

NFSv4
Internet-Draft
Intended status: Standards Track
Expires: March 9, 2009

S. Shepler
M. Eisler
D. Noveck
Editors
September 05, 2008

NFS Version 4 Minor Version 1
draft-ietf-nfsv4-minorversion1-26.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/lid-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 March 9, 2009.

Abstract

This Internet-Draft describes NFS version 4 minor version one, including features retained from the base protocol and protocol extensions made subsequently. Major extensions introduced in NFS version 4 minor version one include: Sessions, Directory Delegations, and parallel NFS (pNFS).

Requirements Language

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

Internet-Draft

NFSv4.1

September 2008

Table of Contents

1.	Introduction	11
1.1.	The NFS Version 4 Minor Version 1 Protocol	11
1.2.	Scope of this Document	11
1.3.	NFSv4 Goals	11
1.4.	NFSv4.1 Goals	12
1.5.	General Definitions	12
1.6.	Overview of NFSv4.1 Features	15
1.6.1.	RPC and Security	15
1.6.2.	Protocol Structure	15
1.6.3.	File System Model	16
1.6.4.	Locking Facilities	18
1.7.	Differences from NFSv4.0	19
2.	Core Infrastructure	20
2.1.	Introduction	20
2.2.	RPC and XDR	20
2.2.1.	RPC-based Security	20
2.3.	COMPOUND and CB_COMPOUND	23
2.4.	Client Identifiers and Client Owners	24
2.4.1.	Upgrade from NFSv4.0 to NFSv4.1	27
2.4.2.	Server Release of Client ID	28
2.4.3.	Resolving Client Owner Conflicts	28
2.5.	Server Owners	29
2.6.	Security Service Negotiation	30
2.6.1.	NFSv4.1 Security Tuples	30
2.6.2.	SECINFO and SECINFO_NO_NAME	31
2.6.3.	Security Error	31
2.7.	Minor Versioning	35
2.8.	Non-RPC-based Security Services	38
2.8.1.	Authorization	38
2.8.2.	Auditing	38
2.8.3.	Intrusion Detection	38
2.9.	Transport Layers	39
2.9.1.	REQUIRED and RECOMMENDED Properties of Transports	39
2.9.2.	Client and Server Transport Behavior	39
2.9.3.	Ports	41
2.10.	Session	41
2.10.1.	Motivation and Overview	41
2.10.2.	NFSv4 Integration	42
2.10.3.	Channels	44
2.10.4.	Server Scope	45
2.10.5.	Trunking	47

2.10.6.	Exactly Once Semantics	51
2.10.7.	RDMA Considerations	64
2.10.8.	Sessions Security	66
2.10.9.	The SSV GSS Mechanism	72
2.10.10.	Session Mechanics – Steady State	76

2.10.11.	Session Inactivity Timer	78
2.10.12.	Session Mechanics – Recovery	78
2.10.13.	Parallel NFS and Sessions	83
3.	Protocol Constants and Data Types	83
3.1.	Basic Constants	84
3.2.	Basic Data Types	84
3.3.	Structured Data Types	86
4.	Filehandles	94
4.1.	Obtaining the First Filehandle	95
4.1.1.	Root Filehandle	95
4.1.2.	Public Filehandle	95
4.2.	Filehandle Types	96
4.2.1.	General Properties of a Filehandle	96
4.2.2.	Persistent Filehandle	97
4.2.3.	Volatile Filehandle	97
4.3.	One Method of Constructing a Volatile Filehandle	98
4.4.	Client Recovery from Filehandle Expiration	99
5.	File Attributes	100
5.1.	REQUIRED Attributes	101
5.2.	RECOMMENDED Attributes	101
5.3.	Named Attributes	102
5.4.	Classification of Attributes	103
5.5.	Set-Only and Get-Only Attributes	104
5.6.	REQUIRED Attributes – List and Definition References	104
5.7.	RECOMMENDED Attributes – List and Definition References	105
5.8.	Attribute Definitions	107
5.8.1.	Definitions of REQUIRED Attributes	107
5.8.2.	Definitions of Uncategorized RECOMMENDED Attributes	109
5.9.	Interpreting owner and owner_group	116
5.10.	Character Case Attributes	118
5.11.	Directory Notification Attributes	118
5.12.	pNFS Attribute Definitions	118
5.13.	Retention Attributes	120
6.	Access Control Attributes	123

6.1.	Goals	123
6.2.	File Attributes Discussion	124
6.2.1.	Attribute 12: acl	124
6.2.2.	Attribute 58: dacl	139
6.2.3.	Attribute 59: sacl	139
6.2.4.	Attribute 33: mode	139
6.2.5.	Attribute 74: mode_set_masked	140
6.3.	Common Methods	141
6.3.1.	Interpreting an ACL	141
6.3.2.	Computing a Mode Attribute from an ACL	142
6.4.	Requirements	143
6.4.1.	Setting the mode and/or ACL Attributes	143

6.4.2.	Retrieving the mode and/or ACL Attributes	145
6.4.3.	Creating New Objects	145
7.	Single-server Namespace	149
7.1.	Server Exports	149
7.2.	Browsing Exports	150
7.3.	Server Pseudo File System	150
7.4.	Multiple Roots	151
7.5.	Filehandle Volatility	151
7.6.	Exported Root	151
7.7.	Mount Point Crossing	152
7.8.	Security Policy and Namespace Presentation	152
8.	State Management	153
8.1.	Client and Session ID	154
8.2.	Stateid Definition	154
8.2.1.	Stateid Types	155
8.2.2.	Stateid Structure	156
8.2.3.	Special Stateids	158
8.2.4.	Stateid Lifetime and Validation	159
8.2.5.	Stateid Use for I/O Operations	162
8.2.6.	Stateid Use for SETATTR Operations	163
8.3.	Lease Renewal	163
8.4.	Crash Recovery	165
8.4.1.	Client Failure and Recovery	166
8.4.2.	Server Failure and Recovery	167
8.4.3.	Network Partitions and Recovery	171
8.5.	Server Revocation of Locks	176
8.6.	Short and Long Leases	177
8.7.	Clocks, Propagation Delay, and Calculating Lease Expiration	177

8.8.	Obsolete Locking Infrastructure From NFSv4.0	178
9.	File Locking and Share Reservations	179
9.1.	Opens and Byte-Range Locks	179
9.1.1.	State-owner Definition	179
9.1.2.	Use of the Stateid and Locking	179
9.2.	Lock Ranges	182
9.3.	Upgrading and Downgrading Locks	183
9.4.	Stateid Seqid Values and Byte-Range Locks	183
9.5.	Issues with Multiple Open-Owners	184
9.6.	Blocking Locks	184
9.7.	Share Reservations	185
9.8.	OPEN/CLOSE Operations	186
9.9.	Open Upgrade and Downgrade	187
9.10.	Parallel OPENS	188
9.11.	Reclaim of Open and Byte-Range Locks	188
10.	Client-Side Caching	189
10.1.	Performance Challenges for Client-Side Caching	189
10.2.	Delegation and Callbacks	190
10.2.1.	Delegation Recovery	192

10.3.	Data Caching	195
10.3.1.	Data Caching and OPENS	195
10.3.2.	Data Caching and File Locking	196
10.3.3.	Data Caching and Mandatory File Locking	198
10.3.4.	Data Caching and File Identity	198
10.4.	Open Delegation	199
10.4.1.	Open Delegation and Data Caching	202
10.4.2.	Open Delegation and File Locks	203
10.4.3.	Handling of CB_GETATTR	203
10.4.4.	Recall of Open Delegation	206
10.4.5.	Clients that Fail to Honor Delegation Recalls	208
10.4.6.	Delegation Revocation	209
10.4.7.	Delegations via WANT_DELEGATION	209
10.5.	Data Caching and Revocation	210
10.5.1.	Revocation Recovery for Write Open Delegation	211
10.6.	Attribute Caching	211
10.7.	Data and Metadata Caching and Memory Mapped Files	213
10.8.	Name and Directory Caching without Directory Delegations	216
10.8.1.	Name Caching	216
10.8.2.	Directory Caching	217
10.9.	Directory Delegations	218

10.9.1.	Introduction to Directory Delegations	218
10.9.2.	Directory Delegation Design	219
10.9.3.	Attributes in Support of Directory Notifications	220
10.9.4.	Directory Delegation Recall	220
10.9.5.	Directory Delegation Recovery	221
11.	Multi-Server Namespace	221
11.1.	Location Attributes	222
11.2.	File System Presence or Absence	222
11.3.	Getting Attributes for an Absent File System	223
11.3.1.	GETATTR Within an Absent File System	224
11.3.2.	REaddir and Absent File Systems	225
11.4.	Uses of Location Information	225
11.4.1.	File System Replication	226
11.4.2.	File System Migration	227
11.4.3.	Referrals	228
11.5.	Location Entries and Server Identity	230
11.6.	Additional Client-side Considerations	230
11.7.	Effecting File System Transitions	231
11.7.1.	File System Transitions and Simultaneous Access	232
11.7.2.	Simultaneous Use and Transparent Transitions	233
11.7.3.	Filehandles and File System Transitions	236
11.7.4.	Fileids and File System Transitions	236
11.7.5.	Fsids and File System Transitions	237
11.7.6.	The Change Attribute and File System Transitions	238
11.7.7.	Lock State and File System Transitions	238
11.7.8.	Write Verifiers and File System Transitions	243

11.7.9.	Readdir Cookies and Verifiers and File System Transitions	243
11.7.10.	File System Data and File System Transitions	243
11.8.	Effecting File System Referrals	245
11.8.1.	Referral Example (LOOKUP)	245
11.8.2.	Referral Example (REaddir)	249
11.9.	The Attribute fs_locations	251
11.10.	The Attribute fs_locations_info	254
11.10.1.	The fs_locations_server4 Structure	258
11.10.2.	The fs_locations_info4 Structure	263
11.10.3.	The fs_locations_item4 Structure	264
11.11.	The Attribute fs_status	266
12.	Parallel NFS (pNFS)	270
12.1.	Introduction	270
12.2.	pNFS Definitions	272

12.2.1.	Metadata	273
12.2.2.	Metadata Server	273
12.2.3.	pNFS Client	273
12.2.4.	Storage Device	273
12.2.5.	Storage Protocol	273
12.2.6.	Control Protocol	273
12.2.7.	Layout Types	274
12.2.8.	Layout	274
12.2.9.	Layout Iomode	275
12.2.10.	Device IDs	275
12.3.	pNFS Operations	277
12.4.	pNFS Attributes	278
12.5.	Layout Semantics	278
12.5.1.	Guarantees Provided by Layouts	278
12.5.2.	Getting a Layout	279
12.5.3.	Layout Stateid	280
12.5.4.	Committing a Layout	281
12.5.5.	Recalling a Layout	284
12.5.6.	Revoking Layouts	292
12.5.7.	Metadata Server Write Propagation	293
12.6.	pNFS Mechanics	293
12.7.	Recovery	294
12.7.1.	Recovery from Client Restart	295
12.7.2.	Dealing with Lease Expiration on the Client	295
12.7.3.	Dealing with Loss of Layout State on the Metadata Server	296
12.7.4.	Recovery from Metadata Server Restart	297
12.7.5.	Operations During Metadata Server Grace Period	299
12.7.6.	Storage Device Recovery	299
12.8.	Metadata and Storage Device Roles	299
12.9.	Security Considerations for pNFS	300
13.	PNFS: NFSv4.1 File Layout Type	301
13.1.	Client ID and Session Considerations	301

