEN4_SHARE_ACCESS_WANT_CANCEL
- OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL
- OPEN4_SHARE_ACCESS_WANT_PUSH_DELEG_WHEN_UNCONTENDED

If (share_access & OPEN4_SHARE_ACCESS_WANT_DELEG_MASK) is not zero, then the client will have specified one and only one of:

- OPEN4_SHARE_ACCESS_WANT_READ_DELEG
- OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG
OPEN4_SHARE_ACCESS_WANT_ANY_DELEG

OPEN4_SHARE_ACCESS_WANT_NO_DELEG

OPEN4_SHARE_ACCESS_WANT_CANCEL

Otherwise the client is indicating no desire for a delegation and the server MAY or MAY not return a delegation in the OPEN response.

If the server supports the new _WANT_ flags and the client issues one or more of the new flags, then in the event the server does not return a delegation, it MUST return a delegation type of OPEN_DELEGATE_NONE_EXT. od_whyNone indicates why no delegation was returned and will be one of:

WND4_NOT_WANTED The client specified OPEN4_SHARE_ACCESS_WANT_NO_DELEG.

WND4CONTENTION There is a conflicting delegation or open on the file.

WND4RESOURCE Resource limitations prevent the server from granting a delegation.

WND4NOT_SUPP_FTYPE The server does not support delegations on this file type.

WND4_WRITE_DELEGNOT_SUPP_FTYPE The server does not support write delegations on this file type.

WND4_NOT_SUPP_UPGRADE The server does not support atomic upgrade of a read delegation to a write delegation.

WND4_NOT_SUPP_DOWNGRADE The server does not support atomic downgrade of a write delegation to a read delegation.

WND4 CANCELED The client specified OPEN4_SHARE_ACCESS_WANT_CANCEL and now any "want" for this file object is cancelled.

WND4 IS DIR The specified file object is a directory, and the operation is OPEN or WANT_DELEGATION which do not support delegations on directories.

OPEN4_SHARE_ACCESS_WANT_READ_DELEG,
OPEN_SHARE_ACCESS_WANT_WRITE_DELEG, or
OPEN_SHARE_ACCESS_WANT_ANY_DELEG mean, respectively, the client wants a read, write, or any delegation regardless which of OPEN4_SHARE_ACCESS_READ, OPEN4_SHARE_ACCESS_WRITE, or
OPEN4_SHARE_ACCESS_BOTH is set. If the client has a read delegation on a file, and requests a write delegation, then the client is requesting atomic upgrade of its read delegation to a write delegation. If the client has a write delegation on a file, and requests a read delegation, then the client is requesting atomic downgrade to a read delegation. A server MAY support atomic upgrade or downgrade. If it does, then the returned delegation_type of OPEN_DELEGATE_READ or OPEN_DELEGATE_WRITE that is different than the delegation type the client currently has, indicates successful upgrade or downgrade. If it does not support atomic delegation upgrade or downgrade, then od_why will be WND4_NOT_SUPP_UPGRADE or WND4_NOT_SUPP_DOWNGRADE.

OPEN4_SHARE_ACCESS_WANT_NO_DELEG means the client wants no delegation.

OPEN4_SHARE_ACCESS_WANT_CANCEL means the client wants no delegation and wants to cancel any previously registered "want" for a delegation.

The client may set one or both of
OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL and
OPEN4_SHARE_ACCESS_WANT_PUSH_DELEG_WHEN_UNCONTENDED. However, they will have no effect unless one of following are set:

- OPEN4_SHARE_ACCESS_WANT_READ_DELEG
- OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG
- OPEN4_SHARE_ACCESS_WANT_ANY_DELEG

If the client specifies OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL, then it wishes to register a "want" for a delegation, in the event the OPEN results do not include a delegation. If so and the server denies the delegation due to insufficient resources, the server MAY later inform the client, via the CB_RECALLABLE_OBJ_AVAIL operation, that the resource limitation condition has eased. The server will tell the client that it intends to send a future CB_RECALLABLE_OBJ_AVAIL operation by setting delegation_type in the results to OPEN_DELEGATE_NONE_EXT, ond_why to WND4_RESOURCE, and ond_server_will_signal_avail set to TRUE. If ond_server_will_signal_avail is set to TRUE, the server MUST later send a CB_RECALLABLE_OBJ_AVAIL operation.

If the client specifies OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_UNCONTENDED, then it wishes to register a "want" for a delegation, in the event the OPEN results
do not include a delegation. If so and the server denies the
delegation due to insufficient resources, the server MAY later inform
the client, via the CB_PUSH_DELEG operation, that the resource
limitation condition has eased. The server will tell the client that
it intends to send a future CB_PUSH_DELEG operation by setting
delegation_type in the results to OPEN_DELEGATE_NONE_EXT, ond_why to
WND4_CONTENTION, and ond_server_will_push_deleg to TRUE. If
ond_server_will_push_deleg is TRUE, the server MUST later send a
CB_RECALLABLE_OBJ_AVAIL operation.

If the client has previously registered a want for a delegation on a
file, and then sends a request to register a want for a delegation on
the same file, the server MUST return a new error:
NFS4ERR_DELEG_ALREADY_WANTED. If the client wishes to register a
different type of delegation want for the same file, it MUST cancel
the existing delegation WANT.

17.16.5. IMPLEMENTATION

The OPEN operation contains support for EXCLUSIVE create. The
mechanism is similar to the support in NFS version 3 [18]. However,
this mechanism is not needed if a server stores its reply cache in
stable storage. If the server indicates (via the csr_persist field
in the response to CREATE_SESSION) the client SHOULD NOT use OPEN’s
approach to exclusive create.

In absence of csr_persist being TRUE, the client invokes exclusive
create by setting the how parameter is EXCLUSIVE. In this case, the
client provides a verifier that can reasonably be expected to be
unique. A combination of a client identifier, perhaps the client
network address, and a unique number generated by the client, perhaps
the RPC transaction identifier, may be appropriate. This mechanism
allows reliable exclusive create semantics even when the server does
not support the storing session reply information in stable storage.

If the object does not exist, the server creates the object and
stores the verifier in stable storage. For file systems that do not
provide a mechanism for the storage of arbitrary file attributes, the
server may use one or more elements of the object meta-data to store
the verifier. The verifier must be stored in stable storage to
prevent erroneous failure on retransmission of the request. It is
assumed that an exclusive create is being performed because exclusive
semantics are critical to the application. Because of the expected
usage, exclusive CREATE does not rely solely on the normally volatile
duplicate request cache for storage of the verifier. The duplicate
request cache in volatile storage does not survive a crash and may
actually flush on a long network partition, opening failure windows.
In the UNIX local file system environment, the expected storage
location for the verifier on creation is the meta-data (time stamps) of the object. For this reason, an exclusive object create may not include initial attributes because the server would have nowhere to store the verifier.

If the server can not support these exclusive create semantics, possibly because of the requirement to commit the verifier to stable storage, it should fail the OPEN request with the error, NFS4ERR_NOTSUPP.

During an exclusive CREATE request, if the object already exists, the server reconstructs the object’s verifier and compares it with the verifier in the request. If they match, the server treats the request as a success. The request is presumed to be a duplicate of an earlier, successful request for which the reply was lost and that the server duplicate request cache mechanism did not detect. If the verifiers do not match, the request is rejected with the status, NFS4ERR_EXIST.

Once the client has performed a successful exclusive create, it must issue a SETATTR to set the correct object attributes. Until it does so, it should not rely upon any of the object attributes, since the server implementation may need to overload object meta-data to store the verifier. The subsequent SETATTR must not occur in the same COMPOUND request as the OPEN. This separation will guarantee that the exclusive create mechanism will continue to function properly in the face of retransmission of the request.

Use of the GUARDED attribute does not provide exactly-once semantics. In particular, if a reply is lost and the server does not detect the retransmission of the request, the 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 GUARDED to prevent against a sequence like: create, write, create (retransmitted) from occurring.

For SHARE reservations, the client must specify a value for share_access that is one of READ, WRITE, or BOTH. For share_deny, the client must specify one of NONE, READ, WRITE, or BOTH. If the client fails to do this, the server must return NFS4ERR_INVAL.

Based on the share_access value (READ, WRITE, or BOTH) the client should check that the requester has the proper access rights to perform the specified operation. This would generally be the results of applying the ACL access rules to the file for the current
requester. However, just as with the ACCESS operation, the client should not attempt to second-guess the server’s decisions, as access rights may change and may be subject to server administrative controls outside the ACL framework. If the requester is not authorized to READ or WRITE (depending on the share_access value), the server must return NFS4ERR_ACCESS. Note that since the NFS version 4 protocol does not impose any requirement that READs and WRITEs issued for an open file have the same credentials as the OPEN itself, the server still must do appropriate access checking on the READs and WRITEs themselves.

If the component provided to OPEN is a symbolic link, the error NFS4ERR_SYMLINK will be returned to the client. If the current filehandle is not a directory, the error NFS4ERR_NOTDIR will be returned.

The use of the OPEN4_RESULT_PRESERVE_UNLINKED result flag allows a client avoid the common implementation practice of renaming an open file to ".nfs<unique value>" after it removes the file. After the server returns OPEN4_RESULT_PRESERVE_UNLINKED, if a client issues a REMOVE operation that would reduce the file’s link count to zero, the server SHOULD report a value of zero for the FATTR4_NUMLINKS attribute on the file.

17.16.5.1. WARNING TO CLIENT IMPLEMENTORS

OPEN resembles LOOKUP in that it generates a filehandle for the client to use. Unlike LOOKUP though, OPEN creates server state on the filehandle. In normal circumstances, the client can only release this state with a CLOSE operation. CLOSE uses the current filehandle to determine which file to close. Therefore the client MUST follow every OPEN operation with a GETFH operation in the same COMPOUND procedure. This will supply the client with the filehandle such that CLOSE can be used appropriately.

Simply waiting for the lease on the file to expire is insufficient because the server may maintain the state indefinitely as long as another client does not attempt to make a conflicting access to the same file.


17.17.1. SYNOPSIS

(cfh) createdir -> (cfh)
17.17.2. ARGUMENTS

/*
 * OPENATTR: open named attributes directory
 */
struct OPENATTR4args {
    /* CURRENT_FH: object */
    bool createdir;
};

17.17.3. RESULTS

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

17.17.4. 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.

17.17.5. IMPLEMENTATION

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

17.18.1. SYNOPSIS

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

17.18.2. ARGUMENTS

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

17.18.3. RESULTS

struct OPEN_DOWNGRADE4resok {
    stateid4    open_stateid;
};

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

17.18.4. DESCRIPTION

This operation is used to adjust the share_access and share_deny bits for a given open. This is necessary when a given lockowner 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 openowner 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.

The seqid value is not used in NFSv4.1. If the value passed is not zero, the server MUST return an NFS4ERR_INVAL error.

On success, the current filehandle retains its value.

17.19. Operation 22: PUTFH - Set Current Filehandle

17.19.1. SYNOPSIS

    filehandle -> (cfh)

17.19.2. ARGUMENTS

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

17.19.3. RESULTS

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

17.19.4. 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.

17.19.5. IMPLEMENTATION

    Commonly used as the first operator in an NFS request to set the context for following operations.
17.20. Operation 23: PUTPUBFH - Set Public Filehandle

17.20.1. SYNOPSIS

- -> (cfh)

17.20.2. ARGUMENT

void;

17.20.3. RESULT

/*
* PUTPUBFH: Set public filehandle
*/

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

17.20.4. DESCRIPTION

Replaces the current filehandle with the filehandle that represents the public filehandle of the server’s name space. This filehandle may be different from the "root" filehandle which may be associated with some other directory on the server.

The public filehandle represents the concepts embodied in RFC2054 [25], RFC2055 [26], RFC2224 [32]. The intent for NFS version 4 is that the public filehandle (represented by the PUTPUBFH operation) be used as a method of providing WebNFS server compatibility with NFS versions 2 and 3.

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.

17.20.5. IMPLEMENTATION

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

With the NFS version 2 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 [32] contains further
discussion of the functionality. With NFS version 4, 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 NFS version 4. 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 [32] 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 NFS version 4. It is likely, therefore that because of server implementation details, an NFS version 3 absolute public filehandle lookup may behave differently than an NFS version 4 absolute resolution.

There is a form of security negotiation as described in RFC2755 [33] that uses the public filehandle a method of employing SNEGO. This method is not available with NFS version 4 as filehandles are not overloaded with special meaning and therefore do not provide the same framework as NFS versions 2 and 3. Clients should therefore use the security negotiation mechanisms described in this RFC.

17.20.6. ERRORS

17.21. Operation 24: PUTROOTFH - Set Root Filehandle

17.21.1. SYNOPSIS

    -> (cfh)

17.21.2. ARGUMENTS

    void;

17.21.3. RESULTS

    /*
     * PUTROOTFH: Set root filehandle
     */
    struct PUTROOTFH4res {
        /* CURRENT_FH: root fh */
        nfsstat4 status;
    };
17.21.4. 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.

17.21.5. IMPLEMENTATION

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

17.22. Operation 25: READ – Read from File

17.22.1. SYNOPSIS

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

17.22.2. ARGUMENTS

/*
 * READ: Read from file
 */

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

17.22.3. RESULTS

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

union READ4res switch (nfsstat4 status) {
   case NFS4_OK:
      READ4resok     resok4;
   default:
      void;
};
17.22.4. 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 record lock or share reservation request. The stateid is
used by the server to verify that the associated share reservation
and any record 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 return; otherwise,
NFS4ERR_INVAL is returned.

For a READ with a stateid value of all bits 0, the server MAY allow
the READ to be serviced subject to mandatory file locks or the
current share deny modes for the file. For a READ with a stateid
value of all bits 1, the server MAY allow READ operations to bypass
locking checks at the server.

On success, the current filehandle retains its value.

17.22.5. IMPLEMENTATION

It is possible for the server to return fewer than count bytes of
data. If the server returns less than the count requested and eof is
set to FALSE, the client should issue another READ to get the
remaining data. A server may return less data than requested under
several circumstances. The file may have been truncated by another
client or perhaps on the server itself, changing the file size from
what the requesting client believes to be the case. This would
reduce the actual amount of data available to the client. It is
possible that the server may back off the transfer size and reduce
the read request return. Server resource exhaustion may also occur
necessitating a smaller read return.

If mandatory file locking is on for the file, and if the region
corresponding to the data to be read from file is write locked by an
owner not associated the stateid, the server will return the
NFS4ERR_LOCKED error. The client should try to get the appropriate
read record lock via the LOCK operation before re-attempting the
READ. When the READ completes, the client should release the record
lock via LOCKU.

17.23. Operation 26: READDIR - Read Directory

17.23.1. SYNOPSIS

    (cfh), cookie, cookieverf, dircount, maxcount, attr_request ->
    cookieverf { cookie, name, attrs }

17.23.2. ARGUMENTS

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

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;
};

17.23.4. 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_TOO_SMALL will be returned to the client.

Finally, attr_request represents the list of attributes to be returned for each directory entry supplied by the server.

On successful return, the server’s response will provide a list of directory entries. Each of these entries contains the name of the directory entry, a cookie value for that entry, and the associated attributes as requested. The "eof" flag has a value of TRUE if there are no more entries in the directory.

The cookie value is only meaningful to the server and is used as a "bookmark" for the directory entry. As mentioned, this cookie is used by the client for subsequent READDIR operations so that it may continue reading a directory. The cookie is similar in concept to a READ offset but should not be interpreted as such by the client. Ideally, the cookie value should not change if the directory is modified since the client may be caching these values.

In some cases, the server may encounter an error while obtaining the attributes for a directory entry. Instead of returning an error for the entire READDIR operation, the server can instead return the attribute ‘fatrr4 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 fatrr4 rdattr_error attribute for this method to work properly. If the client does not request the attribute, the server has no choice but to return failure for the entire READDIR operation.

For some file system environments, the directory entries "." and ".." have special meaning and in other environments, they may not. If the server supports these special entries within a directory, they should not be returned to the client as part of the READDIR response. To
enable some client environments, the cookie values of 0, 1, and 2 are
to be considered reserved. Note that the UNIX client will use these
values when combining the server’s response and local representations
to enable a fully formed UNIX directory presentation to the
application.

For READDIR arguments, cookie values of 1 and 2 should not be used
and for READDIR results cookie values of 0, 1, and 2 should not be
returned.

On success, the current filehandle retains its value.

17.23.5. IMPLEMENTATION

The server’s file system directory representations can differ
greatly. A client’s programming interfaces may also be bound to the
local operating environment in a way that does not translate well
into the NFS protocol. Therefore the use of the dircount and
maxcount fields are provided to allow the client the ability to
provide guidelines to the server. If the client is aggressive about
attribute collection during a READDIR, the server has an idea of how
to limit the encoded response. The dircount field provides a hint on
the number of entries based solely on the names of the directory
entries. Since it is a hint, it may be possible that a dircount
value is zero. In this case, the server is free to ignore the
dircount value and return directory information based on the
specified maxcount value.

The cookieverf may be used by the server to help manage cookie values
that may become stale. It should be a rare occurrence that a server
is unable to continue properly reading a directory with the provided
cookie/cookieverf pair. The server should make every effort to avoid
this condition since the application at the client may not be able to
properly handle this type of failure.

The use of the cookieverf will also protect the client from using
READDIR cookie values that may be stale. For example, if the file
system has been migrated, the server may or may not be able to use
the same cookie values to service READDIR as the previous server
used. With the client providing the cookieverf, the server is able
to provide the appropriate response to the client. This prevents the
case where the server may accept a cookie value but the underlying
directory has changed and the response is invalid from the client’s
context of its previous READDIR.

Since some servers will not be returning "." and ".." entries as has
been done with previous versions of the NFS protocol, the client that
requires these entries be present in READDIR responses must fabricate
17.24. Operation 27: READLINK - Read Symbolic Link

17.24.1. SYNOPSIS

(cfh) -> linktext

17.24.2. ARGUMENTS

/* CURRENT_FH: symlink */
void;

17.24.3. RESULTS

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

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

17.24.4. 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.

17.24.5. 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.

17.25. Operation 28: REMOVE - Remove File System Object

17.25.1. SYNOPSIS

(cfh), filename -> change_info

17.25.2. ARGUMENTS

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

17.25.3. RESULTS

struct REMOVE4resok {
    change_info4 cinfo;
};

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

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

On success, the current filehandle retains its value.