13.1.1.	Sessions Considerations for Data Servers	303
13.2.	File Layout Definitions	304
13.3.	File Layout Data Types	305
13.4.	Interpreting the File Layout	309
13.4.1.	Determining the Stripe Unit Number	309
13.4.2.	Interpreting the File Layout Using Sparse Packing	309
13.4.3.	Interpreting the File Layout Using Dense Packing	311
13.4.4.	Sparse and Dense Stripe Unit Packing	314

13.5.	Data Server Multipathing	315
13.6.	Operations Sent to NFSv4.1 Data Servers	316
13.7.	COMMIT Through Metadata Server	319
13.8.	The Layout Iomode	320
13.9.	Metadata and Data Server State Coordination	320
13.9.1.	Global Stateid Requirements	320
13.9.2.	Data Server State Propagation	321
13.10.	Data Server Component File Size	323
13.11.	Layout Revocation and Fencing	324
13.12.	Security Considerations for the File Layout Type	325
14.	Internationalization	326
14.1.	Stringprep profile for the utf8str_cs type	327
14.2.	Stringprep profile for the utf8str_cis type	328
14.3.	Stringprep profile for the utf8str_mixed type	330
14.4.	UTF-8 Capabilities	331
14.5.	UTF-8 Related Errors	331
15.	Error Values	332
15.1.	Error Definitions	332
15.1.1.	General Errors	334
15.1.2.	Filehandle Errors	336
15.1.3.	Compound Structure Errors	338
15.1.4.	File System Errors	339
15.1.5.	State Management Errors	341
15.1.6.	Security Errors	342
15.1.7.	Name Errors	343
15.1.8.	Locking Errors	343
15.1.9.	Reclaim Errors	345
15.1.10.	pNFS Errors	345
15.1.11.	Session Use Errors	347
15.1.12.	Session Management Errors	348
15.1.13.	Client Management Errors	348
15.1.14.	Delegation Errors	349
15.1.15.	Attribute Handling Errors	350
15.1.16.	Obsoleted Errors	350
15.2.	Operations and their valid errors	351
15.3.	Callback operations and their valid errors	367
15.4.	Errors and the operations that use them	369
16.	NFSv4.1 Procedures	383
16.1.	Procedure 0: NULL - No Operation	383
16.2.	Procedure 1: COMPOUND - Compound Operations	384

17.	Operations: REQUIRED, RECOMMENDED, or OPTIONAL	395
---------------------	--	---------------------

<u>18.</u>	<u>NFSv4.1 Operations</u>	<u>398</u>
<u>18.1.</u>	<u>Operation 3: ACCESS – Check Access Rights</u>	<u>398</u>
<u>18.2.</u>	<u>Operation 4: CLOSE – Close File</u>	<u>404</u>
<u>18.3.</u>	<u>Operation 5: COMMIT – Commit Cached Data</u>	<u>405</u>
<u>18.4.</u>	<u>Operation 6: CREATE – Create a Non-Regular File Object</u>	<u>408</u>
<u>18.5.</u>	<u>Operation 7: DELEGPURGE – Purge Delegations Awaiting Recovery</u>	<u>411</u>
<u>18.6.</u>	<u>Operation 8: DELEGRETURN – Return Delegation</u>	<u>412</u>
<u>18.7.</u>	<u>Operation 9: GETATTR – Get Attributes</u>	<u>412</u>
<u>18.8.</u>	<u>Operation 10: GETFH – Get Current Filehandle</u>	<u>414</u>
<u>18.9.</u>	<u>Operation 11: LINK – Create Link to a File</u>	<u>415</u>
<u>18.10.</u>	<u>Operation 12: LOCK – Create Lock</u>	<u>418</u>
<u>18.11.</u>	<u>Operation 13: LOCKT – Test For Lock</u>	<u>422</u>
<u>18.12.</u>	<u>Operation 14: LOCKU – Unlock File</u>	<u>423</u>
<u>18.13.</u>	<u>Operation 15: LOOKUP – Lookup Filename</u>	<u>425</u>
<u>18.14.</u>	<u>Operation 16: LOOKUPP – Lookup Parent Directory</u>	<u>426</u>
<u>18.15.</u>	<u>Operation 17: NVERIFY – Verify Difference in Attributes</u>	<u>428</u>
<u>18.16.</u>	<u>Operation 18: OPEN – Open a Regular File</u>	<u>429</u>
<u>18.17.</u>	<u>Operation 19: OPENATTR – Open Named Attribute Directory</u>	<u>448</u>
<u>18.18.</u>	<u>Operation 21: OPEN_DOWNGRADE – Reduce Open File Access</u>	<u>449</u>
<u>18.19.</u>	<u>Operation 22: PUTFH – Set Current Filehandle</u>	<u>451</u>
<u>18.20.</u>	<u>Operation 23: PUTPUBFH – Set Public Filehandle</u>	<u>451</u>
<u>18.21.</u>	<u>Operation 24: PUTROOTFH – Set Root Filehandle</u>	<u>453</u>
<u>18.22.</u>	<u>Operation 25: READ – Read from File</u>	<u>454</u>
<u>18.23.</u>	<u>Operation 26: REaddir – Read Directory</u>	<u>456</u>
<u>18.24.</u>	<u>Operation 27: READLINK – Read Symbolic Link</u>	<u>460</u>
<u>18.25.</u>	<u>Operation 28: REMOVE – Remove File System Object</u>	<u>461</u>
<u>18.26.</u>	<u>Operation 29: RENAME – Rename Directory Entry</u>	<u>463</u>
<u>18.27.</u>	<u>Operation 31: RESTOREFH – Restore Saved Filehandle</u>	<u>467</u>
<u>18.28.</u>	<u>Operation 32: SAVEFH – Save Current Filehandle</u>	<u>468</u>
<u>18.29.</u>	<u>Operation 33: SECINFO – Obtain Available Security</u>	<u>469</u>
<u>18.30.</u>	<u>Operation 34: SETATTR – Set Attributes</u>	<u>473</u>
<u>18.31.</u>	<u>Operation 37: VERIFY – Verify Same Attributes</u>	<u>476</u>
<u>18.32.</u>	<u>Operation 38: WRITE – Write to File</u>	<u>477</u>
<u>18.33.</u>	<u>Operation 40: BACKCHANNEL_CTL – Backchannel Control</u>	<u>481</u>
<u>18.34.</u>	<u>Operation 41: BIND_CONN_TO_SESSION – Associate Connection with Session</u>	<u>483</u>
<u>18.35.</u>	<u>Operation 42: EXCHANGE_ID – Instantiate Client ID</u>	<u>486</u>
<u>18.36.</u>	<u>Operation 43: CREATE_SESSION – Create New Session and Confirm Client ID</u>	<u>503</u>
<u>18.37.</u>	<u>Operation 44: DESTROY_SESSION – Destroy a Session</u>	<u>513</u>
<u>18.38.</u>	<u>Operation 45: FREE_STATEID – Free Stateid with No Locks</u>	<u>514</u>
<u>18.39.</u>	<u>Operation 46: GET_DIR_DELEGATION – Get a directory delegation</u>	<u>515</u>

18.40.	Operation 47: GETDEVICEINFO - Get Device Information . .	519
18.41.	Operation 48: GETDEVICELIST - Get All Device Mappings for a File System	521
18.42.	Operation 49: LAYOUTCOMMIT - Commit Writes Made Using a Layout	523
18.43.	Operation 50: LAYOUTGET - Get Layout Information	526
18.44.	Operation 51: LAYOUTRETURN - Release Layout Information	536
18.45.	Operation 52: SECINFO_NO_NAME - Get Security on Unnamed Object	540
18.46.	Operation 53: SEQUENCE - Supply Per-Procedure Sequencing and Control	541
18.47.	Operation 54: SET_SSV - Update SSV for a Client ID . . .	547
18.48.	Operation 55: TEST_STATEID - Test Stateids for Validity	549
18.49.	Operation 56: WANT_DELEGATION - Request Delegation . . .	551
18.50.	Operation 57: DESTROY_CLIENTID - Destroy a Client ID . .	555
18.51.	Operation 58: RECLAIM_COMPLETE - Indicates Reclaims Finished	555
18.52.	Operation 10044: ILLEGAL - Illegal operation	558
19.	NFSv4.1 Callback Procedures	558
19.1.	Procedure 0: CB_NULL - No Operation	559
19.2.	Procedure 1: CB_COMPOUND - Compound Operations	559
20.	NFSv4.1 Callback Operations	563
20.1.	Operation 3: CB_GETATTR - Get Attributes	563
20.2.	Operation 4: CB_RECALL - Recall a Delegation	564
20.3.	Operation 5: CB_LAYOUTRECALL - Recall Layout from Client	565
20.4.	Operation 6: CB_NOTIFY - Notify Client of Directory Changes	569
20.5.	Operation 7: CB_PUSH_DELEG - Offer Previously Requested Delegation to Client	573
20.6.	Operation 8: CB_RECALL_ANY - Keep Any N Recallable Objects	574
20.7.	Operation 9: CB_RECALLABLE_OBJ_AVAIL - Signal Resources for Recallable Objects	577
20.8.	Operation 10: CB_RECALL_SLOT - Change Flow Control Limits	578
20.9.	Operation 11: CB_SEQUENCE - Supply Backchannel Sequencing and Control	579
20.10.	Operation 12: CB_WANTS_CANCELLED - Cancel Pending Delegation Wants	581
20.11.	Operation 13: CB_NOTIFY_LOCK - Notify Client of Possible Lock Availability	582
20.12.	Operation 14: CB_NOTIFY_DEVICEID - Notify Client of Device ID Changes	584

20.13. Operation 10044: CB_ILLEGAL – Illegal Callback	
Operation	586

21. Security Considerations	586
22. IANA Considerations	588
22.1. Named Attribute Definitions	588
22.1.1. Initial Registry	589
22.1.2. Updating Registrations	589
22.2. Device ID Notifications	589
22.2.1. Initial Registry	590
22.2.2. Updating Registrations	590
22.3. Object Recall Types	590
22.3.1. Initial Registry	592
22.3.2. Updating Registrations	592
22.4. Layout Types	592
22.4.1. Initial Registry	593
22.4.2. Updating Registrations	593
22.4.3. Guidelines for Writing Layout Type Specifications .	593
22.5. Path Variable Definitions	595
22.5.1. Path Variables Registry	595
22.5.2. Values for the \${ietf.org:CPU_ARCH} Variable	597
22.5.3. Values for the \${ietf.org:OS_TYPE} Variable	597
23. References	598
23.1. Normative References	598
23.2. Informative References	600
Appendix A. Acknowledgments	601
Appendix B. RFC Editor Notes	603
Authors' Addresses	604
Intellectual Property and Copyright Statements	605

Internet-Draft

NFSv4.1

September 2008

[1.](#) Introduction

[1.1.](#) The NFS Version 4 Minor Version 1 Protocol

The NFS version 4 minor version 1 (NFSv4.1) protocol is the second minor version of the NFS version 4 (NFSv4) protocol. The first minor version, NFSv4.0 is described in [\[20\]](#). It generally follows the guidelines for minor versioning model listed 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 anything but REQUIRED would add undue complexity to protocol definition and implementation. NFSv4.1 accordingly updates the Minor Versioning guidelines ([Section 2.7](#)).

As a minor version, NFSv4.1 is consistent with the overall goals for NFSv4, 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 NFSv4.1.

[1.2.](#) Scope of this Document

This document describes the NFSv4.1 protocol. With respect to NFSv4.0, this document does not:

- o describe the NFSv4.0 protocol, except where needed to contrast with NFSv4.1.

- o modify the specification of the NFSv4.0 protocol.
- o clarify the NFSv4.0 protocol.

1.3. NFSv4 Goals

The NFSv4 protocol is a further revision of the NFS protocol defined already by NFSv3 [21]. It retains the essential characteristics of previous versions: easy recovery; independence of transport protocols, operating systems and file systems; simplicity; and good performance. NFSv4 has the following goals:

- o Improved access and good performance on the Internet.

The protocol is designed to transit firewalls easily, perform well

where latency is high and bandwidth is low, and scale to very large numbers of clients per server.

- o Strong security with negotiation built into the protocol.

The protocol builds on the work of the ONCRPC working group in supporting the RPCSEC_GSS protocol. Additionally, the NFSv4.1 protocol provides a mechanism to allow clients and servers the ability to negotiate security and require clients and servers to support a minimal set of security schemes.

- o Good cross-platform interoperability.

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

- o Designed for protocol extensions.

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

1.4. NFSv4.1 Goals

NFSv4.1 has the following goals, within the framework established by

the overall NFSv4 goals.

- o To correct significant structural weaknesses and oversights discovered in the base protocol.
- o To add clarity and specificity to areas left unaddressed or not addressed in sufficient detail in the base protocol. However, as stated in [Section 1.2](#), it is not a goal to clarify the NFSv4.0 protocol in the NFSv4.1 specification.
- o To add specific features based on experience with the existing protocol and recent industry developments.
- o 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.

[1.5](#). General Definitions

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

Byte This document defines a byte as an octet, i.e. a datum exactly 8 bits in length.

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

A client is uniquely identified by a Client Owner.

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

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

node.

Client ID A 64-bit quantity used as a unique, short-hand reference to a client supplied Verifier and 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 from those connections may share a client owner. The server is expected to treat requests from connections with the same client owner as coming from the same client.

File System The collection of objects on a server (as identified by the major identifier of a Server Owner, which is defined later in this section), that share the same fsid attribute (see [Section 5.8.1.9](#)).

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

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

Lock The term "lock" is used to refer to byte-range (in UNIX environments, also known as record) 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 and is identified by a Server owner. A server can span multiple network addresses.

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

identifier, the client assumes both peers are the same server (the server namespace is the same via each connection), and assumes and lock state is sharable across both connections. When each peer has both the same major and minor identifier, the client assumes each connection might be associable with the same session.

Stable Storage NFSv4.1 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-owner or lock-owner/open-owner pair for a specific file and type of lock.

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

[1.6.](#) Overview of NFSv4.1 Features

To provide a reasonable context for the reader, the major features of the NFSv4.1 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 [\[2\]](#) and [\[3\]](#). A basic knowledge of file systems and distributed file systems is expected as well.

In general this specification of NFSv4.1 will not distinguish those added in minor version one from those present in the base protocol but will treat NFSv4.1 as a unified whole. See [Section 1.7](#) for a summary of the differences between NFSv4.0 and NFSv4.1.

[1.6.1.](#) RPC and Security

As with previous versions of NFS, the External Data Representation (XDR) and Remote Procedure Call (RPC) mechanisms used for the NFSv4.1 protocol are those defined in [\[2\]](#) and [\[3\]](#). To meet end-to-end security requirements, the RPCSEC_GSS framework [\[4\]](#) 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 NFSv4 protocol. Kerberos V5 will be used as described in [\[5\]](#) to provide one security framework. The LIPKEY and SPKM-3 GSS-API mechanisms described in [\[6\]](#) will be used to provide for the use of user password and client/server public key certificates by the NFSv4 protocol. With the use of RPCSEC_GSS, other mechanisms may also be specified and used for NFSv4.1 security.

To enable in-band security negotiation, the NFSv4.1 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.

[1.6.2.](#) Protocol Structure

[1.6.2.1.](#) Core Protocol

Unlike NFSv3, which used a series of ancillary protocols (e.g. NLM, NSM, MOUNT), within all minor versions of NFSv4 a single RPC protocol is used to make requests to the server. Facilities that had been separate protocols, such as locking, are now integrated within a single unified protocol.

[1.6.2.2.](#) 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.

[1.6.3.](#) File System Model

The general file system model used for the NFSv4.1 protocol is the same as previous versions. The server file system is hierarchical with the regular files contained within being treated as opaque byte streams. In a slight departure, file and directory names are encoded with UTF-8 to deal with the basics of internationalization.

The NFSv4.1 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.

[1.6.3.1.](#) Filehandles

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

The NFSv4.1 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

Internet-Draft

NFSv4.1

September 2008

filehandles with more limited validity guarantees, called volatile filehandles.

[1.6.3.2](#). File Attributes

The NFSv4.1 protocol has a rich and extensible file object attribute structure, which is divided into REQUIRED, RECOMMENDED, and named attributes (see [Section 5](#)).

Several (but not all) of the REQUIRED attributes are derived from the attributes of NFSv3 (see definition of the `fattr3` data type in [\[21\]](#)). An example of a REQUIRED attribute is the file object's type ([Section 5.8.1.2](#)) so that regular files can be distinguished from directories (also known as folders in some operating environments) and other types of objects. REQUIRED attributes are discussed in [Section 5.1](#).

An example of three RECOMMENDED attributes are `acl`, `sacl`, and `dacl`. These attributes define an Access Control List (ACL) on a file object ([Section 6](#)). An ACL provides directory and file access control beyond the model used in NFSv3. The ACL definition allows for specification of specific sets of permissions for individual users and groups. In addition, ACL inheritance allows propagation of access permissions and restriction down a directory tree as file system objects are created. RECOMMENDED attributes are discussed in [Section 5.2](#).

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

[1.6.3.3](#). Multi-server Namespace

NFSv4.1 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 systems, which provide specifications for additional locations but no actual file system content. This allows a number of important

facilities:

- o Location attributes may be used with absent file systems to implement referrals whereby one server may direct the client to a file system provided by another server. This allows extensive multi-server namespaces to be constructed.
- o Location attributes may be provided for present file systems to provide the locations of alternate file system instances or replicas to be used in the event that the current file system instance becomes unavailable.
- o Location attributes may be provided when a previously present file system becomes absent. This allows non-disruptive migration of file systems to alternate servers.

1.6.4. 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.

The types of locks are:

- o Share reservations as established by OPEN operations.
- o Byte-range locks.

- o 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.
- o Directory delegations, which are recallable locks that assure the holder that inconsistent directory modifications cannot occur so long as the delegation is held.
- o 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 locks are subject to revocation. In the event of server restart, clients have the opportunity to safely reclaim their locks within a special grace period.

[1.7.](#) Differences from NFSv4.0

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

- o Implementation of the sessions model ([Section 2.10](#)).
- o Parallel access to data ([Section 12](#)).
- o Addition of the RECLAIM_COMPLETE operation to better structure the lock reclamation process ([Section 18.51](#)).
- o Enhanced delegation support as follows.
 - * Delegations on directories and other file types in addition to regular files ([Section 18.39](#), [Section 18.49](#)).
 - * Operations to optimize acquisition of recalled or denied delegations ([Section 18.49](#), [Section 20.5](#), [Section 20.7](#)).
 - * Notifications of changes to files and directories

([Section 18.39](#), [Section 20.4](#)).

- * A method to allow a server to indicate it is recalling one or more delegations for resource management reasons, and thus a method to allow the client to pick which delegations to return ([Section 20.6](#)).
- o Attributes can be set atomically during exclusive file create via the OPEN operation (see the new EXCLUSIVE4_1 creation method in [Section 18.16](#)).
- o Open files can be preserved if removed and the hard link count goes to zero thus obviating the need for clients to rename deleted files to partially hidden names -- colloquially called "silly rename" (see the new OPEN4_RESULT_PRESERVE_UNLINKED reply flag in [Section 18.16](#)).
- o Improved compatibility with Microsoft Windows for Access Control Lists ([Section 6.2.3](#), [Section 6.2.2](#), [Section 6.4.3.2](#)).
- o Data retention ([Section 5.13](#)).

- o Identification of the implementation of the NFS client and server ([Section 18.35](#)).
- o Support for notification of the availability of byte-range locks (see the new OPEN4_RESULT_MAY_NOTIFY_LOCK reply flag in [Section 18.16](#) and see [Section 20.11](#)).

[2.](#) Core Infrastructure

[2.1.](#) Introduction

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 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 [3] and [2].

[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 trusts the client to authenticate all users. Actually, NFS has always depended on RPC for authentication. One of the first forms of RPC authentication, AUTH_SYS, had no strong authentication, and required a host-based authentication approach. NFSv4.1 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 NFSv4.1.

[2.2.1.1.](#) RPC Security Flavors

As described in [section 7.2](#) "Authentication" of [3], RPC security is encapsulated in the RPC header, via a security or authentication flavor, and information specific to the specified 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.

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

[2.2.1.1.1.](#) RPCSEC_GSS and Security Services

RPCSEC_GSS ([4]) uses the functionality of GSS-API [7]. 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 [4] for more information.

NFSv4.1 client and servers MUST support RPCSEC_GSS's integrity and authentication service. NFSv4.1 servers MUST support RPCSEC_GSS's privacy service. NFSv4.1 clients SHOULD support RPCSEC_GSS's privacy service.

2.2.1.1.1.2. Security mechanisms for NFSv4.1

RPCSEC_GSS, via GSS-API, normalizes access to mechanisms that provide security services. Therefore NFSv4.1 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, NFSv4.1 specifies that a QOP of zero (0) is used, leaving it up to the mechanism or the mechanism's configuration to map QOP zero to an appropriate level of protection. Each mandated mechanism specifies minimum set of cryptographic algorithms for implementing integrity and privacy. NFSv4.1 clients and servers MUST be implemented on operating environments that comply with the REQUIRED cryptographic algorithms of each REQUIRED

mechanism.

2.2.1.1.1.2.1. Kerberos V5

The Kerberos V5 GSS-API mechanism as described in [5] 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

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

Note that the number and name of the pseudo flavor is presented here as a mapping aid to the implementor. Because the NFSv4.1 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 NFSv3 since the security negotiation is done via the MOUNT protocol as described in [22].

[2.2.1.1.1.2.2.](#) LIPKEY

The LIPKEY V5 GSS-API mechanism as described in [6] MUST be implemented with the RPCSEC_GSS services as specified in the following table:

1	2	3	4	5	6
390006	lipkey	1.3.6.1.5.5.9	rpc_gss_svc_none	yes	yes
390007	lipkey-i	1.3.6.1.5.5.9	rpc_gss_svc_integrity	yes	yes
390008	lipkey-p	1.3.6.1.5.5.9	rpc_gss_svc_privacy	no	yes

[2.2.1.1.1.2.3.](#) SPKM-3 as a security triple

The SPKM-3 GSS-API mechanism as described in [6] MUST be implemented with the RPCSEC_GSS services as specified in the following table:

1	2	3	4	5	6
-----	-----	-----	-----	-----	-----
390009	spkm3	1.3.6.1.5.5.1.3	rpc_gss_svc_none	yes	yes
390010	spkm3i	1.3.6.1.5.5.1.3	rpc_gss_svc_integrity	yes	yes
390011	spkm3p	1.3.6.1.5.5.1.3	rpc_gss_svc_privacy	no	yes

[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 NFSv4 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.

Internet-Draft

NFSv4.1

September 2008

NFSv4.1 also contains a considerable set of callback operations in which the server makes an RPC directed at the client. Callback RPCs have a similar structure to that of the normal server requests. In all minor versions of the NFSv4 protocol there are two callback RPC procedures, CB_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.

The addition of new server and callback operations within the COMPOUND and CB_COMPOUND request framework provides a 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 reply 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 identifiable by the server.

Each distinct client instance is represented by a client ID. A client ID is a 64-bit identifier representing a specific client at a given time. The client ID is changed whenever the client re-initializes, and may change when the server re-initializes. Client IDs are used to support lock identification and crash recovery.