17.25.5. IMPLEMENTATION

NFS versions 2 and 3 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() 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. NFS version 4 REMOVE can be used to delete any directory entry independent of its file type. The implementor of an NFS version 4 client's entry points from the unlink() and rmdir() system calls should first check the file type against the types the system call is allowed to remove before issuing a REMOVE. Alternatively, the implementor can produce a COMPOUND call that includes a LOOKUP/VERIFY sequence to verify the file type before a REMOVE operation in the same COMPOUND call.

The concept of last reference is server specific. However, if the numlinks field in the previous attributes of the object had the value 1, the client should not rely on referring to the object via a filehandle. Likewise, the client should not rely on the resources (disk space, directory entry, and so on) formerly associated with the object becoming immediately available. Thus, if a client needs to be able to continue to access a file after using REMOVE to remove it, the client should take steps to make sure that the file will still be accessible. The usual mechanism used is to RENAME the file from its old name to a new hidden name.

If the server finds that the file is still open when the REMOVE arrives:

- 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.

- 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.

17.26.1. SYNOPSIS

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

17.26.2. ARGUMENTS

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

17.26.3. RESULTS

struct RENAME4resok {
    change_info4 source_cinfo;
    change_info4 target_cinfo;
};

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

17.26.4. 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 the IMPLEMENTATION subsection of the section
"Operation 28: REMOVE - Remove File System Object" for client and
server actions whenever a target is removed). If they are not
compatible or if the target is a directory but not empty, the server
will return the error, NFS4ERR_EXIST.

If oldname and newname both refer to the same file (they might be
hard links of each other), then RENAME should perform no action and
return success.

For both directories involved in the RENAME, the server returns
change_info4 information. With the atomic field of the change_info4
struct, the server will indicate if the before and after change
attributes were obtained atomically with respect to the rename.

If the oldname refers to a named attribute and the saved and current
filehandles refer to different file system objects, the server will
return NFS4ERR_XDEV just as if the saved and current filehandles
represented directories on different file systems.

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

17.26.5. IMPLEMENTATION

The RENAME operation must be atomic to the client. The statement
"source and target directories must reside on the same file system on
the server" means that the fsid fields in the attributes for the
directories are the same. If they reside on different file systems,
the error, NFS4ERR_XDEV, is returned.

Based on the value of the fh_expire_type attribute for the object,
the filehandle may or may not expire on a RENAME. However, server
implementors are strongly encouraged to attempt to keep filehandles
from expiring in this fashion.

On some servers, the file names "." and ".." are illegal as either
oldname or newname, and will result in the error NFS4ERR_BADNAME. In
addition, on many servers the case of oldname or newname being an
alias for the source directory will be checked for. Such servers
will return the error NFS4ERR_INVAL in these cases.

If either of the source or target filehandles are not directories,
the server will return NFS4ERR_NOTDIR.
17.27. Operation 31: RESTOREFH - Restore Saved Filehandle

17.27.1. SYNOPSIS

(sfh) -> (cfh)

17.27.2. ARGUMENTS

/* SAVED_FH: */
void;

17.27.3. RESULTS

/*
    * RESTOREFH: Restore saved filehandle
    */

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

17.27.4. DESCRIPTION

Set the current filehandle to the value in the saved filehandle. If there is no saved filehandle then return the error NFS4ERR_RESTOREFH.

17.27.5. 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 attributes (pre-op dir attrs)
CREATE optbits "foo" attr
GETATTR attributes (file attributes)
RESTOREFH
GETATTR attributes (post-op dir attrs)
17.27.6. ERRORS

17.28. Operation 32: SAVEFH - Save Current Filehandle

17.28.1. SYNOPSIS

(cfh) -> (sfh)

17.28.2. ARGUMENTS

/* CURRENT_FH: */
void;

17.28.3. RESULTS

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

17.28.4. 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.

17.28.5. IMPLEMENTATION

17.29. Operation 33: SECINFO - Obtain Available Security

17.29.1. SYNOPSIS

(cfh), name -> { secinfo }
17.29.2.  ARGUMENTS

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

17.29.3.  RESULTS

/*
 * 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;
};
17.29.4. 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 the 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 RFC1831 [4]), or RPCSEC_GSS (as defined in
RFC2203 [5]). The field flavor can also any other security flavor
registered with IANA.

For the flavors AUTH_NONE and AUTH_SYS, no additional security
information is returned. The same is true of many (if not most)
other security flavors, including AUTH_DH. For a return value of
RPCSEC_GSS, a security triple is returned that contains the mechanism
object id (as defined in RFC2743 [8]), the quality of protection (as
defined in RFC2743 [8]) and the service type (as defined in RFC2203
[5]). 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.

17.29.5. 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, RESTOREFH, RENAME, and indirectly READDR. 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:

- 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.

- For LINK, PUTFH, PUTROOTFH, PUTPUBFH, RENAME, and RESTOREFH, the client will use SECINFO_NO_NAME { style = SECINFO_STYLE4_CURRENT_FH }. The client will prefix the SECINFO_NO_NAME operation with the appropriate PUTFH, PUTPUBFH, or PUTROOTFH operation that provides the filehandle originally provided by the PUTFH, PUTPUBFH, PUTROOTFH, or RESTOREFH, or for the failed LINK or RENAME, the SAVEFH.

- NOTE: In NFSv4.0, the client was required to use SECINFO, and had to reconstruct the parent of the original file handle, and the component name of the original filehandle.

- For LOOKUPP, the client will use SECINFO_NO_NAME { style = SECINFO_STYLE4_PARENT } and provide the filehandle with equals the filehandle originally provided to LOOKUPP.

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 did 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.

See the section "Security Considerations" for a discussion on the recommendations for security flavor used by SECINFO and SECINFO_NO_NAME.
17.30. Operation 34: SETATTR - Set Attributes

17.30.1. SYNOPSIS

(cfh), stateid, attrmask, attr_vals -> attrsset

17.30.2. ARGUMENTS

/*
 * SETATTR: Set attributes
 */

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

17.30.3. RESULTS

struct SETATTR4res {
    nfsstat4 status;
    bitmap4 attrsset;
};

17.30.4. 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 file 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 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 record locks, for those cases in which a server is implementing mandatory record locking behavior. A valid stateid should always be specified. When the file size attribute is not set, the special stateid consisting of all bits zero should 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.

17.30.5. 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 followed by a SETATTR.

The file size attribute is used to request changes to the size of a file. A value of 0 (zero) causes the file to be truncated, a value less than the current size of the file causes data from new size to the end of the file to be discarded, and a size greater than the current size of the file causes logically zeroed data bytes to be added to the end of the file. Servers are free to implement this using holes or actual zero data bytes. Clients should not make any assumptions regarding a server’s implementation of this feature, beyond that the bytes returned will be zeroed. Servers must support extending the file size via SETATTR.

SETATTR is not guaranteed atomic. A failed SETATTR may partially change a file’s attributes.

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

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 NFS version 3. 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 NFS version 4 emulation. Therefore, NFS version 4 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 attributes bits not requested to be set by the client, and must be equal to the mask of attributes requested to be set only if the SETATTR completes without error.

17.31. Operation 37: VERIFY - Verify Same Attributes

17.31.1. SYNOPSIS

(cfh), fattr -> -

17.31.2. ARGUMENTS

/*
 * VERIFY: Verify attributes same
 */

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

17.31.3. RESULTS

struct VERIFY4res {
    nfsstat4 status;
};

17.31.4. 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
17.31.5. 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.

17.32. Operation 38: WRITE - Write to File

17.32.1. SYNOPSIS

```
(cfh), stateid, offset, stable, data -> count, committed, writeverf
```
17.32.2.  ARGUMENTS

/*
 * WRITE: Write to file
 */
enum stable_how4 {
    UNSTABLE4       = 0,
    DATA_SYNC4      = 1,
    FILE_SYNC4      = 2
};

struct WRITE4args {
    /* CURRENT_FH: file */
    stateid4        stateid;
    offset4         offset;
    stable_how4     stable;
    opaque          data<>;
};

17.32.3.  RESULTS

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

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

17.32.4.  DESCRIPTION

The WRITE operation is used to write data to a regular file. The target file is specified by the current filehandle. The offset specifies the offset where the data should be written. An offset of 0 (zero) specifies that the write should start at the beginning of the file. The count, as encoded as part of the opaque data parameter, represents the number of bytes of data that are to be written. If the count is 0 (zero), the WRITE will succeed and return a count of 0 (zero) subject to permissions checking. The server may choose to write fewer bytes than requested by the client.
Part of the write request is a specification of how the write is to be performed. The client specifies with the stable parameter the method of how the data is to be processed by the server. If stable is FILE_SYNC4, the server must commit the data written plus all file system metadata to stable storage before returning results. This corresponds to the NFS version 2 protocol semantics. Any other behavior constitutes a protocol violation. If stable is DATA_SYNC4, then the server must commit all of the data to stable storage and enough of the metadata to retrieve the data before returning. The server implementor is free to implement DATA_SYNC4 in the same fashion as FILE_SYNC4, but with a possible performance drop. If stable is UNSTABLE4, the server is free to commit any part of the data and the metadata to stable storage, including all or none, before returning a reply to the client. There is no guarantee whether or when any uncommitted data will subsequently be committed to stable storage. The only guarantees made by the server are that it will not destroy any data without changing the value of verf and that it will not commit the data and metadata at a level less than that requested by the client.

The stateid value for a WRITE request represents a value returned from a previous record lock or share reservation request. The stateid is used by the server to verify that the associated share reservation and any record 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 FILE_SYNC4, 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 NFS version 4 protocol.
service and must be unique between instances of the NFS version 4 protocol server, where uncommitted data may be lost.

If a client writes data to the server with the stable argument set to UNSTABLE4 and the reply yields a committed response of DATA_SYNC4 or UNSTABLE4, the client will follow up some time in the future with a COMMIT operation to synchronize outstanding asynchronous data and metadata with the server’s stable storage, barring client error. It is possible that due to client crash or other error that a subsequent COMMIT will not be received by the server.

For a WRITE with a stateid value of all bits 0, the server MAY allow the WRITE to be serviced subject to mandatory file locks or the current share deny modes for the file. For a WRITE with a stateid value of all bits 1, the server MUST NOT allow the WRITE operation to bypass locking checks at the server and are treated exactly the same as if a stateid of all bits 0 were used.

On success, the current filehandle retains its value.

### 17.32.5. 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 NFS version 4 protocol server over which cached, uncommitted data may be lost. In the most likely case, the verifier allows the client to detect server reboots. This information is required so that the client can safely determine whether the server could have lost cached data. If the server fails unexpectedly and the client has uncommitted data from previous WRITE requests (done with the stable argument set to UNSTABLE4 and in which the result committed was returned as UNSTABLE4 as well) it may not have flushed cached data to stable storage. The burden of recovery is on the client and the client will need to retransmit the data to the server.

A suggested verifier would be to use the time that the server was booted or the time the server was last started (if restarting the server without a reboot results in lost buffers).

The committed field in the results allows the client to do more effective caching. If the server is committing all WRITE requests to stable storage, then it should return with committed set to FILE_SYNC4, regardless of the value of the stable field in the arguments. A server that uses an NVRAM accelerator may choose to implement this policy. The client can use this to increase the effectiveness of the cache by discarding cached data that has already been committed on the server.

Some implementations may return NFS4ERR_NOSPC instead of NFS4ERR_DQUOT when a user’s quota is exceeded. 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 record lock via the LOCK operation before re-attempting the WRITE. When the WRITE completes, the client should release the record 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.

17.33. Operation 40: BACKCHANNEL_CTL - Backchannel control

Control aspects of the backchannel

17.33.1. SYNOPSIS

callback program number, credentials -> -

17.33.2. ARGUMENT

/*
 * NFSv4.1 arguments and results
 */
struct gss_cb_handles4 {
    rpc_gss_svc_t        gcbp_service; /* RFC 2203 */
    opaque               gcbp_handle_from_server<>
    opaque               gcbp_handle_from_client>
};

union callback_sec_parms4 switch (uint32_t cb_secflavor) {
    case AUTH_NONE:
        void;
    case AUTH_SYS:
        authsys_parms cbsp_sys_cred; /* RFC 1831 */
    case RPCSEC_GSS:
        gss_cb_handles4 cbsp_gss_handles;
};

struct BACKCHANNEL_CTL4args {
    uint32_t                bca_cb_program;
    callback_sec_parms4     bca_sec_parms<>
};

17.33.3. RESULT

struct BACKCHANNEL_CTL4res {
    nfsstat4                bcr_status;
};
17.33.4. DESCRIPTION

The BACKCHANNEL_CTL operation replaces the backchannel's callback program number and adds (not replaces) RPCSEC_GSS contexts for use by the callback path.

The arguments and results of the BACKCHANNEL_CTL call are a subset of the CREATE_SESSION parameters and have the same meaning. See the descriptions of csa_cb_program and csa_cb_sec_parms in Section 17.36.5.

BACKCHANNEL_CTL MUST appear in a COMPOUND that starts with SEQUENCE.

17.33.5. ERRORS

TBD

17.34. Operation 41: BIND_CONN_TO_SESSION

17.34.1. SYNOPSIS

sessionid, nonce, digest -> nonce, digest

17.34.2. ARGUMENT

struct bctsa_digest_input4 {
    sessionid4 bdai_sessid;
    uint64_t   bdai_nonce1;
    uint64_t   bdai_nonce2;
};

enum channel_dir_from_client4 {
    CDFC4_FORE   = 0x1,
    CDFC4_BACK   = 0x2,
    CDFC4_FORE_OR_BOTH = 0x3,
    CDFC4_BACK_OR_BOTH = 0x7
};

struct BIND_CONN_TO_SESSION4args {
    sessionid4                      bctsa_sessid;
    bool                            bctsa_step1;
    channel_dir_from_client4        bctsa_dir;
    bool                            bctsa_use_conn_in_rdma_mode;
    uint64_t                        bctsa_nonce;
    opaque                          bctsa_digest<>;
};
17.34.3. RESULT

```c
struct bctsr_digest_input4 {
    sessionid4 bdri_sessid;
    uint64_t   bdri_nonce1;
    uint64_t   bdri_nonce2;
};

enum channel_dir_from_server4 {
    CDFS4_FORE        = 0x1,
    CDFS4_BACK        = 0x2,
    CDFS4_BOTH        = 0x3
};

struct BIND_CONN_TO_SESSION4resok {
    sessionid4                      bctsr_sessid;
    bool                            bctsr_challenge;
    channel_dir_from_server4        bctsr_dir;
    bool                            bctsr_use_conn_in_rdma_mode;
    uint64_t                        bctsr_nonce;
    opaque                          bctsr_digest<>
};
```

union BIND_CONN_TO_SESSION4res switch (nfsstat4 bctsr_status) {
    case NFS4_OK:
        BIND_CONN_TO_SESSION4resok bctsr_resok4;
    default:
        void;
};

17.34.4. DESCRIPTION

BIND_CONN_TO_SESSION is used to bind additional connections to a session. It MUST be used on the connection being bound. It MUST be the only operation in the COMPOUND procedure. Any principal, security flavor, or RPCSEC_GSS context can invoke the operation.

If when the session was created, the client opted to not enable enforcement of connection binding (see Section 17.36), the client is not required to use BIND_CONN_TO_SESSION, unless the client wishes to bind the connection to the backchannel. In that case, because the client did not enable connection binding enforcement, it selected no hash algorithms for digest computation. Thus bctsa_digest and bctsr_digest will be zero length, and the neither the client or server verifies either digest.

If the client enabled enforcement of connection binding, then to
prevent replay attacks, BIND_CONN_TO_SESSION implements a challenge response protocol. This means that the client may be directed to issue BIND_CONN_TO_SESSION a second time on the same connection before the connection is bound to the session. The client is first returned a challenge value in bctsr_nonce, and the client must then calculate a digest using SSV as the key, and the challenge value and other information as the input text. Since the server is free to generate nonce values that are unlikely to be re-used, this prevents attackers from engaging in replay attacks to bind rogue connections to the session.

bctsa_sessid identifies the session the connection is to be bound to.

If bctsa_step1 is TRUE, then the client is trying to initiate a binding of a connection to a session.

bctsa_nonce is a nonce used to deter replay attacks on the server.
If bctsa_step1 is FALSE, bctsa_nonce MUST be different from the bctsa_nonce value for a previous BIND_CONN_TO_SESSION operation that had bctsa_step1 set to TRUE.

bctsa_digest is computed as the output of the HMAC RFC2104 [14] using the current SSV as the key, and the XDR encoded value of data of type bctsa_digest_input4 as the input text.

bdai_sessid is the same as bctsa_sessid. bdai_nonce1 is the same as bctsa_nonce. If bctsa_step1 was TRUE, then bdai_nonce2 is zero. Otherwise, bdai_nonce2 is the same as bctsr_nonce from previous response to BIND_CONN_TO_SESSION on the same connection and sessionid.

In the response, bctsr_challenge is set to TRUE if the server is challenging the client to prove it is not attempting a replay attack. If it is set to true, the client MUST follow up with a BIND_CONN_TO_SESSION request with bctsa_step1 set to FALSE. If bctsr_challenge is set to FALSE, the server is either not challenging the client, or the response is in response to a challenge.

bctsr_nonce, MUST NOT be equal to bctsa_nonce and is a nonce used to deter replay attacks on the client and server.

bctsr_digest is the output of the HMAC using the SSV as the key, and the XDR encoded value of data type bctsr_digest_input as the input text.

bdri_sessid is the same as bctsr_sessid which in turn should be the same as bctsa_sessid. bdri_nonce1 is the same as bctsr_nonce.
bdri_nonce2 is the same as bctsa_nonce. If bctsr_challenge is TRUE, bdri_nonce3 is zero. Otherwise bdri_nonce3 is equal to the value of bctsa_nonce as sent in the preceding BIND_CONN_TO_SESSION that had bctsa_step1 set to TRUE.

If server’s computation of bctsa_digest does not match that in the arguments, the server MUST return NFS4ERR_BAD_SESSION_DIGEST.

bctsa_dir indicates whether the client wants to bind the connection to the fore (operations) channel or back channel or both channels. The value CDFC4_FORE_OR_BOTH indicates the client wants to bind to the both the fore and back channel, but will accept the connection being bound to just the fore channel. The value CDFC4_BACK_OR_BOTH indicates the client wants to bind to the both the fore and back channel, but will accept the connection being bound to the back channel. The server replies in bctsr_dir which channel(s) the connection is bound to (but bctsr_dir is only meaningful if bctsr_challenge is FALSE). If the client specified CDFC4_FORE, the server MUST return CDFS4_FORE. If the client specified CDFC4_BACK, the server MUST return CDFS4_BACK. If the client specified CDFC4_FORE_OR_BOTH, the MUST return CDFS4_FORE or CDFS4_BOTH. If the client specified CDFC4_BACK_OR_BOTH, the server MUST return CDFS4_BACK or CDFS4_BOTH. Note that if BIND_CONN_TO_SESSION has to be called in two steps, the server only processes the bctsa_dir value from the second step, and the client only processes the bctsr_dir from the second step.