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 sent. A session is associated with a client ID when the session is created.

Unlike NFSv4.0, the only NFSv4.1 operations possible before a client ID is established are those needed to establish 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 and confirm the client ID 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, layout state, and where the server is not supporting the CLAIM_DELEGATE_PREV claim type, all delegation state associated with the same client with the same identity. For discussion of delegation state recovery, see [Section 10.2.1](#). For discussion of layout state

recovery see [Section 12.7.1](#).

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 8.3](#)) 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 data type:

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

The first field, `co_verifier`, is a client incarnation verifier. The server will start the process of canceling the client's leased state if `co_verifier` is different than what the server has previously recorded for the identified client (as specified in the `co_ownerid` field).

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:

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

anceled.

- o The string should be selected so that subsequent incarnations (e.g. restarts) 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 NFSv4.1 server.
- o The string should be the same for each server network address that the client accesses. This way, if a server has multiple interfaces, the client can trunk traffic over multiple network

paths as described in [Section 2.10.5](#). (Note: the precise opposite was advised in the NFSv4.0 specification [[20](#)].)

- o 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. Thus with dynamic address assignment, if the client includes just the client's network address in the co_ownerid string, there is a real risk 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:

- o If applicable, the client's statically assigned network address.
- o Additional information that tends to be unique, such as one or more of:
 - * The client machine's serial number (for privacy reasons, it is best to perform some one way function on the serial number).
 - * A MAC address (again, a one way function should be performed).

- * The timestamp of when the NFSv4.1 software was first installed on the client (though this is subject to the previously mentioned caution about using information that is stored in a file, because the file might only be accessible over NFSv4.1).
- * A true random number. However since this number ought to be the same between client incarnations, this shares the same problem as that of using the timestamp of the software installation.
- o For a user level NFSv4.1 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.

The 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.

In the event of a server restart, a client may find out that its current client ID is no longer valid when it receives an `NFS4ERR_STALE_CLIENTID` error. The precise circumstances depend on the characteristics of the sessions involved, specifically whether the session is persistent (see [Section 2.10.6.5](#)), but in each case the client will receive this error when it attempts to establish a new session with the existing client ID and receives the error `NFS4ERR_STALE_CLIENTID`, indicating that a new client ID must be obtained via `EXCHANGE_ID` and the new session established with that client ID.

When a session is not persistent, the client will find out that it needs to create a new session as a result of getting an `NFS4ERR_BADSESSION`, since the session in question was lost as part of a server restart. 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 restart, the server will reject the request with the error `NFS4ERR_STALE_CLIENTID`.

In the case of the session being persistent, the client will re-establish communication using the existing session after the restart.

This session will be associated with the existing client ID but may only be used to retransmit operations that the client previously transmitted and did not see replies to. Replies to operations that the server previously performed will come from the reply cache, otherwise NFS4ERR_DEADSESSION will be returned. Hence, such a session is referred to as "dead". In this situation, in order to perform new operations, the client must establish a new session. If an attempt is made to establish this new session with the existing client ID, the server will reject the request with NFS4ERR_STALE_CLIENTID.

When NFS4ERR_STALE_CLIENTID is received in either of these situations, the client must obtain a new client ID by use of the EXCHANGE_ID operation, then use that client ID as the basis of a new session, and then proceed to any other necessary recovery for the server restart case (See [Section 8.4.2](#)).

See the descriptions of EXCHANGE_ID ([Section 18.35](#)) and CREATE_SESSION ([Section 18.36](#)) for a complete specification of these operations.

[2.4.1](#). Upgrade from NFSv4.0 to NFSv4.1

To facilitate upgrade from NFSv4.0 to NFSv4.1, a server may compare a client_owner4 in an EXCHANGE_ID with an nfs_client_id4 established using the SETCLIENTID operation of NFSv4.0. A server that does so will allow an upgraded client to avoid waiting until the lease (i.e.

the lease established by the NFSv4.0 instance client) expires. 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 (per the recommendations of the NFSv4.0 specification [20]), and the NFSv4.1 client does not wish to use a client ID that prevents trunking, it should send 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.

[2.4.2](#). Server Release of Client ID

NFSv4.1 introduces a new operation called DESTROY_CLIENTID ([Section 18.50](#)) which the client SHOULD use to destroy a client ID it no longer needs. This permits graceful, bilateral release of a client ID. The operation cannot be used if there are sessions associated with the client ID, or state with an unexpired lease.

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 in order to conserve resources. 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 client ID. The server ought to 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 2.10.12.1.4](#) for discussion on releasing inactive sessions.

[2.4.3.](#) Resolving Client Owner Conflicts

When the server gets an EXCHANGE_ID for a client owner that currently has no state, or that has state, but the lease has expired, the server MUST allow the EXCHANGE_ID, and confirm the new client ID if followed by the appropriate CREATE_SESSION.

When the server gets an EXCHANGE_ID for a new incarnation of a client owner that currently has an old incarnation with state and an unexpired lease, the server is allowed to dispose of the state of the previous incarnation of the client owner if one of the following are

true:

- o The principal that created the client ID for the client owner is the same as the principal that is issuing the EXCHANGE_ID. Note that if the client ID was created with SP4_MACH_CRED state protection ([Section 18.35](#)), the principal MUST be based on RPCSEC_GSS authentication, the RPCSEC_GSS service used MUST be

integrity or privacy, and the same GSS mechanism and principal must be used as that used when the client ID was created.

- o The client ID was established with SP4_SSV protection ([Section 18.35](#), [Section 2.10.8.3](#)) and the client sends the EXCHANGE_ID with the security flavor set to RPCSEC_GSS using the GSS SSV mechanism ([Section 2.10.9](#)).
- o The client ID was established with SP4_SSV protection, and under the conditions described herein, the EXCHANGE_ID was sent with SP4_MACH_CRED state protection. Because the SSV might not persist across client and server restart, and because the first time a client sends EXCHANGE_ID to a server it does not have an SSV, the client MAY send the subsequent EXCHANGE_ID without an SSV RPCSEC_GSS handle. Instead, as with SP4_MACH_CRED protection, the principal MUST be based on RPCSEC_GSS authentication, the RPCSEC_GSS service used MUST be integrity or privacy, and the same GSS mechanism and principal MUST be used as that used when the client ID was created.

If none of the above situations apply, the server MUST return NFS4ERR_CLID_INUSE.

If the server accepts the principal and co_ownerid as matching that which created the client ID, and the co_verifier in the EXCHANGE_ID differs from the co_verifier used when the client ID was created, then after the server receives a CREATE_SESSION that confirms the client ID, the server deletes state. If the co_verifier values are the same, (e.g. the client is either updating properties of the client ID ([Section 18.35](#)), or the client is attempting trunking ([Section 2.10.5](#)) the server MUST NOT delete state.

[2.5](#). Server Owners

The Server Owner is similar to a Client Owner ([Section 2.4](#)), but unlike the Client Owner, there is no shorthand server ID. The Server Owner is defined in the following data type:

```

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

```

The Server Owner is returned from EXCHANGE_ID. When the so_major_id fields are the same in two EXCHANGE_ID results, the connections each EXCHANGE_ID were 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 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 2.10.5](#) and [Section 18.35](#) in order to avoid acting on falsely matching Server Owner values.

The considerations for generating a so_major_id are similar to that for generating a co_ownerid string (see [Section 2.4](#)). The consequences of two servers generating conflicting 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 2.10.5](#)).

[2.6](#). Security Service Negotiation

With the NFSv4.1 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 21](#) for further discussion.

[2.6.1](#). NFSv4.1 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.

Internet-Draft

NFSv4.1

September 2008

[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. However, the server's policies may also change at any time and force 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 sent 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 NFSv4.1 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 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.

[2.6.3.1.1.](#) Put Filehandle Operations

The term "put filehandle operation" refers to PUTROOTFH, PUTPUBFH, PUTFH, and RESTOREFH. Each of the subsections herein describes how the server handles a subseries of operations that starts with a put

filehandle operation.

[2.6.3.1.1.1.](#) Put Filehandle Operation + SAVEFH

The client is saving a filehandle for a future RESTOREFH, LINK, or RENAME. SAVEFH MUST NOT return NFS4ERR_WRONGSEC. To determine whether the put filehandle operation returns NFS4ERR_WRONGSEC or not, the server implementation pretends SAVEFH is not in the series of operations and examines which of the situations described in the other subsections of [Section 2.6.3.1.1](#) apply.

[2.6.3.1.1.2.](#) 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 N'th put filehandle operation is handled as if it is the first in a subseries of operations. 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 2.6.3.1.1.3](#).

[2.6.3.1.1.3.](#) Put Filehandle Operation + LOOKUP (or OPEN of an Existing Name)

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

In this situation, the client is potentially crossing a security policy boundary, and the set of security tuples the parent directory supports may 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) `sec_policy_child ^ sec_policy_parent != {}` (^ for intersection, {} for the empty set). 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) `sec_policy_child ^ sec_policy_parent == {}`. This means that the set of tuples specified on the security policy of a child directory may not intersect with that of the parent. In other words, there are no restrictions on how the system administrator

may set up these tuples.

In order for a server to support approaches (b) (for the case when a client chooses a flavor that is not a member of `sec_policy_parent`) and (c), the put filehandle operation cannot return `NFS4ERR_WRONGSEC` when there is a security tuple mismatch. Instead, it should be returned from the LOOKUP (or OPEN by existing 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.

2.6.3.1.1.4. 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 via style `SECINFO_STYLE4_PARENT`, which works in the opposite direction as `SECINFO`. As with [Section 2.6.3.1.1.3](#), a put

filehandle operation that is followed by a LOOKUPP MUST NOT return NFS4ERR_WRONGSEC. If the server does not support SECINFO_NO_NAME, the client's only recourse is to send the put filehandle operation, LOOKUPP, GETFH sequence of operations with every security tuple it supports.

Regardless of 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.

[2.6.3.1.1.5.](#) 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 REQUIRED set.

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

[2.6.3.1.1.6.](#) Put Filehandle Operation + Nothing

The NFSv4.1 server MUST NOT return NFS4ERR_WRONGSEC.

[2.6.3.1.1.7.](#) 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 the put filehandle operation must return NFS4ERR_WRONGSEC when there is a security tuple 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).

[2.6.3.1.1.8](#). Operations after SECINFO and SECINFO_NO_NAME

Suppose a client sends a COMPOUND procedure containing the series SEQUENCE, PUTFH, SECINFO_NONAME, READ, and suppose the security tuple used does not match that required for the target file. By rule (see [Section 2.6.3.1.1.5](#)), neither PUTFH nor SECINFO_NO_NAME can return NFS4ERR_WRONGSEC. By rule (see [Section 2.6.3.1.1.7](#)), READ cannot return NFS4ERR_WRONGSEC. The issue is resolved by the fact that SECINFO and SECINFO_NO_NAME consume the current filehandle (note that this is a change from NFSv4.0). This leaves no current filehandle for READ to use, and READ returns NFS4ERR_NOFILEHANDLE.

[2.6.3.1.2](#). LINK and RENAME

The LINK and RENAME operations use both the current and saved filehandles. When the current filehandle is injected into a series of operations via a put filehandle operation, the server MUST return NFS4ERR_WRONGSEC, per [Section 2.6.3.1.1](#). LINK and RENAME MAY return NFS4ERR_WRONGSEC if the security policy of the saved filehandle rejects the security flavor used in the COMPOUND request's credentials. If the server does so, then if there is no intersection between the security policies of saved and current filehandles, this means it will be impossible for client to perform the intended LINK or RENAME operation.

For example, suppose the client sends this COMPOUND request: SEQUENCE, PUTFH bFH, SAVEFH, PUTFH aFH, RENAME "c" "d", where

filehandles bFH and aFH refer to different directories. Suppose no common security tuple exists between the security policies of aFH and bFH. If the client sends the request using credentials acceptable to bFH's security policy but not aFH's policy, then the PUTFH aFH operation will fail with NFS4ERR_WRONGSEC. After a SECINFO_NO_NAME request, the client sends SEQUENCE, PUTFH bFH, SAVEFH, PUTFH aFH, RENAME "c" "d", using credentials acceptable to aFH's security policy, but not bFH's policy. The server returns NFS4ERR_WRONGSEC on the RENAME operation.

To prevent a client from an endless sequence of a request containing LINK or RENAME, followed by a request containing SECINFO_NO_NAME, the server MUST detect when the security policies of the current and saved filehandles have no mutually acceptable security tuple, and MUST NOT return NFS4ERR_WRONGSEC in that situation. Instead the server MUST return NFS4ERR_XDEV.

Thus while a server MAY return NFS4ERR_WRONGSEC from LINK and RENAME, the server implementor may reasonably decide the consequences are not worth the security benefits, and so allow the security policy of the current filehandle to override that of the saved filehandle.

[2.7.](#) Minor Versioning

To address the requirement of an NFS protocol that can evolve as the need arises, the NFSv4.1 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 one or more new RFCs. Minor version zero of the NFSv4 protocol is represented by [\[20\]](#), and minor version one

is represented by this document [\[\[Comment.1: RFC Editor: change "document" to "RFC" when we publish\]\]](#). The COMPOUND and CB_COMPOUND procedures support the encoding of the minor version being requested by the client.

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

definition.

1. Procedures are not added or deleted

To maintain the general RPC model, NFSv4 minor versions will not add to or delete procedures from the NFS program.

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

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

- * Minor versions may append attributes to the bitmap4 that represents sets of attributes and the fattr4 that represents sets of attribute values.

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 and providing corresponding variants of opaque arrays, such as a notify4 used together with such bitmaps.

- * adding bits to existing attributes like ACLs that have flag words
 - * extending enumerated types (including NFS4ERR_*) with new values
 - * adding cases to a switched union
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 MUST NOT be implemented.

Specifying an operation MUST NOT be implemented 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 an attribute MUST NOT be implemented.
 2. Minor versions may declare a flag bit or enumeration value MUST NOT be implemented.
9. Minor versions may downgrade features from REQUIRED to RECOMMENDED, or RECOMMENDED to OPTIONAL.
 10. Minor versions may upgrade features from OPTIONAL to RECOMMENDED or RECOMMENDED to REQUIRED.

11. A client and server that 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 REQUIRED in a minor version.

This rule allows for the introduction of new functionality and forces the use of implementation experience before designating a feature as REQUIRED. On the other hand, some classes of features are infrastructural and have broad effects. Allowing such features to not be REQUIRED 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 $X \neq Y$.

[2.8.](#) Non-RPC-based Security Services

As described in [Section 2.2.1.1.1.1](#), NFSv4.1 relies on RPC for identification, authentication, integrity, and privacy. NFSv4.1 itself provides or enables additional security services as described in the next several subsections.

[2.8.1.](#) Authorization

Authorization to access a file object via an NFSv4.1 operation is ultimately determined by the NFSv4.1 server. A client can predetermine its access to a file object via the OPEN ([Section 18.16](#)) and the ACCESS ([Section 18.1](#)) operations.

Principals with appropriate access rights can modify the authorization on a file object via the SETATTR ([Section 18.30](#)) operation. Attributes that affect access rights include: mode, owner, owner_group, acl, dacl, and sacl. See [Section 5](#).

[2.8.2.](#) Auditing

NFSv4.1 provides auditing on a per file object basis, via the acl and sacl attributes 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

NFSv4.1 provides alarm control on a per file object basis, via the acl and sacl attributes 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

NFSv4.1 works over RDMA and non-RDMA-based transports with the following attributes:

- o The transport supports reliable delivery of data, which NFSv4.1 requires but neither NFSv4.1 nor RPC has facilities for ensuring. [\[23\]](#)
- o 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 [\[3\]](#).

Where an NFSv4.1 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 NFSv4.1 implementation MUST support operation over the TCP transport protocol.

Even if NFSv4.1 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 combined with congestion control by the connectionless transport is 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.

Shepler, et al.

Expires March 9, 2009

[Page 39]

Internet-Draft

NFSv4.1

September 2008

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 backchannel (see [Section 2.10.3.1](#)) 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,

- o A requester MUST NOT retry a request unless the connection the request was sent over was lost before the reply was received.
- o A replier MUST NOT silently drop a request, even if the request is a retry. (The silent drop behavior of RPCSEC_GSS [4] 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 2.10.6.1](#)) or it MAY disconnect the connection.

When sending a reply, the replier MUST send the reply to the same full network address (e.g. if using an IP-based transport, the source port of the requester is part of the full network address) that the requester sent the request from. If using a connection-oriented transport, replies MUST be sent on the same connection the request was received from.

If a connection is dropped after the replier receives the request but before the replier sends the reply, the replier might have an pending reply. If a connection is established with the same source and destination full network address as the dropped connection, then the replier MUST NOT send the reply until the client retries the request.

The reason for this prohibition is that the client MAY retry a request over a different connection than is associated with the session.

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

- o 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.

- o 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 sent from it, and credit information appropriate to the channel must be refreshed by the RPC layer.

In addition, as described in [Section 2.10.6.2](#), while a session is active, the NFSv4.1 requester MUST NOT stop waiting for a reply.

[2.9.3.](#) Ports

Historically, NFSv3 servers have listened over TCP port 2049. The registered port 2049 [\[24\]](#) for the NFS protocol should be the default configuration. NFSv4.1 clients SHOULD NOT use the RPC binding protocols as described in [\[25\]](#).

[2.10.](#) Session

[2.10.1.](#) Motivation and Overview

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

- o Lack of support for Exactly Once Semantics (EOS). This includes lack of support for EOS through server failure and recovery.
- o Limited callback support, including no support for sending callbacks through firewalls, and races between replies to normal requests and callbacks.
- o Limited trunking over multiple network paths.
- o Requiring machine credentials for fully secure operation.

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

- o EOS is enabled by a reply cache with a bounded size, making it feasible to keep the cache in 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 2.10.6](#), NFSv4.1 supports both EOS and unlimited

parallelism.

- o The NFSv4.1 client (defined in [Section 1.5](#), Paragraph 2) creates transport connections and provides them to the server to use for sending callback requests, thus solving the firewall issue ([Section 18.34](#)). Races between responses from client requests, and callbacks caused by the requests are detected via the session's sequencing properties which are a consequence of EOS ([Section 2.10.6.3](#)).
- o The NFSv4.1 client can add an arbitrary number of connections to the session, and thus provide trunking ([Section 2.10.5](#)).
- o The NFSv4.1 client and server produces a session key independent of client and server machine credentials which can be used to compute a digest for protecting critical session management operations ([Section 2.10.8.3](#)).

- o The NFSv4.1 client can also create secure RPCSEC_GSS contexts for use by the session's backchannel that do not require the server to authenticate to a client machine principal ([Section 2.10.8.2](#)).

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. A client may have one or more sessions associated with it so that client-associated state may be accessed using any of the sessions associated with that client's client ID, when connections are associated with those sessions. When no connections are associated with any of a client ID's sessions for an extended time, such objects as locks, opens, delegations, layouts, etc. are subject to expiration. The session serves as an object representing a means of access by a client to the associated client state on the server, independent of the physical means of access to that state.

A single client may create multiple sessions. A single session **MUST NOT** serve multiple clients.

[2.10.2](#). NFSv4 Integration

Sessions are part of NFSv4.1 and not NFSv4.0. Normally, a major infrastructure change such as sessions would require a new major version number to an ONC 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 have been added to NFSv4.1 with little difficulty. 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. NFSv4.1 defines the SEQUENCE operation, which is required for every COMPOUND that operates over an established session, with the exception of some session administration operations, such as DESTROY_SESSION ([Section 18.37](#)).

[2.10.2.1](#). SEQUENCE and CB_SEQUENCE

In NFSv4.1, when the SEQUENCE operation is present, it MUST be 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 2.10.6](#)). Session-enabled NFSv4.1 COMPOUND requests thus have the form:

```
+-----+-----+-----+-----+-----+-----+
| tag | minorversion | numops   | SEQUENCE op | op + args | ...
|      | (= 1)       | (limited) | + args      |           |
+-----+-----+-----+-----+-----+-----+
```

and the replies have the form:

```
+-----+-----+-----+-----+-----+-----+---//
|last status | tag | numres |status + SEQUENCE op + results | //
+-----+-----+-----+-----+-----+-----+---//
//-----+-----+
// status + op + results | ...
//-----+-----+-----+
```

A CB_COMPOUND procedure request and reply has a similar form to COMPOUND, but instead of a SEQUENCE operation, there is a CB_SEQUENCE operation. CB_COMPOUND also has an additional field called "callback_ident", which is superfluous in NFSv4.1 and MUST be ignored by the client. CB_SEQUENCE has the same information as SEQUENCE, and also includes other information needed to resolve callback races ([Section 2.10.6.3](#)).

[2.10.2.2](#). Client ID and Session Association

Each client ID ([Section 2.4](#)) can have zero or more active sessions. A client ID and associated session are required to perform file access in NFSv4.1. Each time a session is used (whether by a client sending a request to the server, or the client replying to a callback request from the server), the state leased to its associated client

ID is automatically renewed.

State such as share reservations, locks, delegations, and layouts ([Section 1.6.4](#)) is tied to the client ID. Client state is not tied

to any individual session. Successive state changing operations from a given state owner MAY go over different sessions, provided the session is associated with the same client ID. A callback MAY arrive over a different session than from the session that originally acquired the state pertaining to the callback. For example, if session A is used to acquire a delegation, a request to recall the delegation MAY arrive over session B if both sessions are associated with the same client ID. [Section 2.10.8.1](#) and [Section 2.10.8.2](#) discuss the security considerations around callbacks.

[2.10.3.](#) Channels

A channel is not a connection. A channel represents the direction ONC RPC requests are sent.

Each session has one or two channels: the fore channel and the backchannel. Because there are at most two channels per session, and because each channel has a distinct purpose, channels are not assigned identifiers.

The fore channel is used for ordinary requests from the client to the server, and carries COMPOUND requests and responses. A session always has a fore channel.

The backchannel used for callback requests from server to client, and carries CB_COMPOUND requests and responses. Whether there is a backchannel or not is a decision by the client, however many features of NFSv4.1 require a backchannel. NFSv4.1 servers MUST support backchannels.

Each session has resources for each channel, including separate reply caches (see [Section 2.10.6.1](#)). Note that even the backchannel requires a reply cache because some callback operations are nonidempotent.

[2.10.3.1.](#) Association of Connections, Channels, and Sessions

Each channel is associated with zero or more transport connections (whether of the same transport protocol or different transport protocols). A connection can be associated with 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 18.36](#)) and the BIND_CONN_TO_SESSION ([Section 18.34](#)) operations. When a session is created via

CREATE_SESSION, the connection that transported the CREATE_SESSION request is automatically associated with the fore channel, and optionally the backchannel. If the client specifies no state protection ([Section 18.35](#)) when the session is created, then when SEQUENCE is transmitted on a different connection, the connection is automatically associated with the fore channel of the session specified in the SEQUENCE operation.

A connection's association with a session is not exclusive. A connection associated with the channel(s) of one session may be simultaneously associated with the channel(s) of other sessions including sessions associated with other client IDs.