See the CREATE_SESSION operation (Section 17.36), and the description of the argument csa_use_conn_in_rdma_mode to understand bctsa_use_conn_in_rdma_mode, and the description of csr_use_conn_in_rdma_mode to understand bctsr_use_conn_in_rdma_mode.

17.34.5. IMPLEMENTATION

If the client’s computation of bctsr_digest does not match that in the results, the client SHOULD NOT accept successful BIND_CONN_TO_SESSION results, and SHOULD assume there has been an attack. Possibilities include:

- The attacker has managed to change the SSV, by binding another connection.

- The attacker has not managed to change the SSV.

The client recovers from a possible attack as follows.

The client can issue SET_SSV to attempt to change the SSV. If SSV is changed successfully, including verification of the digest in the
response to SET_SSV, then this means the attacker did not change the SSV. Thus the attacker has managed to hijack the connection. The client’s only recourse is to disconnect, and bind a new connection. Using IPsec to protect the connection will prevent connection hijacking.

If SET_SSV fails, or the verification of the digest in the response fails, the attacker has changed the SSV. The client’s only recourse is to recreate the session.

If the client loses all connections, it needs to use BIND_CONN_TO_SESSION to bind a new connection. The server will not have the SSV if the server has rebooted and the server doesn’t keep the replay cache in stable storage. In that event, the preceding SEQUENCE op in the same compound will have returned NFS4ERR_BADSESSION, so the client’s state machine goes back to CREATE_SESSION.

There is an issue if SET_SSV is sent, no response is returned, and the last bound connection disconnects. The client, per the sessions model, needs to retry the SET_SSV. But it needs a new connection to do so, and needs to bind that connection to the session. The problem is that the digest calculation for BIND_CONN_TO_SESSION uses the SSV as the key, and the SSV may have changed. While there are multiple recovery strategies, a single, general strategy is described here. First the client reconnects. The client issues BIND_CONN_TO_SESSION with the new SSV used as the digest. If the server returns NFS4ERR_BAD_SESSION_DIGEST then this means the server’s current SSV was not changed, and the SET_SSV was not executed. The client then tries BIND_CONN_TO_SESSION with the old SSV as the digest key. This should not return NFS4ERR_BAD_SESSION_DIGEST. If it does, an implementation error has occurred on either the client or server, and the client has to create a new session.

17.34.6. ERRORS

error list

17.35. Operation 42: EXCHANGE_ID - Instantiate Client ID

Exchange long hand client and server identifiers (owners), and create a client ID

17.35.1. SYNOPSIS

client owner -> client ID, server owner
17.35.2. ARGUMENT

const EXCHGID4_FLAG_SUPP_MOVEDREFER = 0x00000001;
const EXCHGID4_FLAG_SUPP_MOVEDMIGR = 0x00000002;

const EXCHGID4_FLAG_USE_NON_PNFS = 0x00001000;
const EXCHGID4_FLAG_USE_PNFS_MDS = 0x00002000;
const EXCHGID4_FLAG_USE_PNFS_DS = 0x00004000;

struct EXCHANGE_ID4args {
    client_owner4 eia_clientowner;
    uint32_t eia_flags;
    nfs_impl_id4 eia_client_impl_id<1>;
};

17.35.3. RESULT

struct server_owner4 {
    uint64_t so_minor_id;
    opaque so_major_id<NFS4_OPAQUE_LIMIT>;
};

struct EXCHANGE_ID4resok {
    clientid4 eir_clientid;
    sequenceid4 eir_sequenceid;
    uint32_t eir_flags;
    server_owner4 eir_server_owner;
    opaque eir_server_scope<NFS4_OPAQUE_LIMIT>;
    nfs_impl_id4 eir_server_impl_id<1>;
};

union EXCHANGE_ID4res switch (nfsstat4 eir_status) {
    case NFS4_OK:
        EXCHANGE_ID4resok eir_resok4;
    default:
        void;
};

17.35.4. DESCRIPTION

The client uses the EXCHANGE_ID operation to register a particular client owners with the server. The client ID returned from this operation will be necessary for requests that create state on the server and will serve as a parent object to sessions created by the client. In order to confirm the client ID it and the returned sequenceid must first be used as an argument to CREATE_SESSION.
The flags passed as part of the arguments and results to the EXCHANGE_ID operation allow the client and server inform each other of their capabilities as well as indicate how the client ID will be used. Whether a bit is set or cleared on the arguments’ flags does not force the server to set or clear the same bit on the results’ side. Bits not defined above should not be set in the eia_flags field. If they are, the server MUST reject the operation with NFS4ERR_INVAL.

When the EXCHGID4_FLAG_SUPP_MOVEDREFER is set, the client indicates that it is capable of dealing with an NFS4ERR_MOVED error as part of a referral sequence. When this bit is not set, it is still legal for the server to perform a referral sequence. However, a server may use the fact that the client is incapable of correctly responding to a referral, by avoiding it for that particular client. It may, for instance, act as a proxy for that particular file system, at some cost in performance, although it is not obligated to do so. If the server will potentially perform a referral, it MUST set EXCHGID4_FLAG_SUPP_MOVEDREFER in eir_flags.

When the EXCHGID4_FLAG_SUPP_MOVEDMIGR is set, the client indicates that it is capable of dealing with an NFS4ERR_MOVED error as part of a file system migration sequence. When this bit is not set, it is still legal for the server to indicate that a file system has moved, when this in fact happens. However, a server may use the fact that the client is incapable of correctly responding to a migration in its scheduling of file systems to migrate so as to avoid migration of file systems being actively used. It may also hide actual migrations from clients unable to deal with them by acting as a proxy for a migrated file system for particular clients, at some cost in performance, although it is not obligated to do so. If the server will potentially perform a migration, it MUST set EXCHGID4_FLAG_SUPP_MOVEDMIGR in eir_flags.

When EXCHGID4_FLAG_USE_NON_PNFS is set in eia_flags, the client indicates it wants to use the server in a conventional, non-parallel NFS mode of operation. When EXCHGID4_FLAG_USE_NON_PNFS is set in eir_flags, the server is indicating it supports a conventional mode of operation.

When EXCHGID4_FLAG_USE_PNFS_MDS is set in eia_flags, the client indicates it wants to use the server as a metadata server of a parallel NFS cluster. When EXCHGID4_FLAG_USE_PNFS_MDS is set in eir_flags, the server is indicating it supports a metadata server.

When EXCHGID4_FLAG_USE_PNFS_DS is set in eia_flags, the client indicates it wants to use the server as a data server of a parallel NFS cluster. When EXCHGID4_FLAG_USE_PNFS_DS is set in eir_flags, the
A client SHOULD indicate at least one of EXCHGID4_FLAG_USE_NON_PNFS, EXCHGID4_FLAG_USE_PNFS_MDS, and EXCHGID4_USE_PNFS_DS so that a server willing to meet the client’s desires can indicate it is doing so. A server MUST return at least one of the three bits, even if the bit is not among the flag bits sent from the client.

The capabilities indicated in the flags word apply to all sessions created for the resulting client ID and are presumed by the server to remain valid until a new client instance with the same client instance string does an EXCHANGE_ID. The server may update its view of such capabilities when a new EXCHANGE_ID is done by the same client instance but clients should not depend upon such an update being effective until the server receives an EXCHANGE_ID for a new client instance.

The arguments includes an array of up to one element in length called eia_client_impl_id. If eia_client_impl_id is present it contains the information identifying the implementation of the client. Similarly, the results include an array of up to one element in length called eir_server_impl_id that identifies the implementation of the server.Servers MUST allow a zero length eia_client_impl_id array, and clients MUST allow a zero length eir_server_impl_id array. Being able to identify specific implementations can help in planning by administrators or implementors. For example, diagnostic software may extract this information in an attempt to identify interoperability problems, performance workload behaviors or general usage statistics. Since the intent of having access to this information is for planning or general diagnosis only, the client and server MUST NOT interpret this implementation identity information in a way that affects interoperational behavior of the implementation. The reason is the if clients and servers did such a thing, they might use fewer capabilities of the protocol than the peer can support, or the client and server might refuse to interoperate.

Because it is likely some implementations will violate the protocol specification and interpret the identity information, implementations MUST allow the users of the NFSv4 client and server to set the contents of the sent nfs_impl_id structure to any value.

17.35.5. IMPLEMENTATION

A server’s client record is a 5-tuple:

1. co_ownerid
The client identifier string, from the eia_clientowner structure of the EXCHANGE_ID4args structure

2.  co_verifier:

A client-specific value used to indicate reboots, from the eia_clientowner structure of the EXCHANGE_ID4args structure

3.  principal:

The RPCSEC_GSS principal sent via the RPC headers

4.  client ID:

The shorthand client identifier, generated by the server and returned via the eir_clientid field in the EXCHANGE_ID4resok structure

5.  confirmed:

A private field on the server indicating whether or not a client record has been confirmed. A client record is confirmed if there has been a successful CREATE_SESSION operation to confirm it. Otherwise it is unconfirmed. An unconfirmed record is established by a EXCHANGE_ID call. Any unconfirmed record that is not confirmed within a lease period may be removed.

The following identifiers represent special values for the fields in the records.

ownerid_arg:

The value of the eia_clientowner.co_ownerid subfield of the EXCHANGE_ID4args structure of the current request.

verifier_arg:

The value of the eia_clientowner.co_verifier subfield of the EXCHANGE_ID4args structure of the current request.

old_verifier_arg:

A value of the eia_clientowner.co_verifier field of a client record received in a previous request; this is distinct from verifier_arg.
principal_arg:
The value of the RPCSEC_GSS principal for the current request.

old_principal_arg:
A value of the RPCSEC_GSS principal received for a previous request. This is distinct from principal_arg.

clientid_ret:
The value of the eir_clientid field the server will return in the EXCHANGE_ID4resok structure for the current request.

old_clientid_ret:
The value of the eir_clientid field the server returned in the EXCHANGE_ID4resok structure for a previous request. This is distinct from clientid_ret.

Since EXCHANGE_ID is a non-idempotent operation, we must consider the possibility that replays might occur as a result of a client reboot, network partition, malfunctioning router, etc. Replays are identified by the value of the client field of EXCHANGE_ID4args and the method for dealing with them is outlined in the scenarios below.

The scenarios are described in terms of what client records whose eia_clientowner.co_ownerid subfield have a value equal to ownerid_arg existing in the server’s set of client records. Any cases in which there is more than one record with identical values for ownerid_arg represent a server implementation error. Operation in the potential valid cases is summarized as follows.

1. Common case

   If no client records with eia_clientowner.co_ownerid matching ownerid_arg exist, a new shorthand client identifier clientid_ret is generated, and the following unconfirmed record is added to the server’s state.

   { ownerid_arg, verifier_arg, principal_arg, clientid_ret, FALSE }

   Subsequently, the server returns clientid_ret.

2. Router Replay
If the server has the following confirmed record, then this request is likely the result of a replayed request due to a faulty router or lost connection.

\{ ownerid_arg, verifier_arg, principal_arg, clientid_ret, TRUE \}

Since the record has been confirmed, the client must have received the server’s reply from the initial EXCHANGE_ID request. Since this is simply a spurious request, there is no modification to the server’s state, and the server makes no reply to the client.

3. Client Collision

If the server has the following confirmed record, then this request is likely the result of a chance collision between the values of the eia_clientowner.co_ownerid subfield of EXCHANGE_ID4args for two different clients.

\{ ownerid_arg, *, old_principal_arg, clientid_ret, TRUE \}

Since the value of the eia_clientowner.co_ownerid subfield of each client record must be unique, there is no modification of the server’s state. The server either returns NFS4ERR_CLID_INUSE is to indicate the client should retry with a different value for the eia_clientowner.co_ownerid subfield of EXCHANGE_ID4args, or the server considers the principal and ownerid together as the client owner, and treats the EXCHANGE_ID as a unique client owner.

This scenario may also represent a malicious attempt to destroy a client’s state on the server. For security reasons, the server MUST NOT remove the client’s state when there is a principal mismatch.

4. Replay

If the server has the following unconfirmed record then this request is likely the result of a client replay due to a network partition or some other connection failure.

\{ ownerid_arg, verifier_arg, principal_arg, clientid_ret, FALSE \}

Since the response to the EXCHANGE_ID request that created this record may have been lost, it is not acceptable to drop this replayed request. However, rather than processing it
normally, the existing record is left unchanged and
clientid_ret, which was generated for the previous request, is
returned.

5. Change of Principal

If the server has the following unconfirmed record then this
request is likely the result of a client which has for
whatever reasons changed principals (possibly to change
security flavor) after calling EXCHANGE_ID, but before calling
CREATE_SESSION.

{ ownerid_arg, verifier_arg, old_principal_arg, clientid_ret,
FALSE}

Since the client has not changed, the principal field of the
unconfirmed record is updated to principal_arg and
clientid_ret is again returned. There is a small possibility
that this is merely a collision on the client field of
EXCHANGE_ID4args between unrelated clients, but since that is
unlikely, and an unconfirmed record does not generally have
any file system pertinent state, we can assume it is the same
client without risking loss of any important state.

After processing, the following record will exist on the
server.

{ ownerid_arg, verifier_arg, principal_arg, clientid_ret,
FALSE}

6. Client Reboot

If the server has the following confirmed client record, then
this request is likely from a previously confirmed client
which has rebooted.

{ ownerid_arg, old_verifier_arg, principal_arg, clientid_ret,
TRUE }

Since the previous incarnation of the same client will no
longer be making requests, lock and share reservations should
be released immediately rather than forcing the new
incarnation to wait for the lease time on the previous
incarnation to expire. Furthermore, session state should be
removed since if the client had maintained that information
across reboot, this request would not have been issued. If
the server does not support the CLAIM_DELEGATE_PREV claim
type, associated delegations should be purged as well;
otherwise, delegations are retained and recovery proceeds according to the section Delegation Recovery (Section 9.2.1). The client record is updated with the new verifier and its status is changed to unconfirmed.

After processing, clientid_ret is returned to the client and the following record will exist on the server.

{ ownerid_arg, verifier_arg, principal_arg, clientid_ret, FALSE }

7. Reboot before confirmation

If the server has the following unconfirmed record, then this request is likely from a client which rebooted before sending a CREATE_SESSION request.

{ ownerid_arg, old_verifier_arg, *, clientid_ret, FALSE }

Since this is believed to be a request from a new incarnation of the original client, the server updates the value of eia_clientowner.co_verifier and returns the original clientid_ret. After processing, the following state exists on the server.

{ ownerid_arg, verifier_arg, *, clientid_ret, FALSE }

In addition to the client ID and sequenceid, the server returns a server owner (eir_server_owner) and eir_server_scope. The former field is used for network trunking as described in Section 2.10.3.4.1. The latter field is used to allow clients to determine when clientids issued by one server may be recognized by another in the event of file system migration (see Section 10.6.7).

17.36. Operation 43: CREATE_SESSION - Create New Session and Confirm Client ID

Start up session and confirm client ID.

17.36.1. SYNOPSIS

client ID, session_args -> sessionid, session_args
17.36.2. ARGUMENT

struct channel_attrs4 {
    count4 ca_maxrequestsize;
    count4 ca_maxresponsesize;
    count4 ca_maxresponsesize_cached;
    count4 ca_maxoperations;
    count4 ca_maxrequests;
    uint32_t ca_rdma_ird<1>;
};

union conn_binding4args switch (bool cba_enforce) {
    case TRUE:
        sec_oid4 cba_hash_algs<>;
    case FALSE:
        void;
};

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

struct CREATE_SESSION4args {
    clientid4 csa_clientid;
    sequenceid4 csa_sequence;
    uint32_t csa_flags;
    count4 csa_headerpadsize;
    conn_binding4args csa_conn_binding_opts;
    channel_attrs4 csa_fore_chan_attrs;
    channel_attrs4 csa_back_chan_attrs;
    uint32_t csa_cb_program;
    callback_sec_parms4 csa_cb_sec_parms<>;
};
17.36.3. RESULT

```
struct hash_alg_info4 {
    uint32_t                hai_hash_alg;
    uint32_t                hai_ssv_len;
};

union conn_binding4res switch (bool cbr_enforce) {
    case TRUE:
        hash_alg_info4          cbr_hash_alg_info;
    case FALSE:
        void;
};

struct CREATE_SESSION4resok {
    sessionid4              csr_sessionid;
    sequenceid4             csr_sequence;
    uint32_t                csr_flags;
    count4                  csr_headerpadsize;
    conn_binding4res        csr_conn_binding_opts;
    channel_attrs4          csr_fore_chan_attrs;
    channel_attrs4          csr_back_chan_attrs;
};
```

union CREATE_SESSION4res switch (nfsstat4 csr_status) {
    case NFS4_OK:
        CREATE_SESSION4resok    csr_resok4;
    default:
        void;
};

17.36.4. DESCRIPTION

This operation is used by the client to create new session objects on the server. The server MUST accept a CREATE_SESSION operation with no preceding SEQUENCE operation in the COMPOUND procedure. A client MAY precede CREATE_SESSION with SEQUENCE in a COMPOUND procedure; if so, any session created by CREATE_SESSION has no direct relation to the session specified in the SEQUENCE operation.

In addition to creating a session, CREATE_SESSION has the following effects:
The first session created with a new shorthand client identifier (client ID) serves to confirm the creation of that client’s state on the server. The server returns the parameter values for the new session.

The connection CREATE_SESSION is issued over is bound to the session and to the session’s forward channel.

17.36.5. IMPLEMENTATION

To describe the implementation, the same notation for client records introduced in the description of EXCHANGE_ID is used with the following addition:

clientid_arg: The value of the csa_clientid field of the CREATE_SESSION4args structure of the current request.

Since CREATE_SESSION is a non-idempotent operation, we must consider the possibility that replays may occur as a result of a client reboot, network partition, malfunctioning router, etc. For each client ID created by EXCHANGE_ID, the server maintains a separate replay cache similar to the session replay cache used for SEQUENCE operations, with two distinctions. First this is a replay cache just for detecting and processing CREATE_SESSION requests for a given client ID. Second, the size of the client ID replay cache is of one slot (and as a result, the CREATE_SESSION request does not carry a slot number). This means that at most one CREATE_SESSION request for a given client ID can be outstanding. When client issues a successful EXCHANGE_ID it is returned eir_sequenceid, and the client is expected to set the value of csa_sequenceid in the next CREATE_SESSION it sends with that client ID to the value of eir_sequenceid. After EXCHANGE_ID, the server initializes the client ID slot to be equal to eir_sequenceid - 1 (accounting for underflow), and records a contrived CREATE_SESSION result with a "cached" result of NFS4ERR_SEQ_MISORDERED. With the slot thus initialized, the processing of the CREATE_SESSION operation is divided into four phases:

1. Replay cache lookup. The server verifies it has a replay cache for the client ID. If the server contains no records with client ID equal to clientid_arg, then most likely the client’s state has been purged during a period of inactivity, possibly due to a loss of connectivity. NFS4ERR_STALE_CLIENTID is returned, and no changes are made to any client records on the server.

2. Sequenceid processing. If csa_sequenceid is equal to the sequenceid in the client’s slot, then this is a replay of the previous CREATE_SESSION request, and the server returns the
cached result. If csa_sequenceid is not equal to the sequenceid in the slot, and is more than one greater (accounting for wraparound), then the server returns the error NFS4ERR_SEQ_MISORDERED, and does not change the slot. If csa_sequenceid is equal to the slot’s sequenceid + 1 (accounting for wraparound), then the slot’s sequenceid is set to csa_sequenceid, and the CREATE_SESSION processing goes to the next phase. A subsequent new CREATE_SESSION call, MUST use a csa_sequence that is one greater than that recorded in the slot.

3. Client ID confirmation. In case the state for the provided client ID has not been verified, it is confirmed before the session is created. Otherwise the client ID confirmation phase is skipped and only the session creation phase occurs. The operational cases are described in terms of what client records whose client ID field have value equal to clientid_arg exist in the server’s set of client records. Any cases in which there is more than one record with identical values for client ID represent a server implementation error. Operation in the potential valid cases is summarized as follows.

* Common Case

   If the server has the following unconfirmed record, then this is the expected confirmation of an unconfirmed record.

   { *, *, principal_arg, clientid_arg, FALSE }

   The confirmed field of the record is set to TRUE.

   The processing of the operation continues to session creation.

* Principal Change or Collision

   If the server has the following record, then the client has changed principals after the previous EXCHANGE_ID request, or there has been a chance collision between shorthand client identifiers.

   { *, *, old_principal_arg, clientid_arg, *, sequence_arg }

   Neither of these cases are permissible. Processing stops and NFS4ERR_CLID_INUSE is returned to the client. No changes are made to any client records on the server.
4. Session creation. The server confirmed the client ID, either in this CREATE_SESSION operation, or a previous CREATE_SESSION operation. The server examines the remaining fields of the arguments. For each argument field, if the value is acceptable to the server, it is recommended that the server use the provided value to create the new session. If it is not acceptable, the server may use a different value, but must return the value used to the client. These parameters have the following interpretation.

csa_flags:

The csa_flags field contains a list of the following flag bits:

CREATE_SESSION4_FLAG_PERSIST:

If CREATE_SESSION4_FLAG_PERSIST is set, the client desires server support for "reliable" semantics. For sessions in which only idempotent operations will be used (e.g. a read-only session), clients should not set CREATE_SESSION4_FLAG_PERSIST. If the server does not or cannot provide "reliable" semantics the result field csr_flags must not set CREATE_SESSION4_FLAG_PERSIST.

If the server is a pNFS metadata server, for reasons described in Section 12.5.2 it MUST support CREATE_SESSION4_FLAG_PERSIST if it supports the layout_hint (Section 5.13.4) attribute.

CREATE_SESSION4_FLAG_CONN_BACK_CHAN:

If CREATE_SESSION4_FLAG_CONN_BACK_CHAN is set in csa_flags, the client is requesting that the server use the connection CREATE_SESSION is called over for the back channel as well as the forward channel. The server sets CREATE_SESSION4_FLAG_CONN_BACK_CHAN in the result field csr_flags if it agrees. If CREATE_SESSION4_FLAG_CONN_BACK_CHAN is not set in csa_flags, then CREATE_SESSION4_FLAG_CONN_BACK_CHAN MUST NOT be set in csr_flags.

CREATE_SESSION4_FLAG_CONN_RDMA:

If CREATE_SESSION4_FLAG_CONN_RDMA is set in csa_flags, the connection CREATE_SESSION is called over is currently in non-RDMA mode, but has the capability to operate in RDMA mode, and the client is requesting the server agree to
"step up" to RDMA mode on the connection. The server sets CREATE_SESSION4_FLAG_CONN_RDMA in the result field csr_flags if it agrees. If CREATE_SESSION4_FLAG_CONN_RDMA is not set in csa_flags, then CREATE_SESSION4_FLAG_CONN_RDMA MUST NOT be set in csr_flags. Note that once the server agrees to step up, it and the client MUST exchange all future traffic on the connection with RPC RDMA framing and not Record Marking.

[c[Comment.18: add xref]]

csa_headerpadsize:

The maximum amount of padding the client is willing to apply to ensure that write payloads are aligned on some boundary at the server. The server should reply in csr_headerpadsize with its preferred value, or zero if padding is not in use. The server may decrease this value but MUST NOT increase it.

csa_conn_binding_opts:

This argument indicates whether the client wants the server to enforce connection binding (see Section 2.10.6.3), and if so, which one way hash algorithms to use. The corresponding result is csr_conn_binding_opts. The argument contains the following fields.

cba_enforce:

Clients SHOULD set cba_enforce to TRUE so that servers reject the use of connections that are not explicitly bound to the session. If TRUE, the server MUST require the client to issue BIND_CONN_TO_SESSION before using a connection on a channel. If FALSE, then the digests used in SET_SSV and BIND_CONN_TO_SESSION MUST be zero length.

The corresponding result is cbr_enforce which MUST be equal to cba_enforce.

cba_hash_algs:

This is the set of algorithms the client supports for the purpose of computing the digests needed for the SET_SSV and BIND_CONN_TO_SESSION operations. Each algorithm is specified as an object identifier (OID). The REQUIRED algorithms for a server are id-sha1, id-sha224, id-sha256, id-sha384, and id-sha512 RFC4055 [15].
If the server does not support any of the offered hash algorithms, CREATE_SESSION fails with error status NFS4ERR_OP_HASH_ALG_UNSUPP. Otherwise, the corresponding result is cbr_hash_alg_info, which contains two fields, hai_hash_alg and hai_ssv_len. The former is the index of the algorithm list of cba_hash_algs that the server has selected and the client MUST use for SET_SSV and BIND_CONN_TO_SESSION. The latter is the length in octets of the SSV the client MUST use in SET_SSV. The result hai_ssv_len MUST be greater than or equal to the length of the hash produced by the selected algorithm.

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore_chan_attrs

csa_back_chan_attrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chan_attrs and csr_back_chan_attrs. Each structure has the following fields:

csa_fore-chan_atrrs

csa_back-chan_atrrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chan_atrrs and csr_back-chan_atrrs. Each structure has the following fields:

csa_fore_chanattrs

csa_back_chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chanattrs and csr_back_chanattrs. Each structure has the following fields:

csa_fore_chanattrs

csa_back_chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore_chanattrs and csr_back_chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:

csa_fore-chanattrs

csa_back-chanattrs

These two fields apply to attributes of the fore channel (aka the operations channel, which conveys requests originating from the client to the server), and the back channel (the channel that conveys callback requests originating from the server to the client). The results are in corresponding structures called csr_fore-chanattrs and csr_back-chanattrs. Each structure has the following fields:
the description in Section 2.10.4.4.

ca_maxresponsesize_cached:

Like ca_maxresponsesize, but the maximum size of a reply that will be stored in the reply cache (Section 2.10.4.1). If ca_maxresponsesize Cached is less than ca_maxresponsesize, then this is an indication to the client that it needs to be selective about which replies it tells the server to cache; large replies (e.g. READ results), should not be cached. The client can decide which replies to cache via the SEQUENCE (Section 17.46) or CB_SEQUENCE (Section 19.9) operations. If a sender sends a request for which the size of the reply would exceed this value, the receiver will return NFS4ERR_REP_TOO_BIG_TO_CACHE, per the description in Section 2.10.4.4.

ca_maxoperations:

The maximum number of operations requests the receiver will accept in a COMPOUND or CB_COMPOUND. If client or server do not have a limit, they will set ca_maxoperations to 0xffffffff. The server MUST NOT increase ca_maxoperations in the reply to CREATE_SESSION. If the requester issues a COMPOUND or CB_COMPOUND with more operations than ca_maxoperations, the replier MUST return NFS4ERR_TOO_MANY_OPS.

ca_maxrequests:

The maximum number of concurrent COMPOUND or CB_COMPOUND requests the sender will issue on the session. Subsequent requests will each be assigned a slot identifier by the client within the range 0 to ca_maxrequests - 1 inclusive.

csa_rdma_ird:

This array has a maximum of one element. If this array has one element, then the element contains the inbound RDMA read queue depth (IRD).

csa_cb_program

This is the program number the server must use in any callbacks sent through the back channel to the client.
csa_cb_sec_parms

This is an array of acceptable security credentials. Three security flavors are supported: AUTH_NONE, AUTH_SYS, and RPCSEC_GSS. If AUTH_NONE is specified for a credential, then this says the client is allowed to use AUTH_NONE on all callbacks for the session. If AUTH_SYS is specified, then the client is allowed to use AUTH_SYS on all callbacks, using the credential specified cbsp_sys_cred. If RPCSEC_GSS is specified, then the server is allowed to use the RPCSEC_GSS context specified in cbsp_gss_parms as the RPCSEC_GSS context in the credential of the RPC header of callbacks to the client.

The RPCSEC_GSS context is specified with two RPCSEC_GSS handles. The first handle, gcbp_handle_from_server, is the fore handle the server returned to the client when the RPCSEC_GSS context was created on the server. The second handle, gcbp_handle_from_client is the back handle the client will map to the RPCSEC_GSS context to. The server can immediately use the RPCSEC_GSS context using gcbp_handle_from_client as the value for "handle" in the structure rpc_gss_cred_vers_1_t of the RPCSEC_GSS handle, and gss_proc set to RPCSEC_GSS_DATA. Note that while the GSS context state is shared between the fore and back RPCSEC_GSS contexts, the fore and back RPCSEC_GSS context state are independent of each other as far as the RPCSEC_GSS sequence number.

Implementing RPCSEC_GSS callback support requires the client and server change their RPCSEC_GSS implementations. One possible set of changes includes:

+ Adding a data structure that wraps the GSS-API context with a reference count.

+ New functions to increment and decrement the reference count. If the reference count is decremented to zero, the wrapper data structure and the GSS-API context it refers to would be freed.

+ Change RPCSEC_GSS to create the wrapper data structure upon receiving GSS-API context from gss_accept_sec_context() and gss_init_sec_context(). The reference count would be initialized to 1.

+ Adding a function to map an existing RPCSEC_GSS handle to a pointer to the wrapper data structure. The reference count
would be incremented.

+ Adding a function to create a new RPCSEC_GSS handle from a pointer to the wrapper data structure. The reference count would be incremented.

+ Replacing calls from RPCSEC_GSS that free GSS-API contexts, with calls to decrement the reference count on the wrapper data structure.

5. The server creates the session by recording the parameter values used (including whether the CREATE_SESSION4_FLAG_PERSIST flag is set and has been accepted by the server) and allocating space for the session replay cache. For each slot in the replay cache, the server sets the sequenceid to zero (0), and records a result containing a result for a COMPOUND with a single SEQUENCE operation, with the cached error of NFS4ERR_SEQ_MISORDERED. Thus, the first SEQUENCE operation a client issues on a slot after the session is created MUST start with a sequenceid of one (1). The client initializes its replay cache for receiving callbacks in the same way, and similarly, the first CB_SEQUENCE operation on a slot after session creation must have a sequenceid of one.

6. If the session state is created successfully, the server associates the session with the client ID provided by the client.

17.37. Operation 44: DESTROY_SESSION - Destroy existing session

Destroy existing session.

17.37.1. SYNOPSIS

    sessionid -> status

17.37.2. ARGUMENT

    struct DESTROY_SESSION4args {
        sessionid4 dsa_sessionid;
    };

17.37.3. RESULT

    struct DESTROY_SESSION4res {
        nfsstat4 dsr_status;
    };

17.37.4. DESCRIPTION

The DESTROY_SESSION operation closes the session and discards the
replay cache. Any remaining connections bound to the session are
immediately unbound and may additionally be closed by the server.
Locks, delegations, layouts, wants, and the lease, which are all tied
to the client ID, are not affected by DESTROY_SESSION.

If the COMPOUND request starts with SEQUENCE, then DESTROY_SESSION
MUST be the final, or only operation, unless the sessionid specified
in SEQUENCE is different from the sessionid specified in
DESTROY_SESSION. DESTROY_SESSION MAY be the only operation in a
COMPOUND request. Because the operation results in destruction of
the session, any reply caching for this request, as well as
previously completed requests, will be lost. For this reason, it is
advisable to not place this operation in a COMPOUND request with
other state-modifying operations (unless those operations are for a
different session, as specified by SEQUENCE).

Because the session is destroyed, a client that retransmits the
request may receive an error in response, even though the original
request was successful.

If there is a backchannel on the session and the server has
outstanding CB_SEQUENCE operations, then the server MAY refuse to
destroy the session and return NFS4ERR_BACK_CHAN_BUSY. The client
SHOULD respond to all outstanding CB_COMPOUNDS before re-issuing
DESTROY_SESSION.

17.37.5. IMPLEMENTATION

No discussion at this time.

17.38. Operation 45: FREE_STATEID - Free stateid with no locks

Test a series of stateids for validity.

17.38.1. SYNOPSIS

    stateid ->

17.38.2. ARGUMENT

    struct FREE_STATEID4args {
        stateid4    fsa_stateid;
    };
17.38.3.  RESULT

struct FREE_STATEID4res {
    nfsstat4 fsr_status;
};

17.38.4.  DESCRIPTION

The FREE_STATEID operation is used to free a stateid which no longer has any associated locks (including opens, record locks, delegations, layouts). This may be cause of client unlock operations or because of server revocation. If there are valid locks (of any kind) associated with the stateid in question, the error NFS4ERR_LOCKS_HELD will be returned, and the associated stateid will not be freed.

When a stateid is freed which had been associated with revoked locks, the client, by doing the FREE_STATEID acknowledges the loss of those locks, allowing the server, once all such revoked state, is acknowledged to allow that client again to reclaim locks, without encountering the edge conditions discussed in Section 8.6.2.

Once a successful FREE_STATEID is done for a given stateid, any subsequent use of that stateid will result in an NFS4ERR_BAD_STATEID error.

17.38.5.  IMPLEMENTATION

No discussion at this time.

17.39.  Operation 46: GET_DIR_DELEGATION - Get a directory delegation

Obtain a directory delegation.

17.39.1.  SYNOPSIS

(cfh), requested notification ->
    (cfh), cookieverf, stateid, supported notification
17.39.2. ARGUMENT

/*
 * Notification types.
 */
const DIR_NOTIFICATION4_NONE = 0x00000000;
const DIR_NOTIFICATION4_CHANGE_CHILD_ATTRIBUTES = 0x00000001;
const DIR_NOTIFICATION4_CHANGE_DIR_ATTRIBUTES = 0x00000002;
const DIR_NOTIFICATION4_REMOVE_ENTRY = 0x00000004;
const DIR_NOTIFICATION4_ADD_ENTRY = 0x00000008;
const DIR_NOTIFICATION4_RENAME_ENTRY = 0x00000010;
const DIR_NOTIFICATION4_CHANGE_COOKIE_VERIFIER = 0x00000020;

typedef uint32_t dir_notification_type4;

typedef nfstime4 attr_notice4;

struct GET_DIR_DELEGATION4args {
    bool gdda_signal_deleg_avail;
    dir_notification_type4 gdda_notification_type;
    attr_notice4 gdda_child_attr_delay;
    attr_notice4 gdda_dir_attr_delay;
    bitmap4 gdda_child_attributes;
    bitmap4 gdda_dir_attributes;
};
17.39.3. RESULT

struct GET_DIR_DELEGATION4resok {
    verifier4         gddr_cookieverf;
    /* Stateid for get_dir_delegation */
    stateid4         gddr_stateid;
    /* Which notifications can the server support */
    dir_notification_type4 gddr_notification;
    bitmap4          gddr_child_attributes;
    bitmap4          gddr_dir_attributes;
};

denum gddrnf4_status {
    GDD4_OK         = 0,
    GDD4_UNAVAIL    = 1
};

union GET_DIR_DELEGATION4res_non_fatal
    switch (gddrnf4_status gddrnf_status) {
    case GDD4_OK:
        GET_DIR_DELEGATION4resok        gddrnf_resok4;
    case GDD4_UNAVAIL:
        bool                            gddrnf_will_signal_deleg_avail;
    }

union GET_DIR_DELEGATION4res
    switch (nfsstat4 gddr_status) {
    case NFS4_OK:
        /* CURRENT_FH: delegated dir */
        GET_DIR_DELEGATION4res_non_fatal        gddr_res_non_fatal4;
    default:
        void;
    }

17.39.4. DESCRIPTION

The GET_DIR_DELEGATION operation is used by a client to request a directory delegation. The directory is represented by the current filehandle. The client also specifies whether it wants the server to notify it when the directory changes in certain ways by setting one or more bits in a bitmap. The server may also choose not to grant the delegation. In that case the server will return NFS4ERR_DIRDELEG_UNAVAIL. If the server decides to hand out the delegation, it will return a cookie verifier for that directory. If the cookie verifier changes when the client is holding the delegation, the delegation will be recalled unless the client has asked for notification for this event. In that case a notification
will be sent to the client.

The server will also return a directory delegation stateid in addition to the cookie verifier as a result of the GET_DIR_DELEGAUTION operation. This stateid will appear in callback messages related to the delegation, such as notifications and delegation recalls. The client will use this stateid to return the delegation voluntarily or upon recall. A delegation is returned by calling the DELEGRETURN operation.