It is permissible for connections of multiple transport types to be associated with the same channel. For example both a TCP and RDMA connection can be associated with the fore channel. In the event an RDMA and non-RDMA connection are associated with the same channel, the maximum number of slots SHOULD be at least one more than the total number of RDMA credits ([Section 2.10.6.1](#)). This way if all RDMA credits are used, the non-RDMA connection can have at least one outstanding request. If a server supports multiple transport types, it MUST allow a client to associate connections from each transport to a channel.

It is permissible for a connection of one type of transport to be associated with the fore channel, and a connection of a different type to be associated with the backchannel.

[2.10.4](#). Server Scope

Servers each specify a server scope value in the form of an opaque string `eir_server_scope` returned as part of the results of an EXCHANGE_ID operation. The purpose of the server scope is to allow a group of servers to indicate to clients that a set of servers sharing the same server scope value have arranged to use compatible values of otherwise opaque identifiers. Thus the identifiers generated by one server of that set may be presented to another of that same scope.

The use of such compatible values does not imply that a value generated by one server will always be accepted by another. In most cases, it will not. However, a server will not accept a value generated by another inadvertently. When it does accept it, it will be because it is recognized as valid and carrying the same meaning as on another server of the same scope.

When servers are of the same server scope, this compatibility of values applies to the follow identifiers:

- o Filehandle values. A filehandle value accepted by two servers of the same server scope denotes the same object. A write done to one server is reflected immediately in a read done to the other and locks obtained on one server conflict with those requested on the other.
- o Session ID values. A session ID value accepted by two servers of the same server scope denotes the same session.
- o Client ID values. A client ID value accepted as valid by two servers of the same server scope is associated with two clients with the same client owner and verifier.
- o State ID values when the corresponding client ID is recognized as valid. If the same stateid value is accepted as valid on two servers of the same scope and the client IDs on the two servers represent the same client owner and verifier, then the two stateid values designate the same set of locks and are for the same file
- o Server owner values. When the server scope values are the same, server owner value may be validly compared. In cases where the server scope are different, server owner values are treated as different even if they contain all identical bytes.

The co-ordination among servers required to provide such compatibility can be quite minimal, and limited to a simple partition of the ID space. The recognition of common values requires additional implementation, but this can be tailored to the specific situations in which that recognition is desired.

Clients will have occasion to compare the server scope values of multiple servers under a number of circumstances, each of which will be discussed under the appropriate functional section.

- o When server owner values received in response to EXCHANGE_ID operations issued to multiple network addresses are compared for the purpose of determining the validity of various forms of trunking, as described in [Section 2.10.5](#).
- o When network or server reconfiguration causes the same network

address to possibly be directed to different servers, with the necessity for the client to determine when lock reclaim should be attempted, as described in [Section 8.4.2.1](#)

- o When file system migration causes the transfer of responsibility for a file system between servers and the client needs to determine whether state has been transferred with the file system (as described in [Section 11.7.7](#)) or whether the client needs to

reclaim state on a similar basis as in the case of server restart, as described in [Section 8.4.2](#).

When two replies from EXCHANGE_ID each from two different server network addresses have the same server scope, there are a number of ways a client can validate that the common server scope is due to two servers cooperating in a group.

- o If both EXCHANGE_ID requests were sent with RPCSEC_GSS authentication and the server principal is the same for both targets, the equality of server scope is validated. It is RECOMMENDED that two servers intending to share the same server scope also share the same principal name.
- o The client may accept the appearance of the second server in fs_locations or fs_locations_info attribute for a relevant file system. For example, if there is a migration event for a particular file system or there are locks to be reclaimed on a particular file system, the attributes for that particular file system may be used. The client sends the GETATTR request to the first server for the fs_locations or fs_locations_info attribute with RPCSEC_GSS authentication. It may need to do this in advance of the need to verify the common server scope. If the client successfully authenticates the reply to GETATTR, and the GETATTR request and reply containing the fs_locations or fs_locations_info attribute refers to the second server, then the equality of server scope is supported. A client may choose to limit the use of this form of support to information relevant to the specific file system involved (e.g. a file system being migrated).

[2.10.5](#). Trunking

Trunking is the use of multiple connections between a client and

server in order to increase the speed of data transfer. NFSv4.1 supports two types of trunking: session trunking and client ID trunking.

NFSv4.1 servers MUST support both forms of trunking within the context of a single server network address and MUST support both forms within the context of the set of network addresses used to access a single server. NFSv4.1 servers in a clustered configuration MAY allow network addresses for different servers to use client ID trunking.

Clients may use either form of trunking as long as they do not, when trunking between different server network addresses, violate the servers' mandates as to the kinds of trunking to be allowed (see below). With regard to callback channels, the client MUST allow the

server to choose among all callback channels valid for a given client ID and MUST support trunking when the connections supporting the backchannel allow session or client ID trunking to be used for callbacks

Session trunking is essentially the association of multiple connections, each with potentially different target and/or source network addresses, to the same session. When the target network addresses (server addresses) of the two connections are the same, the server MUST support such session trunking. When the target network addresses are different, the server MAY indicate such support using the data returned by the EXCHANGE_ID operation (see below).

Client ID trunking is the association of multiple sessions to the same client ID. Servers MUST support client ID trunking for two target network addresses whenever they allow session trunking for those same two network addresses. In addition, a server MAY, by presenting the same major server owner ID ([Section 2.5](#)), and server scope ([Section 2.10.4](#)) allow an additional case of client ID trunking. When two servers return the same major server owner and server scope, it means that the two servers are cooperating on locking state management which is a prerequisite for client ID trunking.

Understanding and distinguishing when the client is allowed to use session and client ID trunking requires understanding how the results

of the EXCHANGE_ID ([Section 18.35](#)) operation identify a server. Suppose a client sends EXCHANGE_ID over two different connections each with a possibly different target network address but each EXCHANGE_ID operation has the same value in the eia_clientowner field. If the same NFSv4.1 server is listening over each connection, then each EXCHANGE_ID result MUST return the same values of eir_clientid, eir_server_owner.so_major_id and eir_server_scope. The client can then treat each connection as referring to the same server (subject to verification, see Paragraph 8 later in this section), and it can use each connection to trunk requests and replies. The client's choice is whether session trunking or client ID trunking applies.

Session Trunking. If the eia_clientowner argument is the same in two different EXCHANGE_ID requests, and the eir_clientid, eir_server_owner.so_major_id, eir_server_owner.so_minor_id, and eir_server_scope results match in both EXCHANGE_ID results, then the client is permitted to perform session trunking. If the client has no session mapping to the tuple of eir_clientid, eir_server_owner.so_major_id, eir_server_scope, eir_server_owner.so_minor_id, then it creates the session via a CREATE_SESSION operation over one of the connections, which

associates the connection to the session. If there is a session for the tuple, the client can send BIND_CONN_TO_SESSION to associate the connection to the session.

Of course, if the client does not desire to use session trunking, it is not required to do so. It can invoke CREATE_SESSION on the connection. This will result in client ID trunking as described below. It can also decide to drop the connection if it does not choose to use trunking.

Client ID Trunking. If the eia_clientowner argument is the same in two different EXCHANGE_ID requests, and the eir_clientid, eir_server_owner.so_major_id, and eir_server_scope results match in both EXCHANGE_ID results, then the client is permitted to perform client ID trunking (regardless whether the eir_server_owner.so_minor_id results match). The client can associate each connection with different sessions, where each session is associated with the same server.

The client completes the act of client ID trunking by invoking `CREATE_SESSION` on each connection, using the same client ID that was returned in `eir_clientid`. These invocations create two sessions and also associate each connection with its respective session. The client is free to choose not to use client ID trunking by simply dropping the connection at this point.

When doing client ID trunking, locking state is shared across sessions associated with that same client ID. This requires the server to coordinate state across sessions.

The client should be prepared for the possibility that `eir_server_owner` values may be different on subsequent `EXCHANGE_ID` requests made to the same network address, as a result of various sorts of reconfiguration events. When this happens and the changes result in the invalidation of previously valid forms of trunking, the client should cease to use those forms, either by dropping connections or by adding sessions. For a discussion of lock reclaim as it relates to such reconfiguration events, see [Section 8.4.2.1](#).

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:

- o For session trunking, clients SHOULD reliably verify if connections between different network paths are in fact associated

with the same NFSv4.1 server and usable on the same session, and servers MUST allow clients to perform reliable verification. When a client ID is created, the client SHOULD specify that `BIND_CONN_TO_SESSION` is to be verified according to the `SP4_SSV` or `SP4_MACH_CRED` ([Section 18.35](#)) state protection options. For `SP4_SSV`, reliable verification depends on a shared secret (the SSV) that is established via the `SET_SSV` ([Section 18.47](#)) operation.

When a new connection is associated with the session (via the `BIND_CONN_TO_SESSION` operation, see [Section 18.34](#)), if the client specified `SP4_SSV` state protection for the `BIND_CONN_TO_SESSION`

operation, the client MUST send the BIND_CONN_TO_SESSION with RPCSEC_GSS protection, using integrity or privacy, and an RPCSEC_GSS handle created with the GSS SSV mechanism ([Section 2.10.9](#)).

If the client mistakenly tries to associate 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 RPCSEC_GSS authentication fails. Even if the server mistakenly or maliciously accepts the connection association attempt, the RPCSEC_GSS verifier it computes in the response will not be verified by the client, so the client will know it cannot use the connection for trunking the specified session.

If the client specified SP4_MACH_CRED state protection, the BIND_CONN_TO_SESSION operation will use RPCSEC_GSS integrity or privacy, using the same credential that was used when the client ID was created. Mutual authentication via RPCSEC_GSS assures the client that the connection is associated with the correct session of the correct server.

- o For client ID trunking, the client has at least two options for verifying that the same client ID obtained from two different EXCHANGE_ID operations came from the same server. The first option is to use RPCSEC_GSS authentication when issuing each EXCHANGE_ID. Each time an EXCHANGE_ID is sent with RPCSEC_GSS authentication, the client notes the principal name of the GSS target. If the EXCHANGE_ID results indicate client ID trunking is possible, and the GSS targets' principal names are the same, the servers are the same and client ID trunking is allowed.

The second option for verification is to use SP4_SSV protection. When the client sends EXCHANGE_ID it specifies SP4_SSV protection. The first EXCHANGE_ID the client sends always has to be confirmed

by a CREATE_SESSION call. The client then sends SET_SSV. Later the client sends EXCHANGE_ID to a second destination network address different from the one the first EXCHANGE_ID was sent to. The client checks that each EXCHANGE_ID reply has the same eir_clientid, eir_server_owner.so_major_id, and eir_server_scope.

If so, the client verifies the claim by issuing a `CREATE_SESSION` to the second destination address, protected with `RPCSEC_GSS` integrity using an `RPCSEC_GSS` handle returned by the second `EXCHANGE_ID`. If the server accepts the `CREATE_SESSION` request, and if the client verifies the `RPCSEC_GSS` verifier and integrity codes, then the client has proof the second server knows the SSV, and thus the two servers are co-operating for the purposes of specifying server scope and client ID trunking.

[2.10.6.](#) 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 fore and back channels.

Each `COMPOUND` or `CB_COMPOUND` request that is sent with a leading `SEQUENCE` or `CB_SEQUENCE` operation **MUST** be executed by the receiver exactly once. This requirement holds regardless of whether the request is sent with reply caching specified (see [Section 2.10.6.1.3](#)). The requirement holds even if the requester is issuing the request over a session created between a pNFS data client and pNFS data server. To understand the rationale for this requirement, divide the requests into three classifications:

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

An example of a non-idempotent request is `RENAME`. It 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 a single time. Nevertheless, enforcing EOS for `WRITES` and other idempotent modifying requests is necessary to avoid data corruption.

Suppose a client sends `WRITE A` 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-sends WRITE A. Now, the server has outstanding two instances of A. The server can be in a situation in which it executes and replies to the retry of A, while the first A is still waiting in the server's internal I/O system for some resource. Upon receiving the reply to the second attempt of WRITE A, the client believes its write is done so it is free to send WRITE B which overlaps the range of A. When the original A is dispatched from the server's I/O system, and executed (thus the second time A will have been written), then what has been written by B can be overwritten and thus corrupted.

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, to keep the implementation simple, the replier MUST enforce EOS for all requests whether idempotent and non-modifying or not.

Note that true and complete EOS is not possible unless the server persists the reply cache in stable storage, unless the server is somehow implemented to never require a restart (indeed if such a server exists, the distinction between a reply cache kept in stable storage versus one that is not is one without meaning). See [Section 2.10.6.5](#) for a discussion of persistence in the reply cache. Regardless, even if the server does not persist the reply cache, EOS improves robustness and correctness over previous versions of NFS because the legacy duplicate request/reply caches were based on the ONC RPC transaction identifier (XID). [Section 2.10.6.1](#) explains the shortcomings of the XID as a basis for a reply cache and describes how NFSv4.1 sessions improve upon the XID.

[2.10.6.1](#). Slot Identifiers and Reply Cache

The RPC layer provides a transaction ID (XID), which, while required to be unique, is not convenient for tracking requests for two reasons. First, the XID is only meaningful to the requester; it cannot be interpreted by the replier except to test for equality with previously sent requests. When consulting an RPC-based duplicate request cache, the opaqueness of the XID requires a computationally expensive lookup (often via a hash that includes XID and source address). NFSv4.1 requests use a non-opaque slot ID which is an index into a slot table, which is far more efficient. Second, because RPC requests can be executed by the replier in any order, there is no bound on the number of requests that may be outstanding at any time. To achieve perfect EOS using ONC RPC would require storing all replies in the reply cache. XIDs are 32 bits; storing over four billion (2^{32}) replies in the reply cache is not practical.

Internet-Draft

NFSv4.1

September 2008

In practice, previous versions of NFS have chosen to store a fixed number of replies in the cache, and use a least recently used (LRU) approach to replacing cache entries with new entries when the cache is full. In NFSv4.1, the number of outstanding requests is bounded by the size of the slot table, and a sequence ID per slot is used to tell the replier when it is safe to delete a cached reply.

In the NFSv4.1 reply cache, when the requester sends a new request, it selects a slot ID in the range 0..N, where N is the replier's current maximum slot ID granted to the requester on the session over which the request is to be sent. The value of N starts out as equal to `ca_maxrequests - 1` ([Section 18.36](#)), but can be adjusted by the response to SEQUENCE or CB_SEQUENCE as described later in this section. The slot ID 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 slot ID.

A slot contains a sequence ID and the cached reply corresponding to the request sent with that sequence ID. The sequence ID is a 32 bit unsigned value, and is therefore in the range 0..0xFFFFFFFF ($2^{32} - 1$). The first time a slot is used, the requester MUST specify a sequence ID of one (1) ([Section 18.36](#)). Each time a slot is reused, the request MUST specify a sequence ID that is one greater than that of the previous request on the slot. If the previous sequence ID was 0xFFFFFFFF, then the next request for the slot MUST have the sequence ID set to zero (i.e. $(2^{32} - 1) + 1 \bmod 2^{32}$).

The sequence ID accompanies the slot ID in each request. It is for the critical check at the server: it used to efficiently determine 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 whether 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 replier compares each received request's sequence ID with the last one previously received for that slot ID, to see if the new request is:

- o A new request, in which the sequence ID is one greater than that previously seen in the slot (accounting for sequence wraparound).

The replier proceeds to execute the new request, and the replier MUST increase the slot's sequence ID by one.

- o A retransmitted request, in which the sequence ID is equal to that currently recorded in the slot. If the original request has executed to completion, the replier returns the cached reply. See

[Section 2.10.6.2](#) for direction on how the replier deals with retries of requests that are still in progress.

- o A misordered retry, in which the sequence ID is less than (accounting for sequence wraparound) that previously seen in the slot. The replier MUST return NFS4ERR_SEQ_MISORDERED (as the result from SEQUENCE or CB_SEQUENCE).
- o A misordered new request, in which the sequence ID is two or more than (accounting for sequence wraparound) than that previously seen in the slot. Note that because the sequence ID must wraparound to zero (0) once it reaches 0xFFFFFFFF, a misordered new request and a misordered retry 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 that unlike XID-indexed reply caches (also known as duplicate request caches - DRCs), the slot ID-based reply cache cannot be overflowed. Through use of the sequence ID 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 facilities make it practical to maintain all the required entries for an effective reply cache.

The slot ID, sequence ID, and session ID therefore take over the traditional role of the XID and source network address in the replier's reply cache implementation. This approach is considerably more portable and completely robust - it is not subject to the 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

requester. While the replier does not care about the XID for the purposes of reply cache management (but the replier MUST return the same XID that was in the request), nonetheless there are considerations for the XID in NFSv4.1 that are the same as all other previous versions of NFS. The RPC XID remains in each message and must be formulated in NFSv4.1 requests as in any other ONC RPC request. The reasons include:

- o The RPC layer retains its existing semantics and implementation.
- o The requester and replier must be able to interoperate at the RPC layer, prior to the NFSv4.1 decoding of the SEQUENCE or CB_SEQUENCE operation.

- o If an operation is being used that does not start with SEQUENCE or CB_SEQUENCE (e.g. BIND_CONN_TO_SESSION), then the RPC XID is needed for correct operation to match the reply to the request.
- o The SEQUENCE or CB_SEQUENCE operation may generate an error. If so, the embedded slot ID, sequence ID, and session ID (if present) in the request will not be in the reply, and the requester has only the XID to match the reply to the request.

Given that well formulated XIDs continue to be required, this begs the question why SEQUENCE and CB_SEQUENCE replies have a session ID, slot ID and sequence ID? Having the session ID in the reply means the requester does not have to use the XID to lookup the session ID, which would be necessary if the connection were associated with multiple sessions. Having the slot ID and sequence ID in the reply means requester does not have to use the XID to lookup the slot ID and sequence ID. Furthermore, since the XID is only 32 bits, it is too small to guarantee the re-association of a reply with its request ([26]); having session ID, slot ID, and sequence ID in the reply allows the client to validate that the reply in fact belongs to the matched request.

The SEQUENCE (and CB_SEQUENCE) operation also carries a "highest_slotid" value which carries additional requester slot usage information. The requester must always indicate the slot ID 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.

- o The enforced highest_slotid indicates the highest slot ID the requester is permitted to use on a subsequent SEQUENCE or CB_SEQUENCE operation. The replier's enforced highest_slotid SHOULD be no less than the highest_slotid the requester indicated in the SEQUENCE or CB_SEQUENCE arguments.

If a replier detects the client is being intransigent, i.e. it fails in a series of requests to honor the target highest_slotid even though the replier knows there are no outstanding requests a higher slot ids, it MAY take more forceful action. When faced with intransigence, the replier MAY reply with a new enforced highest_slotid that is less than its previous enforced highest_slotid. Thereafter, if the requester continues to send requests with a highest_slotid that is greater than the replier's new enforced highest_slotid the server MAY return NFS4ERR_BAD_HIGHSLOT, unless the slot ID in the request is greater than the new enforced highest_slotid, and the request is a retry.

The replier SHOULD retain the slots it wants to retire until the requester sends a request with a highest_slotid less than or equal to the replier's new enforced highest_slotid. Also if a request

is received with a slot that is higher than the new enforced `highest_slotid`, and the sequence ID is one higher than what is in the slot's reply cache, then the server can both retire the slot and return `NFS4ERR_BADSLOT` (however the server MUST NOT do one and not the other). (The reason it is safe to retire the slot is because that by using the next sequence ID, the client is indicating it has received the previous reply for the slot.) Once the replier has forcibly lowered the enforced `highest_slotid`, the requester is only allowed to send retries to the to-be-retired slots.

- o The requester SHOULD 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 `highest_slotid`, it will not be able to use only the receipt of the slot ID and `highest_slotid` in the request. Neither the slot ID 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 it can infer that the requester has seen a reply containing the new granted `highest_slotid`. The replier can infer that requester as seen such a reply when it receives a new request with the same slot ID as the request replied to and the

next higher sequence ID.

[2.10.6.1.1](#). Caching of SEQUENCE and CB_SEQUENCE Replies

When a SEQUENCE or CB_SEQUENCE operation is successfully executed, its reply MUST always be cached. Specifically, session ID, sequence ID, and slot ID MUST be cached in the reply cache. The reply from SEQUENCE also includes the highest slot ID, target highest slot ID, and status flags. Instead of caching these values, the server MAY re-compute the values from the current state of the fore channel, session and/or client ID as appropriate. Similarly, the reply from CB_SEQUENCE includes a highest slot ID and target highest slot ID. The client MAY re-compute the values from the current state of the session as appropriate.

Regardless of whether a replier is re-computing highest slot ID, target slot ID, and status on replies to retries or not, the requester MUST NOT assume the values are being re-computed whenever it receives a reply after a retry is sent, since it has no way of knowing whether the reply it has received was sent by the server in response to the retry, or is a delayed response to the original request. Therefore, it may be the case that highest slot ID, target slot ID, or status bits may reflect the state of affairs when the request was first executed. Although acting based on such delayed information is valid, it may cause the receiver to do unneeded work. Requesters MAY choose to send additional requests to get the current state of affairs or use the state of affairs reported by subsequent requests, in preference to acting immediately on data which may be out of date.

[2.10.6.1.2.](#) Errors from SEQUENCE and CB_SEQUENCE

Any time SEQUENCE or CB_SEQUENCE return an error, the sequence ID 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.

[2.10.6.1.3.](#) 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 [23]. Caching the reply may offer little benefit. If the reply is too large (see [Section 2.10.6.4](#)), it may not be cacheable anyway. Even if the reply to idempotent request is small enough to cache, unnecessarily caching

the reply slows down the server and increases RPC latency.

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 sequence ID 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, and with the next operation in `COMPOUND` or `CB_COMPOUND` having the error `NFS4ERR_RETRY_UNCACHED_REP`. Thus if the requester later retries the request, it will get `NFS4ERR_RETRY_UNCACHED_REP`.

2.10.6.2. Retry and Replay of Reply

A requester **MUST NOT** retry a request, unless the connection it used to send the request disconnects. The requester can then reconnect and re-send the request, or it can re-send the request over a different connection that is associated with the same session.

If the requester is a server wanting to re-send a callback operation over the backchannel of session, the requester of course cannot reconnect because only the client can associate connections with the backchannel. The server can re-send the request over another connection that is bound to the same session's backchannel. If there is no such connection, the server **MUST** indicate that the session has no backchannel by setting the `SEQ4_STATUS_CB_PATH_DOWN_SESSION` flag bit in the response to the next `SEQUENCE` operation from the client. The client **MUST** then associate a connection with the session (or destroy the session).

Note that it is not fatal for a client to retry without a disconnect between the request and retry. However the retry does consume resources, especially with RDMA, where each request, retry or not, consumes a credit. Retries for no reason, especially retries sent shortly after the previous attempt, are a poor use of network bandwidth and defeat the purpose of a transport's inherent congestion control system.

A requester **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 requester does not know what sequence ID to use for the slot on its

next request. For example, suppose a requester sends a request with sequence ID 1, and does not wait for the response. The next time it uses the slot, it sends the new request with sequence ID 2. If the replier has not seen the request with sequence ID 1, then the replier is not expecting sequence ID 2, and rejects the requester'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 RPC/RDMA "chunk" ([8]) are valid on a scope 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.