The server may not be able to support notifications of certain events. If the client asks for such notifications, the server must inform the client of its inability to do so as part of the GET_DIR_DELEGATION reply by not setting the appropriate bits in the supported notifications bitmask contained in the reply.

The GET_DIR_DELEGATION operation can be used for both normal and named attribute directories. It covers all the entries in the directory except the ".." entry. That means if a directory and its parent both hold directory delegations, any changes to the parent will not cause a notification to be sent for the child even though the child’s ".." entry points to the parent.

If client sets gdda_signal_deleg_avail to TRUE, then it is registering with the client a "want" for a directory delegation. If the server supports and will honor the "want", the results will have gddrnf_will_signal_deleg_avail set to TRUE. If so the client should expect a future CB_RECALLABLE_OBJ_AVAIL operation to indicate that a directory delegation is available.

17.39.5. IMPLEMENTATION

Directory delegation provides the benefit of improving cache consistency of namespace information. This is done through synchronous callbacks. A server must support synchronous callbacks in order to support directory delegations. In addition to that, asynchronous notifications provide a way to reduce network traffic as well as improve client performance in certain conditions. Notifications would not be requested when the goal is just cache consistency.

Notifications are specified in terms of potential changes to the directory. A client can ask to be notified of events by setting one or more flags in gdda_notification_type. The client can ask for notifications on addition of entries to a direction (by setting the DIR_NOTIFICATION4_ADD_ENTRY in gdda_notification_type), notifications on entry removal (DIR_NOTIFICATION4_REMOVE_ENTRY), renames (DIR_NOTIFICATION4_RENAME_ENTRY), directory attribute changes
(DIR_NOTIFICATION4_CHANGE_DIR_ATTRIBUTES), and cookie verifier changes (DIR_NOTIFICATION4_CHANGE_COOKIE_VERIFIER) by setting one more corresponding flags in the gdda_notification_type field.

The client can also ask for notifications of changes to attributes of directory entries (DIR_NOTIFICATION4_CHANGE_CHILD_ATTRIBUTES) in order to keep its attribute cache up to date. However any changes made to child attributes do not cause the delegation to be recalled. If a client is interested in directory entry caching, or negative name caching, it can set the gdda_notification_type appropriately and the server will notify it of all changes that would otherwise invalidate its name cache. The kind of notification a client asks for may depend on the directory size, its rate of change and the applications being used to access that directory. However, the conditions under which a client might ask for a notification, is out of the scope of this specification.

For attribute notifications, the client will set bits in the gdda_dir_attributes bitmap to indicate which attributes it wants to be notified of. If the server does not support notifications for changes to a certain attribute, it should not set that attribute in the supported attribute bitmap specified in the reply (gddr_dir_attributes). The client will also set in the gdda_child_attributes bitmap the attributes of directory entries it wants to be notified of, and the server will indicate in gddr_child_attributes which attributes of directory entries it will notify the client of.

The client will also let the server know if it wants to get the notification as soon as the attribute change occurs or after a certain delay by setting a delay factor; gdda_child_attr_delay is for attribute changes to directory entries and gdda_dir_attr_delay is for attribute changes to the directory. If this delay factor is set to zero, that indicates to the server that the client wants to be notified of any attribute changes as soon as they occur. If the delay factor is set to N seconds, the server will make a best effort guarantee that attribute updates are not out of sync by more than that. If the client asks for a delay factor that the server does not support or that may cause significant resource consumption on the server by causing the server to send a lot of notifications, the server should not commit to sending out notifications for attributes and therefore must not set the appropriate bit in the gddr_child_attributes and gddr_dir_attributes bitmaps in the response.

The client should use a security flavor that the file system is exported with. If it uses a different flavor, the server should return NFS4ERR_WRONGSEC to the operation that precedes
GET_DIR_DELEGATION and sets the current filehandle.

17.40. Operation 47: GETDEVICEINFO - Get Device Information

17.40.1. SYNOPSIS

(cfh), device_id, layout_type, maxcount -> device_addr

17.40.2. ARGUMENT

struct GETDEVICEINFO4args {
    /* CURRENT_FH: file */
    deviceid4               gdia_device_id;
    layouttype4             gdia_layout_type;
    count4                  gdia_maxcount;
};

17.40.3. RESULT

struct GETDEVICEINFO4resok {
    device_addr4            gdir_device_addr;
};

union GETDEVICEINFO4res switch (nfsstat4 gdir_status) {
    case NFS4_OK:
        GETDEVICEINFO4resok     gdir_resok4;
    default:
        void;
};

17.40.4. DESCRIPTION

Returns device address information for a specified device. The device address MUST correspond to the layout type specified by the GETDEVICEINFO4args. The current filehandle (cfh) is used to identify the file system; device IDs are unique per file system (FSID) and are qualified by the layout type.

See Section 12.2.12 for more details on device ID assignment.

If the size of the device address exceeds gdia_maxcount bytes, the metadata server will return the error NFS4ERR_TOOSMALL. If an invalid device ID is given, the metadata server will respond with NFS4ERR_INVAL.
17.40.5. IMPLEMENTATION

17.41. Operation 48: GETDEVICELIST

17.41.1. SYNOPSIS

(cfh), layout_type, maxcount, cookie, cookieverf ->
cookie, cookieverf, device info list

17.41.2. ARGUMENT

struct GETDEVICELIST4args {
  /* CURRENT_FH: file */
  layout_type4             gdla_layout_type;
  count4                  gdla_maxcount;
  nfs_cookie4             gdla_cookie;
  verifier4               gdla_cookieverf;
};

17.41.3. RESULT

struct GETDEVICELIST4resok {
  nfs_cookie4             gdlr_cookie;
  verifier4               gdlr_cookieverf;
  devlist_item4           gdlr_devinfo_list<>
  bool                    gdlr_eof;
};

union GETDEVICELIST4res switch (nfsstat4 gdlr_status) {
  case NFS4_OK:
    GETDEVICELIST4resok     gdlr_resok4;
  default:
    void;
};

17.41.4. DESCRIPTION

In some applications, especially SAN environments, it is convenient
to find out about all the devices associated with a file system.
This lets a client determine if it has access to these devices, e.g.,
at mount time.

This operation returns an array of items (devlist_item4) that
establish the association between the short deviceid4 and the
addressing information for that device, for a particular layout type.
This operation may not be able to fetch all device information at...
once, thus it uses a cookie based approach, similar to READDIR, to fetch additional device information (see Section 17.23). The "eof" flag has a value of TRUE if there are no more entries to fetch. As in GETDEVICEINFO, the current filehandle (cfh) is used to identify the file system.

As in GETDEVICEINFO, gdla_maxcount specifies the maximum number of bytes to return. If the metadata server is unable to return a single device address, it will return the error NFS4ERR_TOSMALL. If an invalid device ID is given, the metadata server will respond with NFS4ERR_INVAL.

17.41.5. IMPLEMENTATION

17.42. Operation 49: LAYOUTCOMMIT - Commit writes made using a layout

17.42.1. SYNOPSIS

(client ID), (cfh), offset, length, reclaim, last_write_offset, time_modify, time_access, layoutupdate -> newsize

17.42.2. ARGUMENT

union newtime4 switch (bool nt_timechanged) {
  case TRUE:
    nfstime4          nt_time;
  case FALSE:
    void;
};

union newoffset4 switch (bool no_newoffset) {
  case TRUE:
    offset4          no_offset;
  case FALSE:
    void;
};

struct LAYOUTCOMMIT4args {
  /* CURRENT_FH: file */
  offset4                 loca_offset;
  length4                 loca_length;
  bool                    loca_reclaim;
  newoffset4              loca_last_write_offset;
  newtime4                loca_time_modify;
  newtime4                loca_time_access;
  layoutupdate4           loca_layoutupdate;
};
17.42.3. RESULT

union newsize4 switch (bool ns_sizechanged) {
  case TRUE:
    length4 ns_size;
  case FALSE:
    void;
};

struct LAYOUTCOMMIT4resok {
  newsize4 locr_newsize;
};

union LAYOUTCOMMIT4res switch (nfsstat4 locr_status) {
  case NFS4_OK:
    LAYOUTCOMMIT4resok locr_resok4;
  default:
    void;
};

17.42.4. DESCRIPTION

Commits changes in the layout segment represented by the current file handle, client ID (derived from the sessionid in the preceding SEQUENCE operation), and octet range. Since layout segments are subdividable, a smaller portion of a layout segment, retrieved via LAYOUTGET, may be committed. The region being committed is specified through the octet range (loca_offset and loca_length).

The LAYOUTCOMMIT operation indicates that the client has completed writes using a layout obtained by a previous LAYOUTGET. The client may have only written a subset of the data range it previously requested. LAYOUTCOMMIT allows it to commit or discard provisionally allocated space and to update the server with a new end of file. The layout segment referenced by LAYOUTCOMMIT is still valid after the operation completes and can be continued to be referenced by the client ID, filehandle, octet range, and layout type.

If the loca_reclaim field is set to TRUE, this indicates that the client is attempting to commit changes to a layout after the reboot of the metadata server during the metadata server’s recovery grace period. This type of request may be necessary when the client has uncommitted writes to provisionally allocated regions of a file which were sent to the storage devices before the reboot of the metadata server. In this case the layout provided by the client MUST be a subset of a writable layout that the client held immediately before the reboot of the metadata server. The metadata server is free to
accept or reject this request based on its own internal metadata consistency checks. If the metadata server finds that the layout provided by the client does not pass its consistency checks, it MUST reject the request with the status NFS4ERR_RECLAIM_BAD. The successful completion of the LAYOUTCOMMIT request with loca_reclaim set to TRUE does NOT provide the client with a layout segment for the file. It simply commits the changes to the layout segment specified in the loca_layoutupdate field. To obtain a layout segment for the file the client must issue a LAYOUTGET request to the server after the server’s grace period has expired. If the metadata server receives a LAYOUTCOMMIT request with loca_reclaim set to TRUE when the metadata server is not in its recovery grace period, it MUST reject the request with the status NFS4ERR_NO_GRACE.

Setting the loca_reclaim field to TRUE is required if and only if the committed layout was acquired before the metadata server reboot. If the client is committing a layout segment that was acquired during the metadata server’s grace period, it MUST set the "reclaim" field to FALSE.

The loca_last_write_offset field specifies the offset of the last octet written by the client previous to the LAYOUTCOMMIT. Note: this value is never equal to the file’s size (at most it is one octet less than the file’s size). The metadata server may use this information to determine whether the file’s size needs to be updated. If the metadata server updates the file’s size as the result of the LAYOUTCOMMIT operation, it must return the new size (locr_newsize.ns_size) as part of the results.

The loca_time_modify and loca_time_access fields allow the client to suggest times it would like the metadata server to set. The metadata server may use these time values or it may use the time of the LAYOUTCOMMIT operation to set these time values. If the metadata server uses the client provided times, it should ensure time does not flow backwards. If the client wants to force the metadata server to set an exact time, the client should use a SETATTR operation in a compound right after LAYOUTCOMMIT. See Section 12.5.3 for more details. If the new client desires the resultant mtime or atime, it should construct the COMPOUND so that a GETATTR follows the LAYOUTCOMMIT.

The loca_layoutupdate argument to LAYOUTCOMMIT provides a mechanism for a client to provide layout specific updates to the metadata server. For example, the layout update can describe what regions of the original layout have been used and what regions can be deallocated. There is no NFSv4.1 file layout-specific layoutupdate4 structure.
The layout information is more verbose for block devices than for objects and files because the latter two hide the details of block allocation behind their storage protocols. At the minimum, the client needs to communicate changes to the end of file location back to the server, and, if desired, its view of the file modify and access time. For block/volume layouts, it needs to specify precisely which blocks have been used.

If the layout segment identified in the arguments does not exist, the error NFS4ERR_BADLAYOUT is returned. The layout segment being committed may also be rejected if it does not correspond to an existing layout with an iomode of LAYOUTIOMODE4_RW.

On success, the current filehandle retains its value.

17.42.5. IMPLEMENTATION

Optionally, the client can also use LAYOUTCOMMIT with the loca_reclaim field set to TRUE to convey hints to modified file attributes or to report layout-type specific information such as I/O errors for object-based storage layouts, as normally done during normal operation. Doing so may help the metadata server to recover files more efficiently after reboot. For example, some file system implementations may require expansive recovery of file system objects if the metadata server does not get a positive indication from all clients holding a write layout that they have successfully completed all their writes. Sending a LAYOUTCOMMIT (if required) and then following with LAYOUTRETURN can provide such an indication and allow for graceful and efficient recovery.

17.43. Operation 50: LAYOUTGET - Get Layout Information

17.43.1. SYNOPSIS

(cfh), signal_avail, layout_type, iomode, offset, length, minlength, maxcount -> layout example synopsis
17.43.2.  ARGUMENT

struct LAYOUTGET4args {
    /* CURRENT_FH: file */
    bool  loga_signal_layout_avail;
    layouttype4  loga_layout_type;
    layoutiomode4  loga_iomode;
    offset4  loga_offset;
    length4  loga_length;
    length4  loga_minlength;
    count4  loga_maxcount;
};

17.43.3.  RESULT

struct LAYOUTGET4resok {
    bool  logr_return_on_close;
    layout4  logr_layout;
};

union LAYOUTGET4res switch (nfsstat4 logr_status) {
    case NFS4_OK:  
        LAYOUTGET4resok  logr_resok4;
    case NFS4ERR_LAYOUTTRYLATER:  
        bool  logr_will_signal_layout_avail;
    default:  void;
};

17.43.4.  DESCRIPTION

Requests a layout segment from the metadata server for reading or
writing (and reading) the file given by the filehandle at the octet
range specified by offset and length. Layouts are identified by the
client ID (derived from the sessionid in the preceding SEQUENCE
operation), current filehandle, and layout type (loga_layout_type).
The use of the loga_iomode depends upon the layout type, but should
reflect the client’s data access intent.

If the metadata server is in a grace period, and does not persist
layout segments and device ID to device address mappings, then it
MUST return NFS4ERR_GRACE (see Section 8.6.2.1).

The LAYOUTGET operation returns layout information for the specified
octet range: a layout segment. To get a layout segment from a
specific offset through the end-of-file, regardless of the file’s
length, a loga_length field with all bits set to 1 (one) should be used. If loga_length is zero, or if a loga_length which is not all bits set to one is specified, and loga_length when added to loga_offset exceeds the maximum 64-bit unsigned integer value, the error NFS4ERR_INVAL will result.

The loga_minlength field specifies the minimum size overlap with the requested offset and length that is to be returned. If this requirement cannot be met, no layout must be returned; the error NFS4ERR_LAYOUTTRYLATER can be returned.

The loga_maxcount field specifies the maximum layout size (in octets) that the client can handle. If the size of the layout structure exceeds the size specified by maxcount, the metadata server will return the NFS4ERR_TOOSMALL error.

As well, the metadata server may adjust the range of the returned layout segment based on striping patterns and usage implied by the loga_iomode. The client must be prepared to get a layout segment that does not line up exactly with its request; there MUST be at least an overlap of loga_minlength between the layout returned by the server and the client’s request, or the server SHOULD reject the request. See Section 12.5.2 for more details.

The metadata server may also return a layout segment with an lo_iomode other than that requested by the client. If it does so, it must ensure that the lo_iomode is more permissive than the loga_iomode requested. E.g., this allows an implementation to upgrade read-only requests to read/write requests at its discretion, within the limits of the layout type specific protocol. A lo_iomode of either LAYOUTIOMODE4_READ or LAYOUTIOMODE4_RW must be returned.

The logr_return_on_close result field is a directive to return the layout before closing the file. When the server sets this return value to TRUE, it must be prepared to recall the layout in the case the client fails to return the layout before close. For the server that knows a layout must be returned before a close of the file, this return value can be used to communicate the desired behavior to the client and thus removing one extra step from the client and server’s interaction.

The format of the returned layout (lo_content) is specific to the underlying file system. Layout types other than the NFSv4.1 file layout type are specified outside this document.

If layouts are not supported for the requested file or its containing file system the server SHOULD return NFS4ERR_LAYOUTUNAVAILABLE. If the layout type is not supported, the metadata server should return
NFS4ERR_UNKNOWN_LAYOUTTYPE. If layouts are supported but no layout matches the client provided layout identification, the server should return NFS4ERR_BADLAYOUT. If an invalid loga_iomode is specified, or a loga_iomode of LAYOUTIOMODE4_ANY is specified, the server should return NFS4ERR_BADIOMODE.

If the layout for the file is unavailable due to transient conditions, e.g. file sharing prohibits layouts, the server must return NFS4ERR_LAYOUTTRYLATER.

If the layout request is rejected due to an overlapping layout recall, the server must return NFS4ERR_RECALLCONFLICT. See Section 12.5.4.2 for details.

If the layout conflicts with a mandatory octet range lock held on the file, and if the storage devices have no method of enforcing mandatory locks, other than through the restriction of layouts, the metadata server should return NFS4ERR_LOCKED.

If client sets loga_signal_layout_avail to TRUE, then it is registering with the client a "want" for a layout in the event the layout cannot be obtained due to resource exhaustion. If the server supports and will honor the "want", the results will have logr_will_signal_layout_avail set to TRUE. If so the client should expect a CB_RECALLABLE_OBJ_AVAIL operation to indicate that a layout is available.

On success, the current filehandle retains its value.

17.43.5. IMPLEMENTATION

Typically, LAYOUTGET will be called as part of a compound RPC after an OPEN operation and results in the client having location information for the file; a client may also hold a layout across multiple OPENs. The client specifies a layout type that limits what kind of layout the server will return. This prevents servers from issuing layouts that are unusable by the client.

17.44. Operation 51: LAYOUTRETURN - Release Layout Information

17.44.1. SYNOPSIS

(cfh), layout_type, iomode, layoutreturn, reclaim -> -
17.44.2.  ARGUMENT

struct LAYOUTRETURN4args {
    /* CURRENT_FH: file */
    bool lora_reclaim;
    layouttype4 lora_layout_type;
    layoutiomode4 lora_iomode;
    layoutreturn4 lora_layoutreturn;
};

17.44.3.  RESULT

struct LAYOUTRETURN4res {
    nfsstat4 lorr_status;
};

17.44.4.  DESCRIPTION

Returns one or more layouts or layout segments represented by the client ID (derived from the sessionid in the preceding SEQUENCE operation), lora_layout_type, and lora_iomode. When layoutreturn is LAYOUTRETURN4_FILE the returned layout segment is further identified by the current filehandle, lrf_offset, and lrf_length. When layoutreturn is LAYOUTRETURN4_FSID the current filehandle is used to identify the file system and all layouts or layout segments matching the client ID, lora_layout_type, and lora_iomode are returned. When layoutreturn is LAYOUTRETURN4_ALL all layouts or layout segments matching the client ID, lora_layout_type, and lora_iomode are returned and the current filehandle is not used. After this call, the client MUST NOT use the returned layout segment(s) or layout(s) and the associated storage protocol to access the file data. A layout segment being returned may be a subdivision of a layout segment previously fetched through LAYOUTGET. As well, it may be a subset or superset of a layout segment specified by CB_LAYOUTRECALL. However, if it is a subset, the recall is not complete until the full recalled scope (LAYOUTRETURN4_FILE octet range, LAYOUTRETURN4_FSID, or LAYOUTRETURN4_ALL) has been returned. It is also permissible, and no error should result, for a client to return a octet range covering a layout it does not hold. If the lrf_length is all 1s, the layout covers the range from lrf_offset to EOF. An iomode of LAYOUTIOMODE4_ANY specifies that all layouts that match the other arguments to LAYOUTRETURN (i.e., client ID, lora_layout_type, and one of current filehandle and range; fsid derived from current filehandle; or LAYOUTRETURN4_ALL) are being returned.

When lr_returntype is set to LAYOUTRETURN4_FSID or LAYOUTRETURN4_ALL
the client also invalidates all the storage device ID to storage
device address in the affected file system(s). Any device ID
returned by a subsequent LAYOUTGET in the affected file system(s)
will have to be resolved using either GETDEVICEINFO or GETDEVICELIST.

The lora_reclaim field set to TRUE in a LAYOUTRETURN request
specifies that the client is attempting to return a layout that was
acquired before the reboot of the metadata server during the metadata
server’s grace period. When returning layouts that were acquired
during the metadata server’s grace period MUST set the lora_reclaim
field to FALSE. The lora_reclaim field MUST be set to FALSE also
when lr_layoutreturn is LAYOUTRETURN4_FSID or LAYOUTRETURN4_ALL. See
LAYOUTCOMMIT (Section 17.42) for more details.

Layouts 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 layout to
the storage device(s) or to the metadata server before returning the
layout.

If a client fails to return a layout in a timely manner, then the
metadata server should use its control protocol with the storage
devices to fence the client from accessing the data referenced by the
layout. See Section 12.5.4 for more details.

If the layout identified in the arguments does not exist, the error
NFS4ERR_BADLAYOUT is returned. If a layout exists, but the iomode
does not match, NFS4ERR_BADIOMODE is returned.

If the LAYOUTRETURN request sets the lora_reclaim field to TRUE after
the metadata server’s grace period, NFS4ERR_NO_GRACE is returned.

If the LAYOUTRETURN request sets the lora_reclaim field to TRUE and
lr_returntype is set to LAYOUTRETURN4_FSID or LAYOUTRETURN4_ALL,
NFS4ERR_INVAL is returned.

On success, the current filehandle retains its value.

[[Comment.20: Should LAYOUTRETURN be modified to handle FSID
callbacks?]]

17.44.5. IMPLEMENTATION

The final LAYOUTRETURN operation in response to a CB_LAYOUTRECALL
callback MUST be serialized with any outstanding, intersecting
LAYOUTRETURN operations. Note that it is possible that while a
client is returning the layout for some recalled range the server may
recall a superset of that range (e.g. LAYOUTRECALL4_ALL); the final
return operation for the latter must block until the former layout recall is done – when its corresponding final return operation is replied.

Returning all layouts in a file system using LAYOUTRETURN4_FSID is typically done in response to a CB_LAYOUTRECALL for that file system as the final return operation. Similarly, LAYOUTRETURN4_ALL is used in response to a recall callback for all layouts. It is possible that the client already returned some outstanding layouts via individual LAYOUTRETURN calls and the call for LAYOUTRETURN4_FSID or LAYOUTRETURN4_ALL marks the end of the LAYOUTRETURN sequence. See Section 12.5.4.1 for more details.

17.45. Operation 52: SECINFO_NO_NAME – Get Security on Unnamed Object

Obtain available security mechanisms with the use of the parent of an object or the current filehandle.

17.45.1. SYNOPSIS

\[
\text{(cfh), secinfo\_style} \rightarrow \{ \text{secinfo} \}
\]

17.45.2. ARGUMENT

\[
\text{enum secinfo\_style4} \{
\text{SECINFO\_STYLE4\_CURRENT\_FH} = 0,
\text{SECINFO\_STYLE4\_PARENT} = 1
\};
\]

\[
\text{typedef secinfo\_style4 SECINFO\_NO\_NAME4\_args;}
\]

17.45.3. RESULT

\[
\text{typedef SECINFO4\_res SECINFO\_NO\_NAME4\_res;}
\]

17.45.4. DESCRIPTION

Like the SECINFO operation, SECINFO_NO_NAME is used by the client to obtain a list of valid RPC authentication flavors for a specific file object. Unlike SECINFO, SECINFO_NO_NAME only works with objects that are accessed by filehandle.

There are two styles of SECINFO_NO_NAME, as determined by the value of the secinfo_style4 enumeration. If SECINFO_STYLE4_CURRENT_FH is passed, then SECINFO_NO_NAME is querying for the required security for the current filehandle. If SECINFO_STYLE4_PARENT is passed, then
SECINFO_NO_NAME is querying for the required security of the current filehandles’s parent. If the style selected is SECINFO_STYLE4_PARENT, then SECINFO should apply the same access methodology used for LOOKUPP when evaluating the traversal to the parent directory. Therefore, if the requester does not have the appropriate access to LOOKUPP the parent then SECINFO_NO_NAME must behave the same way and return NFS4ERR_ACCESS.

Note that if PUTFH, PUTPUBFH, or PUTROOTFH return NFS4ERR_WRONGSEC, this is tantamount to the server asserting that the client will have to guess what the required security is, because there is no way to query. Therefore, the client must iterate through the security triples available at the client and reattempt the PUTFH, 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 other forms (e.g. AUTH_SYS and AUTH_NONE), but because such forms lack integrity checks, this puts the client at risk.

The server implementor should pay particular attention to Section 2.6 for instructions on avoiding NFS4ERR_WRONGSEC error returns from PUTFH, PUTROOTFH, PUTPUBFH, or RESTOREFH.

Everything else about SECINFO_NO_NAME is the same as SECINFO. See the discussion on SECINFO (Section 17.29.4).

17.45.5. IMPLEMENTATION

See the discussion on SECINFO (Section 17.29.5).

17.46. Operation 53: SEQUENCE – Supply per-procedure sequencing and control

Supply per-procedure sequencing and control

17.46.1. SYNOPSIS

ccontrol -> control
17.46.2. ARGUMENT

    struct SEQUENCE4args {
        sessionid4 sa_sessionid;
        sequenceid4 sa_sequenceid;
        slotid4 sa_slotid;
        slotid4 sa_highest_slotid;
        bool sa_cachethis;
    };

17.46.3. RESULT

    const SEQ4_STATUS_CB_PATH_DOWN                  = 0x00000001;
    const SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRING      = 0x00000002;
    const SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRED       = 0x00000004;
    const SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED     = 0x00000008;
    const SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED    = 0x00000010;
    const SEQ4_STATUS_ADMIN_STATE_REVOKED           = 0x00000020;
    const SEQ4_STATUS_RECALLABLE_STATE_REVOKED      = 0x00000040;
    const SEQ4_STATUS_LEASE_MOVED                    = 0x00000080;
    const SEQ4_STATUS_RESTART_RECLAIM_NEEDED        = 0x00000100;

    struct SEQUENCE4resok {
        sessionid4 sr_sessionid;
        sequenceid4 sr_sequenceid;
        slotid4 sr_slotid;
        slotid4 sr_highest_slotid;
        slotid4 sr_target_highest_slotid;
        uint32_t sr_status_flags;
    };

    union SEQUENCE4res switch (nfsstat4 sr_status) {
        case NFS4_OK:
            SEQUENCE4resok sr_resok4;
        default:
            void;
    };

17.46.4. DESCRIPTION

    The SEQUENCE operation is used to manage operational accounting for
    the session on which the operation is sent. The contents include the
    client and session to which this request belongs, slotid and
    sequenceid, used by the server to implement session request control
    and the duplicate reply cache semantics, and exchanged slot counts
which are used to adjust these values.

This operation MUST appear as the first operation of any COMPOUND in which it appears. The error NFS4ERR_SEQUENCE_POS will be returned when it is found in any position in a COMPOUND beyond the first. Operations other than SEQUENCE, BIND_CONN_TO_SESSION, EXCHANGE_ID, CREATE_SESSION, and DESTROY_SESSION, may not appear as the first operation in a COMPOUND. Such operations will get the error NFS4ERR_OP_NOT_IN_SESSION if they do appear at the start of a COMPOUND.

If SEQUENCE is received on a connection not bound to the session via CREATE_SESSION or BIND_CONN_TO_SESSION, and the client specified connecting binding enforcement when the session was created (see Section 17.36), then the server returns NFS4ERR_CONN_NOT_BOUND_TO_SESSION.

If sa_cachethis is TRUE, then the client is requesting that the server cache the reply in the server’s reply cache. The server MUST cache the reply (see Section 2.10.4.1.2).

The response to the SEQUENCE operation contains a word of status flags (sr_status_flags) that that can provide to the client information related to the status of the client’s lock state and communications paths. Note that any status bits relating to lock state are MAY reset when lock state is lost due to a server reboot or the establishment of a new client instance. Note that if the client ID implied by sa_sessionid was established with

\[
\text{eir\_flags} \ & \ ( \text{EXCHGID4\_FLAG\_USE\_PNFS\_DS} \\
& \text{EXCHGID4\_FLAG\_USE\_PNFS\_MDS} \\
& \text{EXCHGID4\_FLAG\_USE\_NON\_PNFS}) \\
\] == \text{EXCHGID4\_FLAG\_USE\_PNFS\_DS}

in the EXCHANGE_ID results (i.e the client ID is only for data servers), then sr_status_flags MUST always be zero.

SEQ4\_STATUS\_CB\_PATH\_DOWN
When set, indicates that the client has no operational callback path, making it necessary for the client to re-establish one, return his recallable locks, or both. This bit remains set until the callback path is again available.
SEQ4_STATUS_CB_GSSCONTEXTS_EXPIRING
When set, indicates that the GSS contexts to be used for callbacks are expected to expire within a period equal to the lease time. This bit remains set until the expiration time of the contexts is beyond the lease period from the current time.

SEQ4_STATUS_CB_GSSCONTEXTS_EXPIRED
When set, indicates the GSS contexts to be used for callbacks have expired. This bit remains set until new non-expired contexts are provided.

SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED
When set, indicates that the lease has expired and as a result the server released all of the client’s locking state. This status bit remains set until the loss of all such locks has been acknowledged by use of FREE_BADLOCK, or by establishing a new client instance by destroying all sessions (via DESTROY_SESSION), the client ID (via DESTROY_CLIENT), and then invoking EXCHANGE_ID and CREATE_SESSION to establish a new client ID.

SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED
When set indicates that some subset of the client’s locks have been revoked due to expiration of the lease period followed by another client’s conflicting lock request. This status bit remains set until the loss of all such locks has been acknowledged by use of FREE_BADLOCK.

SEQ4_STATUS_ADMIN_STATE_REVOKED
When set indicates that one or more locks have been revoked without expiration of the lease period, due to administrative action. This status bit remains set until the loss of all such locks has been acknowledged by use of FREE_BADLOCK.

SEQ4_STATUS_RECALLABLE_STATE_REVOKED
When set indicates that one or more recallable locks have been revoked without expiration of the lease period, due to the client’s failure to return them when recalled. This status bit remains set until the loss of all such locks has been acknowledged by use of FREE_BADLOCK.

SEQ4_STATUS_LEASE_MOVED
When set indicates that responsibility for lease renewal has been transferred to one or more new servers. This condition will continue until the client receives an NFS4ERRMOVED error and the server receives the subsequent GETATTR for the fs_locations or fs_locations_info attribute for an access to each file system for which a lease has been moved to a new server.
SEQ4_STATUS_RESTART_RECLAIM_NEEDED
When set indicates that due to server restart or reboot. The reason SEQ4_STATUS_RESTART_RECLAIM_NEEDED is not reset after server restart or reboot is that the session and client ID have persisted (usually due the CREATE_SESSION result having returned the CREATE_SESSION4_FLAG_PERSIST flag in csr_flags), all other leased state has been lost. The client must reclaim the lost state via the procedure described in Section 8.6.2, although re-establishing a clientid and session is neither necessary nor recommended.

If the difference between sa_sequenceid and the sequenceid the server has for the slot is two (2) or more, then server MUST return NFS4ERR_SEQ_MISORDERED. If sa_sequenceid is less than the server’s cached sequenceid (accounting for wraparound of the unsigned sequenceid value), then the server MUST return NFS4ERR_SEQ_MISORDERED. If sa_sequenceid and the cached sequenceid are the same, this is a replay, and the server returns the response to the COMPOUND that is cached. Otherwise, if sa_sequenceid is one greater (accounting for wraparound) than the cached sequenceid, then this is a new request, and the slot’s sequenceid is incremented. The operations subsequent to SEQUENCE, if any, are processed. If there are no other operations, the only other effects are to cache the SEQUENCE reply in the slot, maintain the session’s activity, and renew the lease of state related to the client ID.

If SEQUENCE returns an error, then the state of the slot (sequenceid, cached reply) is not changed, nor is the associated lease renewed.

If SEQUENCE returns NFS4_OK, then the associated lease is renewed, except if SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED is returned in the status word.

The server returns two "highest_slotid" values: sr_highest_slotid, and sr_target_highest_slotid. The former is the highest slotid the server will accept in future SEQUENCE operation, and must not be less than the the value of sa_highest_slotid. The latter is the highest slotid the server would prefer the client use on a future SEQUENCE operation.

17.46.5. IMPLEMENTATION

The server MUST maintain a mapping of sessionid to client ID in order to validate any operations that follow SEQUENCE that take a stateid as an argument and/or result.

If the client establishes a persistent session, then the server MUST also persist the client ID, such that it is valid through server
reboot or restart. If the session and client ID are not persistent, then in the event of server reboot or restart, if the client ID is no longer valid, upon encountering an sa_sessionid that maps to a stale client ID, the server SHOULD return NFS4ERR_STATE_CLIENTID, which indicates that both the client ID and sessionid are stale.

The server’s implementation constraints may require constructing a sessionid such that it is impossible to discern a sessionid that is invalid due to malformation from one that is invalid due to server restart. In that event, when the client receives NFS4ERR_BADSESSION, it may check for stale client ID by issuing a CREATE_SESSION with the client ID. If CREATE_SESSION succeeds, the client has a session to use, and it MAY retry the original COMPOUND with the new sessionid (unless SEQ4_STATUS_RESTART_RECLAIM_NEEDED is returned in sr_status_flags; in which case the client MUST first reclaim state as described in Section 8.6.2.1).

17.47. Operation 54: SET_SSV

17.47.1. SYNOPSIS

ssv, digest -> digest

17.47.2. ARGUMENT

struct ssa_digest_input4 {
    SEQUENCE4args sdi_seqargs;
};

struct SET_SSV4args {
    opaque ssa_ssv<>
    opaque ssa_digest<>
};
17.47.3. RESULT

```
struct ssr_digest_input4 {
    SEQUENCE4res sdi_seqres;
};
struct SET_SSV4resok {
    opaque ssr_digest<>;
};
union SET_SSV4res switch (nfsstat4 ssr_status) {
    case NFS4_OK:
        SET_SSV4resok ssr_resok4;
    default:
        void;
};
```

17.47.4. DESCRIPTION

This operation is used to set or update the SSV for a session. It MUST be preceded by SEQUENCE in the same COMPOUND. It MUST be invoked only on a connection bound to the session. It MUST NOT be used if the client did not enable connecting binding enforcement when the session was created (see Section 17.36); the server returns NFS4ERR_OP_CONN_BINDING_NOT_ENFORCED in that case. If the client enabled connection binding enforcement, then SET_SSV MUST be invoked at least once prior to a BIND_CONN_TO_SESSION operation.

`sra_digest` is computed as the output of the HMAC RFC2104 [14] using the current SSV as the key, and an XDR encoded value of data type `ssr_digest_input4`. The field `sdi_seqargs` is equal to the arguments of the SEQUENCE operation for the COMPOUND procedure that SET_SSV is within.

The ssa_ssv is XORed with the current SSV to produce the new SSV.

In the response, `ssr_digest` is the output of the HMAC using the new SSV as the key, and an XDR encoded value of data type `ssr_digest_input4`. The field `sdi_seqres` is equal to the results of the SEQUENCE operation for the COMPOUND procedure that SET_SSV is within.

17.47.5. IMPLEMENTATION

When the server receives `ssa_digest`, it MUST verify the digest by computing the digest the same way the client did and comparing it with `ssa_digest`. If the server gets a different result, this is an
error, NFS4ERR_BAD_SESSION_DIGEST. Generally, in order to change the
SSV or bind new connections to the session, the client has no
recourse but to recreate the session with CREATE_SESSION. However,
the IMPLEMENTATION section BIND_CONN_TO_SESSION describes a scenario
where a client can legitimately get NFS4ERR_BAD_SESSION_DIGEST for a
SET_SSV, and how to recover from it.

Clients SHOULD NOT send an ssa_ssv that is equal to a previous
ssa_ssv, nor equal to a previous SSV.

Clients SHOULD issue SET_SSV with RPCSEC_GSS privacy. Servers MUST
support RPCSEC_GSS with privacy for any COMPOUND that has { SEQUENCE,
SET_SSV }.

17.48. Operation 55: TEST_STATEID - Test stateids for validity

Test a series of stateids for validity.

17.48.1. SYNOPSIS

stateids<> -> error_codes<>  

17.48.2. ARGUMENT

struct TEST_STATEID4args {
    stateid4     ts_stateids<>;
};

17.48.3. RESULT

struct TEST_STATEID4resok {
    nfsstat4    tsr_status_codes<>;
};

union TEST_STATEID4res switch (nfsstat4 tsr_status) {
    case NFS4_OK:
        TEST_STATEID4resok tsr_resok4;
    default:
        void;
};

17.48.4. DESCRIPTION

The TEST_STATEID operation is used to check the validity of a set of
stateids. It is intended to be used when the client receives an
indication that one or more of its stateids have been invalidated due
To lock revocation, TEST_STATEID allows a large set of such stateids to be tested and allows problems with earlier stateids not to interfere with checking of subsequent ones as would happen if individual stateids are tested by operation in a COMPOUND.

For each stateid, the server provides the status code that would be returned if that stateid were to be used in normal operation. Returning such an status indication is not an error and does not cause processing to terminate. Checks for the validity of the stateid proceed as they would for normal operations with two exceptions. There is no check for the type of stateid object, as would be the case for normal and there is no reference to the current filehandle.

The errors which are validly returned within the status_code array are: NFS4ERR_OK, NFS4ERR_BAD_STATEID, NFS4ERR_EXPIRED, NFS4ERR_ADMIN_REVOKED, and NFS4ERR_DELEG_REVOKED.

17.48.5. IMPLEMENTATION

No discussion at this time.

17.49. Operation 56: WANT_DELEGATION

17.49.1. SYNOPSIS

(cfh), (client ID) -> stateid, delegation
17.49.2.  ARGUMENT

union deleg_claim4 switch (open_claim_type4 dc_claim) {
  /*
   * No special rights to object. Ordinary delegation
   * request of the specified object. Object identified
   * by filehandle.
   */
  case CLAIM_FH: /* new to v4.1 */
    void;

  /*
   * Right to file based on a delegation granted to a previous boot
   * instance of the client. File is specified by filehandle.
   */
  case CLAIM_DELEG_PREV_FH: /* new to v4.1 */
    /* CURRENT_FH: file being opened */
    void;

  /*
   * Right to the file established by an open previous to server
   * reboot. File identified by filehandle.
   * Used during server reclaim grace period.
   */
  case CLAIM_PREVIOUS:
    /* CURRENT_FH: file being reclaimed */
    open_delegation_type4   dc_delegate_type;
};

struct WANT_DELEGATION4args {
  uint32_t        wda_want;
  deleg_claim4    wda_claim;
};

17.49.3.  RESULT

union WANT_DELEGATION4res switch (nfsstat4 wdr_status) {
  case NFS4_OK:
    open_delegation4 wdr_resok4;
  default:
    void;
};
17.49.4. DESCRIPTION

Where this description mandates the return of a specific error code for a specific condition, and where multiple conditions apply, the server MAY return any of the mandated error codes.

This operation allows a client to get a delegation on all types of files except directories. The server MAY support this operation. If the server does not support this operation, it MUST return NFS4ERR_NOTSUPP. This operation also allows the client to register a "want" for a delegation for the specified file object, and be notified via a callback when the delegation is available. The server MAY support notifications of availability via callbacks. If the server does not support registration of wants it MUST NOT return an error to indicate that.

The client SHOULD NOT set OPEN4_SHARE_ACCESS_READ and SHOULD NOT set OPEN4_SHARE_ACCESS_WRITE in wda.want. If it does, the server MUST ignore them.

The meanings of the following flags in wda.want are the same as they are in OPEN:

OPEN4_SHARE_ACCESS_WANT_READ_DELEG
OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG
OPEN4_SHARE_ACCESS_WANT_ANY_DELEG
OPEN4_SHARE_ACCESS_WANT_NO_DELEG
OPEN4_SHARE_ACCESS_WANT_CANCEL
OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL
OPEN4_SHARE_ACCESS_WANT_PUSH_DELEG_WHEN_UNCONTENDED

The handling of the above flags in WANT_DELEGATION is the same as in OPEN.

A request for a conflicting delegation MUST NOT trigger the recall of the existing delegation.

The successful results of WANT_DELEG are of type open_delegation4 which is the same type as the "delegation" field in the results of the OPEN operation. The server constructs wdr_resok4 the same way it constructs OPEN's "delegation" with one differences: WANT_DELEGATION MUST NOT return a delegation type of OPEN_DELEGATE_NONE. As with
OPEN, if (wda_want & OPEN4_SHARE_ACCESS_WANT_DELEG_MASK) is zero then the client is indicating no desire for a delegation and the server MAY or MAY not return a delegation in the WANT_DELEG response.

17.49.5. IMPLEMENTATION

TBD

17.50. Operation 57: DESTROY_CLIENTID - Destroy existing client ID

Destroy existing client ID.

17.50.1. SYNOPSIS

client ID -> -

17.50.2. ARGUMENT

struct DESTROY_CLIENTID4args {
    clientid4 dca_clientid;
};

17.50.3. RESULT

struct DESTROY_CLIENTID4res {
    nfsstat4 dcr_status;
};

17.50.4. DESCRIPTION

The DESTROY_CLIENTID operation destroys the client ID if there are no sessions, opens, locks, delegations, layouts, and wants, associated with the client ID.

If the COMPOUND request starts with SEQUENCE, then the session identified in SEQUENCE must not be one bound to the client ID identified in DESTROY_CLIENTID or the DESTROY_CLIENTID operation will fail because there is still a session bound to the client ID. DESTROY_CLIENTID MAY be the only operation in a COMPOUND request.

Note that because the operation can be sent outside of a session, a client that retransmits the request may receive an error in response, because though the original request resulted in the successful destruction of the client ID.
17.50.5. IMPLEMENTATION

DESTROY_CLIENTID allows a server to immediately reclaim the resources consumed by an unused client ID, and also to forget that it ever generated the client ID. By forgetting it ever generated the the client ID the server can safely reuse the client ID on a future EXCHANGE_ID operation.

17.51. Operation 10044: ILLEGAL - Illegal operation

17.51.1. SYNOPSIS

-> ()

17.51.2. ARGUMENTS

void;

17.51.3. RESULTS

/*
 * ILLEGAL: Response for illegal operation numbers
 */
struct ILLEGAL4res {
    nfsstat4 status;
};

17.51.4. DESCRIPTION

This operation is a placeholder for encoding a result to handle the case of the client sending an operation code within COMPOUND that is not supported. See the COMPOUND procedure description for more details.

The status field of ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

17.51.5. 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.
18. NFS version 4.1 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.

18.1. Procedure 0: CB_NULL - No Operation

18.1.1. SYNOPSIS

18.1.2. ARGUMENTS

void;

18.1.3. RESULTS

void;

18.1.4. 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.

18.1.5. ERRORS

None.

18.2. Procedure 1: CB_COMPOUND - Compound Operations

18.2.1. SYNOPSIS

compoundargs -> compoundres
18.2.2. ARGUMENTS

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            opcbillegal;
};

struct CB_COMPOUND4args {
    utf8str_cs      tag;
    uint32_t        minorversion;
    nfs_cb_argop4   argarray<>;
};

18.2.3. RESULTS

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

struct CB_COMPOUND4res {
    nfsstat4 status;
    utf8str_cs      tag;
    nfs_cb_resop4   resarray<>;
};

18.2.4. 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. This is discussed in Section 2.10.4.4.

The minorversion field of the arguments MUST be the same as the minorversion of the COMPOUND procedure used to created the client ID and session. For NFSv4.1, minorversion MUST be set to 1.
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 the section "Procedure 1: COMPOUND - Compound Operations". [[Comment.21: Need an xref.]]

Illegal operation codes are handled in the same way as they are handled for the COMPOUND procedure.

18.2.5. 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.

18.2.6. ERRORS

NFS4ERR_BADHANDLE NFS4ERR_BAD_STATEID NFS4ERR_BADXDR
NFS4ERR_OP_ILLEGAL NFS4ERR_RESOURCE NFS4ERR_SERVERFAULT

19. NFS version 4.1 Callback Operations

19.1. Operation 3: CB_GETATTR - Get Attributes

19.1.1. SYNOPSIS

    fh, attr_request -> attrmask, attr_vals
19.1.2. ARGUMENT

/*
 * NFS4 Callback Procedure Definitions and Program
 */

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

19.1.3. RESULT

struct CB_GETATTR4resok {
    fattr4 obj_attributes;
};

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

19.1.4. DESCRIPTION

The CB_GETATTR operation is used by the server to obtain the current modified state of a file that has been write delegated. The attributes size and change are the only ones guaranteed to be serviced by the client. See the section "Handling of CB_GETATTR" 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 write open delegation, an NFS4ERR_BADHANDLE error is returned.

19.1.5. 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).
19.2. Operation 4: CB_RECALL - Recall an Open Delegation

19.2.1. SYNOPSIS

stateid, truncate, fh -> ()

19.2.2. ARGUMENT

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

19.2.3. RESULT

struct CB_RECALL4res {
    nfsstat4  status;
};

19.2.4. 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.

19.2.5. 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.
19.3. Operation 5: CB_LAYOUTRECALL

19.3.1. SYNOPSIS

layout_type, iomode, layoutchanged, layoutrecall -> -

19.3.2. ARGUMENT

/ *
* NFSv4.1 callback arguments and results
*/

enum layoutrecall_type4 {
  LAYOUTRECALL4_FILE = 1,
  LAYOUTRECALL4_FSID = 2,
  LAYOUTRECALL4_ALL = 3
};

struct layoutrecall_file4 {
  nfs_fh4         lor_fh;
  offset4         lor_offset;
  length4         lor_length;
};

union layoutrecall4 switch(layoutrecall_type4 recalltype) {
  case LAYOUTRECALL4_FILE:
    layoutrecall_file4 lor_layout;
  case LAYOUTRECALL4_FSID:
    fsid4              lor_fsid;
  case LAYOUTRECALL4_ALL:
    void;
};

struct CB_LAYOUTRECALL4args {
  layouttype4             clora_type;
  layoutiomode4           clora_iomode;
  bool                    clora_changed;
  layoutrecall4           clora_recall;
};

19.3.3. RESULT

struct CB_LAYOUTRECALL4res {
  nfsstat4        clorr_status;
};
19.3.4. DESCRIPTION

The CB_LAYOUTRECALL operation is used to begin the process of recalling layout segments, a layout, all layouts pertaining to a particular file system (FSID), or layouts in all file systems (ALL). If LAYOUTRECALL4_FILE is specified, the lrf_offset and lrf_length fields specify the layout segments. If a lrf_length of all ones is specified then all layout segments identified by the current file handle, clora_type, clora_iomode, and corresponding to the octet range from lrf_offset to the end-of-file MUST be returned (via LAYOUTRETURN, see Section 17.44). The clora_iomode specifies the set of layouts to be returned. An clora_iomode of LAYOUTIOMODE4_ANY specifies that all matching layout segments regardless of iomode, must be returned; otherwise, only layout segments that exactly match the iomode must be returned. If clora_iomode is LAYOUTIOMODE4_ANY, lo_offset is zero, and lo_length is all ones, then the entire layout is to be returned.

If the clora_changed field is TRUE, then the client SHOULD not write and commit its modified data to the storage devices specified by the layout being recalled. Instead, it is preferable for the client to write and commit the modified data through the metadata server. Alternatively, the client may attempt to obtain a new layout. Note: in order to obtain a new layout the client must first return the old layout. Since obtaining a new layout is not guaranteed to succeed, the client must be ready to write and commit its modified data through the metadata server.

If the client does not hold any layout segment either matching or overlapping with the requested layout, it returns NFS4ERR_NOMATCHING_LAYOUT.

If LAYOUTRECALL4_FSID is specified, the fsid specifies the file system for which any outstanding layouts MUST be returned. If LAYOUTRECALL4_ALL is specified, all outstanding layouts MUST be returned. In addition, LAYOUTRECALL4_FSID and LAYOUTRECALL4_ALL specify that all the storage device ID to storage device address mappings in the affected file system(s) are also recalled. The respective LAYOUTRETURN with either LAYOUTRETURN4_FSID or LAYOUTRETURN4_ALL acknowledges to the server that the client invalidated the said device mappings. Device mappings are invalidated also when no layouts are found for LAYOUTRECALL4_FSID or LAYOUTRECALL4_ALL and NFS4ERR_NOMATCHING_LAYOUT is returned.

19.3.5. IMPLEMENTATION

The client should reply to the callback immediately. Replying does not complete the recall except when an error is returned; otherwise
the recall is not complete until the layout(s) are returned using a
LAYOUTRETURN operation.

The client should complete any in-flight I/O operations using the
recalled layout(s) before returning it/them via LAYOUTRETURN. If the
client has buffered modified data there are a number of options for
writing and committing that data. If clora_changed is false, the
client may choose to write modified data directly to storage before
calling LAYOUTRETURN. However, if clora_changed is true, the client
may either choose to write it later using normal NFSv4 WRITE
operations to the metadata server or it may attempt to obtain a new
layout, after first returning the recalled layout, using the new
layout to write the modified data. Regardless of whether the client
is holding a layout, it may always write data through the metadata
server.

If modified data is written while the layout is held, the client must
still issue LAYOUTCOMMIT operations at the appropriate time,
especially before issuing the LAYOUTRETURN. If a large amount of
modified data is outstanding, the client may issue LAYOUTRETURNs for
portions of the layout being recalled; this allows the server to
monitor the client’s progress and adherence to the callback.
However, the last LAYOUTRETURN in a sequence of returns, MUST specify
the full range being recalled (see Section 12.5.4.1 for details).

19.4. Operation 6: CB_NOTIFY - Notify directory changes

Tell the client of directory changes.

19.4.1. SYNOPSIS

stateid, notification -> {}

19.4.2. ARGUMENT

/* Changed entry information. */
struct notify_entry4 {
    component4 ne_file;
    fattr4 ne_attrs;
};

/* Previous entry information */
struct prev_entry4 {
    notify_entry4 pe_prev_entry;
    /* what readdir returned for this entry */
    nfs_cookie4 pe_prev_entry_cookie;
};
struct notify_add4 {
    notify_entry4        nad_new_entry;
    /* what READDIR would have returned for this entry */
    nfs_cookie4          nad_new_entry_cookie<1>;
    prev_entry4          nad_prev_entry<1>;
    bool                 nad_last_entry;
};

struct notify_attr4 {
    notify_entry4    na_changed_entry;
};

struct notify_remove4 {
    notify_entry4    nrm_old_entry;
    nfs_cookie4      nrm_old_entry_cookie;
};

struct notify_rename4 {
    notify_entry4    nrn_old_entry;
    notify_add4      nrn_new_entry;
};

struct notify_verifier4 {
    verifier4        nv_old_cookieverf;
    verifier4        nv_new_cookieverf;
};

enum notify_type4 {
    NOTIFY4_CHANGE_CHILD_ATTRS = 0,
    NOTIFY4_CHANGE_DIR_ATTRS = 1,
    NOTIFY4_REMOVE_ENTRY = 2,
    NOTIFY4_ADD_ENTRY = 3,
    NOTIFY4_RENAME_ENTRY = 4,
    NOTIFY4_CHANGE_COOKIE_VERIFIER = 5
};

/* Notification information sent to the client. */
union notify4 switch (notify_type4 n_type) {
    case NOTIFY4_CHANGE_CHILD_ATTRS:
        notify_attr4    n_change_child_attrs;
    case NOTIFY4_CHANGE_DIR_ATTRS:
        fattr4          n_change_dir_attrs;
    case NOTIFY4_REMOVE_ENTRY:
        notify_remove4  n_remove_notify;
    case NOTIFY4_ADD_ENTRY:
        notify_add4     n_add_notify;
}
case NOTIFY4_RENAME_ENTRY:
    notify_rename4  n_rename_notify;
    case NOTIFY4_CHANGE_COOKIE_VERIFIER:
        notify_verifier4 n_verf_notify;
    }
};

struct CB_NOTIFY4args {
    stateid4    cna_stateid;
    nfs_fh4     cna_fh;
    notify4     cna_changes<>;
};

19.4.3.  RESULT

struct CB_NOTIFY4res {
    nfsstat4    cnr_status;
};

19.4.4.  DESCRIPTION

The CB_NOTIFY operation is used by the server to send notifications to clients about changes in a delegated directory. These notifications are sent over the callback path. The notification is sent once the original request has been processed on the server. The server will send an array of notifications for all changes that might have occurred in the directory. The notify_type4 can only have one bit set for each notification in the array. If the client holding the delegation makes any changes in the directory that cause files or sub directories to be added or removed, the server will notify that client of the resulting change(s). If the client holding the delegation is making attribute or cookie verifier changes only, the server does not need to send notifications to that client. The server will send the following information for each operation:

ADDING A FILE  The server will send information about the new entry being created along with the cookie for that entry. The entry information (data type notify_add4) includes the component name of the entry and attributes. If this entry is added to the end of the directory, the server will set the nad_last_entry flag to true. If the file is added such that there is at least one entry before it, the server will also return the previous entry information (nad_prev_entry, a variable length array of up to one element. If the array is of zero length, there is no previous entry), along with its cookie. This is to help clients find the right location in their DNLC or directory caches where this entry should be cached. If the new entry’s cookie is available, it will
be in nad_new_entry_cookie (another variable length array of up to one element).

REMOVING A FILE  The server will send information about the directory entry being deleted. The server will also send the cookie value for the deleted entry so that clients can get to the cached information for this entry.

RENAMING A FILE  The server will send information about both the old entry and the new entry. This includes name and attributes for each entry. This notification is only sent if both entries are in the same directory. If the rename is across directories, the server will send a remove notification to one directory and an add notification to the other directory, assuming both have a directory delegation.

FILE/DIR ATTRIBUTE CHANGE  The client will use the attribute mask to inform the server of attributes for which it wants to receive notifications. This change notification can be requested for both changes to the attributes of the directory as well as changes to any file attributes in the directory by using two separate attribute masks. The client cannot ask for change attribute notification per file. One attribute mask covers all the files in the directory. Upon any attribute change, the server will send back the values of changed attributes. Notifications might not make sense for some file system wide attributes and it is up to the server to decide which subset it wants to support. The client can negotiate the frequency of attribute notifications by letting the server know how often it wants to be notified of an attribute change. The server will return supported notification frequencies or an indication that no notification is permitted for directory or child attributes by setting the dir_notif_delay and dir_entry_notif_delay attributes respectively.

COOKIE VERIFIER CHANGE  If the cookie verifier changes while a client is holding a delegation, the server will notify the client so that it can invalidate its cookies and reissue a READDIR to get the new set of cookies.

19.4.5. IMPLEMENTATION

19.5. Operation 7: CB_PUSH_DELEG

19.5.1. SYNOPSIS

   fh, stateid -> { }
19.5.2. ARGUMENT

    struct CB_PUSH_DELEG4args {
        stateid4 cpda_stateid;
        nfs_fh4 cpda_fh;
        open_delegation4 cpda_delegation;
    };

19.5.3. RESULT

    struct CB_PUSH_DELEG4res {
        nfsstat4 cpdr_status;
    };

19.5.4. DESCRIPTION

    CB_PUSH_DELEG is used by the server to both signal to the client that
    the delegation it wants is available and to simultaneously offer the
    delegation to the client. The client has the choice of accepting the
    delegation by returning NFS4_OK to the server, delaying the decision
    to accept the offered delegation by returning NFS4ERR_DELAY or
    permanently rejecting the offer of the delegation via any other error
    status.

    The server MUST send in cpda_delegation a delegation corresponding to
    the type of what the client requested in the OPEN, WANT_DELEGATION,
    or GET_DIR_DELEGATION request.

    If the client does return NFS4ERR_DELAY and there is a conflicting
    delegation request, the server MAY process it at the expense of the
    client that returned NFS4ERR_DELAY. The client’s want will not be
    cancelled, but MAY processed behind other delegation requests or
    registered wants.

19.5.5. IMPLEMENTATION

    TBD

19.6. Operation 8: CB_RECALL_ANY - Keep any N delegations

    Notify client to return delegation and keep N of them.

19.6.1. SYNOPSIS

    N, type_mask -> {}
19.6.2. ARGUMENT

const RCA4_TYPE_MASK_RDATA_DLG = 0;
const RCA4_TYPE_MASK_WDATA_DLG = 1;
const RCA4_TYPE_MASK_DIR_DLG = 2;
const RCA4_TYPE_MASK_FILE_LAYOUT = 3;
const RCA4_TYPE_MASK_BLK_LAYOUT_MIN = 4;
const RCA4_TYPE_MASK_BLK_LAYOUT_MAX = 7;
const RCA4_TYPE_MASK_OBJ_LAYOUT_MIN = 8;
const RCA4_TYPE_MASK_OBJ_LAYOUT_MAX = 11;
const RCA4_TYPE_MASK_OTHER_LAYOUT_MIN = 12;
const RCA4_TYPE_MASK_OTHER_LAYOUT_MAX = 15;

struct CB_RECALL_ANY4args {
    uint32_t craa_objects_to_keep;
    bitmap4 craa_type_mask;
};

19.6.3. RESULT

struct CB_RECALL_ANY4res {
    nfsstat4 crar_status;
};

19.6.4. DESCRIPTION

The server may decide that it cannot hold all of the state for recallable objects, such as delegations and layouts, without running out of resources. In such a case, it is free to recall individual objects to reduce the load but this would be far from optimal.

Because the general purpose of such recallable objects as delegations is to eliminate client interaction with the server, the server cannot interpret lack of recent use as indicating that the object is no longer useful. The absence of visible use may be the result of a large number of potential operations eliminated. In the case of layouts, the layout will be used explicitly but the meta-data server does not have direct knowledge of such use.

In order to implement an effective reclaim scheme for such objects, the server’s knowledge of available resources must be used to determine when objects must be recalled with the clients selecting the actual objects to be returned.

Server implementations may differ in their resource allocation requirements. For example, one server may share resources among all
classes of recallable objects whereas another may use separate
resource pools for layouts and for delegations, or further separate
resources by types of delegations.

When a given resource pool is over-utilized, the server can issue a
CB_RECALL_ANY to clients holding recallable objects of the types
involved, allowing it to keep a certain number of such objects and
return any excess. A mask specifies which types of objects are to be
limited. The client chooses, based on its own knowledge of current
usefulness, which of the objects in that class should be returned.

For NFSv4.1, sixteen bits are defined. For some of these, ranges are
defined and it is up to the definition of the storage protocol to
specify how these are to be used. There are ranges for blocks-based
storage protocols, for object-based storage protocols and a reserved
range for other experimental storage protocols. The RFC defining
such a storage protocol needs to specify how particular bits within
its range are to be used. For example, it may specify a mapping
between attributes of the layout (read vs. write, size of area) and
the bit to be used or it may define a field in the layout where the
associated bit position is made available by the server to the
client.

When an undefined bit is set in the type mask, NFS4ERR_INVAL should
be returned. However even if a client does not support an object of
the specified type, if the bit is defined, NFS4ERR_INVAL should not
be returned. Future minor versions of NFSv4 may expand the set of
valid type mask bits.

CB_RECALL_ANY specifies a count of objects that the client may keep
as opposed to a count that the client must return. This is to avoid
potential race between a CB_RECALL_ANY that had a count of objects to
free with a set of client-originated operations to return layouts or
delegations. As a result of the race, the client and server would
have differing ideas as to how many objects to return. Hence the
client could mistakenly free too many.

If resource demands prompt it, the server may send another
CB_RECALL_ANY with a lower count, even it has not yet received an
acknowledgement from the client for a previous CB_RECALL_ANY with the
same type mask. Although the possibility exists that these will be
received by the client in a order different from the order in which
they were sent, any such permutation of the callback stream is
harmless. It is the job of the client to bring down the size of the
recallable object set in line with each CB_RECALL_ANY received and
until that obligation is met it cannot be canceled or modified by any
subsequent CB_RECALL_ANY for the same type mask. Thus if the server
sends two CB_RECALL_ANY’s, the effect will be the same as if the
lower count was sent, whatever the order of recall receipt. Note that this means that a server may not cancel the effect of a
CB_RECALL_ANY by sending another recall with a higher count. When a
CB_RECALL_ANY is received and the count is already within the limit
set or is above a limit that the client is working to get down to,
that callback has no effect.

The client can choose to return any type of object specified by the
mask. If a server wishes to limit use of objects of a specific type,
it should only specify that type in the mask sent. The client may
not return requested objects and it is up to the server to handle
this situation, typically by doing specific recalls to properly limit
resource usage. The server should give the client enough time to
return objects before proceeding to specific recalls. This time
should not be less than the lease period.

Servers are generally free not to give out recallable objects when
insufficient resources are available. Note that the effect of such a
policy is implicitly to give precedence to existing objects relative
to requested ones, with the result that resources might not be
optimally used. To prevent this, servers are well advised to make
the point at which they start issuing CB_RECALL_ANY callbacks
somewhat below that at which they cease to give out new delegations
and layouts. This allows the client to purge its less-used objects
whenever appropriate and so continue to have its subsequent requests
given new resources freed up by object returns.

19.6.5. IMPLEMENTATION

19.7. Operation 9: CB_RECALLABLE_OBJ_AVAIL

19.7.1. SYNOPSIS

TBD

19.7.2. ARGUMENT

typedef CB_RECALL_ANY4args CB_RECALLABLE_OBJ_AVAIL4args;

19.7.3. RESULT

struct CB_RECALLABLE_OBJ_AVAIL4res {
    nfsstat4        croa_status;
};
19.7.4. DESCRIPTION

CB_RECALLABLE_OBJ_AVAIL is used by the server to signal the client that the server has resources to grant recallable objects that might previously have been denied by OPEN, WANT_DELEGATION, GET_DIR_DELEG, or LAYOUTGET.

The argument, objects_to_keep means the total number of recallable objects of the types indicated in the argument type_mask that the server believes it can allow the client to have, including the number of such objects the client already has. A client that tries to acquire more recallable objects than the server informs it can have runs the risk of having objects recalled.

19.7.5. IMPLEMENTATION

TBD

19.8. Operation 10: CB_RECALL_SLOT - change flow control limits

Change flow control limits

19.8.1. SYNOPSIS

targetcount -> status

19.8.2. ARGUMENT

struct CB_RECALL_SLOT4args {
    uint32_t rsa_target_highest_slotid;
};

19.8.3. RESULT

struct CB_RECALL_SLOT4res {
    nfsstat4 rsr_status;
};

19.8.4. DESCRIPTION

The CB_RECALL_SLOT operation requests the client to return session slots, and if applicable, transport credits (e.g. RDMA credits for connections bound to the operations channel) to the server. CB_RECALL_SLOT specifies rsa_target_highest_slotid, the target highest_slot the server wants for the session. The client, should then work toward reducing the highest_slot to the target.
If the session has only non-RDMA connections bound to its operations channel, then the client need only wait for all outstanding requests with a slotid > rsa_target_highest_slotid to complete, then issue a single COMPOUND consisting of a single SEQUENCE operation, with the sa_highslot field set to rsa_target_highest_slotid. If there are RDMA-based connections bound to operation channel, then the client needs to also issue enough zero-length RDMA Sends to take the total RDMA credit count to rsa_target_highest_slotid + 1 or below.

19.8.5. IMPLEMENTATION

No discussion at this time.

19.9. Operation 11: CB_SEQUENCE - Supply callback channel sequencing and control

Sequence and control

19.9.1. SYNOPSIS

control -> control

19.9.2. ARGUMENT

struct referring_call4 {
    sequenceid4    rc_sequenceid;
    slotid4        rc_slotid;
};

struct referring_call_list4 {
    sessionid4     rcl_sessionid;
    referring_call4 rcl_referring_calls<>
};

struct CB_SEQUENCE4args {
    sessionid4           csa_sessionid;
    sequenceid4          csa_sequenceid;
    slotid4              csa_slotid;
    slotid4              csa_highest_slotid;
    bool                 csa_cachethis;
    referring_call_list4 csa_referring_call_lists<>
};
19.9.3. RESULT

```c
struct CB_SEQUENCE4resok {
    sessionid4 csr_sessionid;
    sequenceid4 csr_sequenceid;
    slotid4 csr_slotid;
    slotid4 csr_highest_slotid;
    slotid4 csr_target_highest_slotid;
};
```

union CB_SEQUENCE4res switch (nfsstat4 csr_status) {
    case NFS4_OK:
        CB_SEQUENCE4resok csr_resok4;
    default:
        void;
};

19.9.4. DESCRIPTION

The CB_SEQUENCE operation is used to manage operational accounting for the callback channel of the session on which the operation is sent. The contents include the session to which this request belongs, slotid and sequenceid used by the server to implement session request control and exactly once semantics, and exchanged slot maximums which are used to adjust the size of the replay cache. This operation MUST appear once as the first operation in each CB_COMPOUND sent procedure after the callback channel is successfully bound, or a protocol error must result. See Section 17.46.4 for a description of how slots are processed.

If csa_cachethis is TRUE, then the server is requesting that the client cache the reply in the callback reply cache. The client MUST cache the reply (see Section 2.10.4.1.2).

The csa_referring_call_lists array is the list of COMPOUND calls, identified by sessionid, slotid and sequenceid, that the client previously sent to the server that could have triggered the callback. A sessionid is included because leased state is tied to a client ID, and a client ID can have multiple sessions. See Section 2.10.4.3 Resolving server callback races with sessions.

If the difference between csa_sequenceid and the sequenceid the client has for the slot is two (2) or more, then client MUST return NFS4ERR_SEQ_MISORDERED. If csa_sequenceid is less than the client’s cached sequenceid (accounting for wraparound of the unsigned sequenceid value), then the client MUST return NFS4ERR_SEQ_MISORDERED. If sa_sequenceid and the cached sequenceid...
are the same, this is a replay, and the client returns the response to the CB_COMPOUND that is cached. Otherwise, if \textit{sa\_sequenceid} is one greater (accounting for wraparound) than the cached \textit{sequenceid}, then this is a new request, and the slot’s \textit{sequenceid} is incremented. The operations subsequent to CB\_SEQUENCE, if any, are processed. If there are no other operations, the only other effects are to cache the CB\_SEQUENCE reply in the slot.

If CB\_SEQUENCE returns an error, then the state of the slot (\textit{sequenceid}, cached reply) is not changed.

The client returns two "highest_slotid" values: \textit{csr\_highest\_slotid}, and \textit{csr\_target\_highest\_slotid}. The former is the highest slotid the client will accept in a future CB\_SEQUENCE operation, and must not be less than the the value of \textit{csa\_highest\_slotid}. The latter is the highest slotid the client would prefer the client use on a future CB\_SEQUENCE operation.

19.9.5. IMPLEMENTATION

19.10. Operation 12: CB\_WANTS\_CANCELLED

19.10.1. SYNOPSIS

\texttt{fh, size \to \text{-}}

19.10.2. ARGUMENT

\begin{verbatim}
struct CB\_WANTS\_CANCELLED\_4args {
   bool cwca\_contended\_wants\_cancelled;
   bool cwca\_resourced\_wants\_cancelled;
};
\end{verbatim}

19.10.3. RESULT

\begin{verbatim}
struct CB\_WANTS\_CANCELLED\_4res {
   nfsstat4 cwcr\_status;
};
\end{verbatim}

19.10.4. DESCRIPTION

The CB\_WANTS\_CANCELLED operation is used to notify the client that the some or all wants it registered for recallable delegations and layouts have been canceled.

If \texttt{cwca\_contended\_wants\_cancelled} is \texttt{TRUE}, this indicates the server
will not be pushing to the client any delegations that become available after contention passes.

If cwca_resourced_wants_cancelled is TRUE, this indicates the server will not notify the client when there are resources on the server grant delegations or layouts.

After receiving a CB_WANTS_CANCELLED operation, the client is free to attempt to acquire the delegations or layouts it was waiting for, and possibly re-register wants.

19.10.5. IMPLEMENTATION

19.11. Operation 13: CB_NOTIFY_LOCK - Notify of possible lock availability

19.11.1. SYNOPSIS

    fh, lockowner -> ()

19.11.2. ARGUMENT

    struct CB_NOTIFY_LOCK4args {
        lock_owner4 cnla_lock_owner;
        nfs_fh4     cnla_fh;
    };

19.11.3. RESULT

    struct CB_NOTIFY_LOCK4res {
        nfsstat4        cnlr_status;
    };

19.11.4. DESCRIPTION

    The server may use this operation to indicate that a lock for the given file and lockowner may have become available.

    This callback is meant to be used by servers to help reduce the latency of blocking locks in the case where they recognize that a client which has been polling for a blocking lock may now be able to acquire the lock. The notification is purely a hint, provided as a possible performance optimization, and is not required for correctness.
19.11.5. IMPLEMENTATION

The server must not grant the lock to the client unless and until it receives an actual lock request from the client. Similarly, the client receiving this callback cannot assume that it now has the lock, or that a subsequent request for the lock will be successful.

The server is not required to implement this callback, and even if it does, it is not required to use it in any particular case. Therefore the client must still rely on polling for blocking locks, as described in the "Blocking Locks" section.

Similarly, the client is not required to implement this callback, and even if it does, is still free to ignore it. Therefore the server must not assume that the client will act based on the callback.

If the server supports this callback for a given file, it should set the OPEN4_RESULT_MAY_NOTIFY_LOCK flag when responding to successful opens for that file. This does not commit the server to use of CB_NOTIFY_LOCK, but the client may use this as a hint to decide how frequently poll for locks derived from that open.


19.12.1. SYNOPSIS

<null> -> ()

19.12.2. ARGUMENT

void;

19.12.3. RESULT

/*
 * CB_ILLEGAL: Response for illegal operation numbers
 */
struct CB_ILLEGAL4res {
    nfsstat4 status;
};

19.12.4. DESCRIPTION

This operation is a placeholder for encoding a result to handle the case of the client sending an operation code within COMPOUND that is not supported. See the COMPOUND procedure description for more details.
The status field of CB_ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

19.12.5. 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.

20. Security Considerations

TBD

21. IANA Considerations

21.1. Defining new layout types

New layout type numbers will be requested from IANA. IANA will only provide layout type numbers for Standards Track RFCs approved by the IESG, in accordance with Standards Action policy defined in RFC2434 [16].

The author of a new pNFS layout specification must follow these steps to obtain acceptance of the layout type as a standard:

1. The author devises the new layout specification.

2. The new layout type specification MUST, at a minimum:

   * Define the contents of the layout-type-specific fields of the following data types:

     + the da_addr_body field of the device_addr4 data type;
     + the loc_body field of the layouthint4 data type;
     + the loc_body field of layout_content4 data type (which in turn is the lo_content field of the layout4 data type);
     + the lou_body field of the layoutupdate4 data type;

   * Describe or define the storage access protocol used to access the data servers
* Describe the methods of recovery from storage device restart, and loss of layout state on the metadata server (see Section 12.7.3).

* Include a security considerations section

3. The author documents the new layout specification as an Internet Draft.

4. The author submits the Internet Draft for review through the IETF standards process as defined in "Internet Official Protocol Standards" (STD 1). The new layout specification will be submitted for eventual publication as a standards track RFC.

5. The layout specification progresses through the IETF standards process; the new option will be reviewed by the NFSv4 Working Group (if that group still exists), or as an Internet Draft not submitted by an IETF working group.

22. References

22.1. Normative References


22.2. Informative References


Appendix A.  Acknowledgments

The initial drafts for the SECINFO extensions were edited by Mike Eisler with contributions from Peng Dai, Sergey Klyushin, and Carl Burnett.

The initial drafts for the SESSIONS extensions were edited by Tom Talpey, Spencer Shepler, Jon Bauman with contributions from Charles Antonelli, Brent Callaghan, Mike Eisler, John Howard, Chet Juszczak,
Trond Myklebust, Dave Noveck, John Scott, Mike Stolarchuk and Mark Wittle.  [[Comment.22: global namespace stuff?]]

The initial drafts for the Directory Delegations support were contributed by Saadia Khan with input from Dave Noveck, Mike Eisler, Carl Burnett, Ted Anderson and Tom Talpey.

The initial drafts for the ACL explanations were contributed by Sam Falkner and Lisa Week.

The initial drafts for the parallel NFS support were edited by Brent Welch and Garth Goodson. Additional authors for those documents were Benny Halevy, David Black, and Andy Adamson. Additional input came from the informal group which contributed to the construction of the initial pNFS drafts; specific acknowledgement goes to Gary Grider, Peter Corbett, Dave Noveck, and Peter Honeyman. The pNFS work was inspired by the NASD and OSD work done by Garth Gibson. Gary Grider of the national labs (LANL) has also been a champion of high-performance parallel I/O.

Fredric Isaman found several errors in draft versions of the ONC RPC XDR description of the NFSv4.1 protocol.

Authors' Addresses

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

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

Mike Eisler
Network Appliance, Inc.
5765 Chase Point Circle
Colorado Springs, CO  80919
USA

Phone: +1-719-599-9026
Email: email2mre@yahoo.com
URI: Insert ietf2 between the - and @ symbols in the above address
David Noveck  
Network Appliance, Inc.  
1601 Trapelo Road, Suite 16  
Waltham, MA 02454  
USA  

Phone: +1-781-768-5347  
Email: dnoveck@netapp.com