A retry might be sent while the original request is still in progress on the replier. The replier SHOULD deal with the issue by returning NFS4ERR_DELAY as the reply to SEQUENCE or CB_SEQUENCE operation, but implementations MAY return NFS4ERR_MISORDERED. Since errors from SEQUENCE and CB_SEQUENCE are never recorded in the reply cache, this approach allows the results of the execution of the original request to be properly recorded in the reply cache (assuming the requester specified the reply to be cached).

2.10.6.3. Resolving Server Callback Races

It is possible for server callbacks to arrive at the client before the reply from related fore 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 backchannel, which may be on a different transport connection, perhaps even a different network, or even a different session associated with the same client ID

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 { session ID, slot ID, sequence ID } triple. 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 race condition.

For each client operation which might result in some sort of server callback, the server SHOULD "remember" the { session ID, slot ID, sequence ID } triple of the client request until the slot ID

Internet-Draft

NFSv4.1

September 2008

retirement rules allow the server to determine that the client has, in fact, seen the server's reply. Until the time the { session ID, slot ID, sequence ID } request triple 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" { session ID, slot ID, sequence ID } triples. If the client finds the request corresponding to the referring session ID, slot ID and sequence ID to 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. If the client does not find the request corresponding the referring triple to be outstanding (including the case of a session ID referring to a destroyed session), then there is no race with respect to this triple. The server SHOULD limit the referring triples to requests that refer to just those that apply to the objects referred to in the CB_COMPOUND procedure.

The client must not simply wait forever for the expected server reply to arrive before responding to the CB_COMPOUND that won the race, because it is possible that it will be delayed indefinitely. The client should assume the likely case that the reply will arrive within the average round trip time for COMPOUND requests to the server, and wait that period of time. If that period of time expires it can respond to the CB_COMPOUND with NFS4ERR_DELAY.

There are other scenarios under which callbacks may race replies. Among them are pNFS layout recalls as described in [Section 12.5.5.2](#).

[2.10.6.4](#). 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 18.36](#)), for each channel (fore and back), 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 (which means no operations in the request executed, and the

state of the slot in the reply cache is unchanged), or it MAY opt to return it on a subsequent operation in the same `COMPOUND` or `CB_COMPOUND` request (which means at least one operation did execute and the state of the slot in reply cache does change). The replier SHOULD set `NFS4ERR_REQ_TOO_BIG` on the operation that exceeds `ca_maxrequestsize`.

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 opt to return it on a subsequent operation (in the same `COMPOUND` or `CB_COMPOUND` reply). A replier MAY return `NFS4ERR_REP_TOO_BIG` in the reply to `SEQUENCE` or `CB_SEQUENCE`, even if the response would still exceed `ca_maxresponsesize`.

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 2.10.6.1.2](#)). 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, the 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`, `PUTROOTFH` and

RESTOREFH) 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 to obtain the desired filehandle. For the OPEN operation (see [Section 18.16](#)), retry is not always available as an option. The following guidelines for the handling of filehandle changing operations are advised:

- o Within the same COMPOUND procedure, a client SHOULD send GETFH immediately after a current filehandle changing operation. A client MUST send GETFH after a current filehandle changing operation that is also non-idempotent (e.g., the OPEN operation), unless the operation is RESTOREFH. RESTOREFH is an exception,

because even though it is non-idempotent, the filehandle RESTOREFH produced originated from an operation that is either idempotent (e.g. PUTFH, LOOKUP), or non-idempotent (e.g. OPEN, CREATE). If the origin is non-idempotent, then because the client MUST send GETFH after the origin operation, the client can recover if RESTOREFH returns an error.

- o 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.
- o 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.
- o A server MAY return NFS4ERR_UNSAFE_COMPOUND to a non-idempotent current filehandle changing operation, if it looks at the next operation (in the same COMPOUND procedure) and finds it is not GETFH. The server SHOULD do this if it is unable to determine in advance whether the total response size would exceed ca_maxresponsesize_cached or ca_maxresponsesize.

[2.10.6.5](#). Persistence

Since the reply cache is bounded, it is practical for the reply cache to persist across server restarts. The replier MUST persist the

following information if it agreed to persist the session (when the session was created; see [Section 18.36](#)):

- o The session ID.
- o The slot table including the sequence ID and cached reply for each slot.

The above are sufficient for a replier to provide EOS semantics for any requests that were sent and executed before the server restarted. If the replier is a client then there is no need for it to persist any more information, unless the client will be persisting all other state across client restart. In which case, the server will never see any NFSv4.1-level protocol manifestation of a client restart. If the replier is a server, with just the slot table and session ID persisting, any requests the client retries after the server restart will return the results that are cached in reply cache. and any new requests (i.e. the sequence ID is one (1) greater than the slot's sequence ID) MUST be rejected with NFS4ERR_DEADSESSION (returned by

SEQUENCE). Such a session is considered dead. A server MAY re-animate a session after a server restart so that the session will accept new requests as well as retries. To re-animate a session the server needs to persist additional information through server restart:

- o The client ID. This is a prerequisite to let the client to create more sessions associated with the same client ID as the
- o The client ID's sequence ID that is used for creating sessions (see [Section 18.35](#) and [Section 18.36](#)). This is a prerequisite to let the client create more sessions.
- o The principal that created the client ID. This allows the server to authenticate the client when it sends EXCHANGE_ID.
- o The SSV, if SP4_SSV state protection was specified when the client ID was created (see [Section 18.35](#)). This lets the client create new sessions, and associate connections with the new and existing sessions.
- o The properties of the client ID as defined in [Section 18.35](#).

A persistent reply cache places certain demands on the server. The execution of the sequence of operations (starting with SEQUENCE) and placement of its results in the persistent cache MUST be atomic. If a client retries an sequence of operations that was previously executed on the server the only acceptable outcomes are either the original cached reply or an indication that client ID or session has been lost (indicating a catastrophic loss of the reply cache or a session that has been deleted because the client failed to use the session for an extended period of time).

A server could fail and restart in the middle of a COMPOUND procedure that contains one or more non-idempotent or idempotent-but-modifying operations. This creates an even higher challenge for atomic execution and placement of results in the reply cache. One way to view the problem is as a single transaction consisting of each operation in the COMPOUND followed by storing the result in persistent storage, then finally a transaction commit. If there is a failure before the transaction is committed, then the server rolls back the transaction. If server itself fails, then when it restarts, its recovery logic could roll back the transaction before starting the NFSv4.1 server.

While the description of the implementation for atomic execution of the request and caching of the reply is beyond the scope of this document, an example implementation for NFSv2 [\[27\]](#) is described in

[\[28\]](#).

[2.10.7.](#) RDMA Considerations

A complete discussion of the operation of RPC-based protocols over RDMA transports is in [\[8\]](#). A discussion of the operation of NFSv4, including NFSv4.1, over RDMA is in [\[9\]](#). 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.

[2.10.7.1.](#) 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 basic registration, these buffers must be posted by the RPC layer to handle receives. These buffers remain in use by the RPC/NFSv4.1 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. If multiple RDMA connections are associated with a session, then if the total number of credits across all RDMA connections associated with the session is X, and the number slots in the session is Y, then the maximum number of outstanding requests is lesser of X and Y.

[2.10.7.2](#). Flow Control

Previous versions of NFS do not provide 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 18.36](#)). These limits then provide

the maxima which each connection associated with the session's channel(s) must remain within. RDMA connections are managed within these limits as described in [section 3.3](#) ("Flow Control"[[Comment.2: RFC Editor: please verify section and title of the RPCRDMA document which is currently at <http://tools.ietf.org/html/draft-ietf-nfsv4-rpcrdma-08#section-3.3>]]) of [8]; if there are multiple RDMA connections, then the maximum number of requests for a channel will be divided among the RDMA

connections. Put a different way, the onus is on the replier to ensure that total number of RDMA credits across all connections associated with the replier's channel does exceed the channel's maximum number of outstanding requests.

The limits may also be modified dynamically at the replier's choosing by manipulating certain parameters present in each NFSv4.1 reply. In addition, the CB_RECALL_SLOT callback operation (see [Section 20.8](#)) can be sent 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.

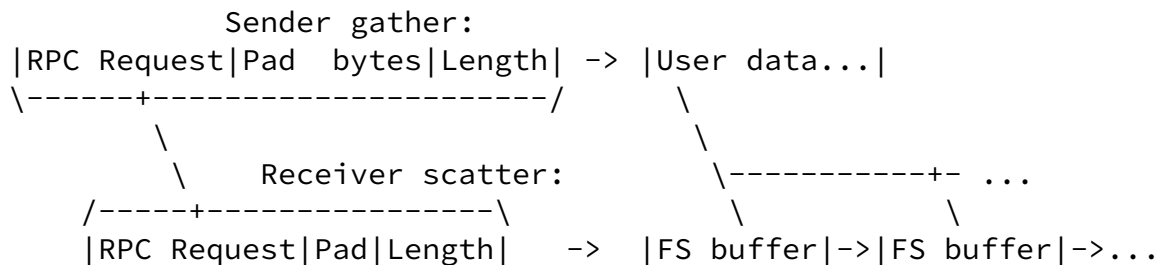
[2.10.7.3](#). Padding

Header padding is requested by each peer at session initiation (see the `ca_headerpadsize` argument to `CREATE_SESSION` in [Section 18.36](#)), and subsequently used by the RPC RDMA layer, as described in [\[8\]](#). Zero padding is permitted.

Padding leverages the useful property that RDMA 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,

COMPOUNDS, sizes of strings such as those passed to RENAME, etc. all go into the determination of a maximal NFSv4.1 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. [[Comment.3: RFC editor please keep this diagram on one page.]]



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.

[2.10.7.4.](#) Dual RDMA and Non-RDMA Transports

Some RDMA transports (e.g., [\[10\]](#)), permit 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 associated with a session, the client and server negotiate whether the connection is used in RDMA or non-RDMA mode (see [Section 18.36](#) and [Section 18.34](#)).

[2.10.8.](#) Sessions Security

[2.10.8.1.](#) Session Callback Security

Via session / connection association, NFSv4.1 improves security over that provided by NFSv4.0 for the backchannel. The connection is client-initiated (see [Section 18.34](#)), and subject to the same firewall and routing checks as the fore channel. The connection cannot be hijacked by an attacker who connects to the client port prior to the intended server as is possible with NFSv4.0. At the client's option (see [Section 18.35](#)), connection association is fully authenticated before being activated (see [Section 18.34](#)). Traffic

from the server over the backchannel is authenticated exactly as the client specifies (see [Section 2.10.8.2](#)).

[2.10.8.2](#). Backchannel RPC Security

When the NFSv4.1 client establishes the backchannel, it informs the server of the security flavors and principals to use when sending requests. 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 of the flavor/principal combinations the client offers, but it MUST NOT use unoffered combinations. This way, the client need not provide a target GSS principal for the backchannel as it did with NFSv4.0, nor the server have to implement an RPCSEC_GSS initiator as it did with NFSv4.0 [[20](#)].

The CREATE_SESSION ([Section 18.36](#)) and BACKCHANNEL_CTL ([Section 18.33](#)) operations allow the client to specify flavor/principal combinations.

Also note that the SP4_SSV state protection mode (see [Section 18.35](#) and [Section 2.10.8.3](#)) has the side benefit of providing SSV-derived RPCSEC_GSS contexts ([Section 2.10.9](#)).

[2.10.8.3](#). Protection from Unauthorized State Changes

As described to this point in the specification, the state model of NFSv4.1 is vulnerable to an attacker that sends a SEQUENCE operation with a forged session ID and with a slot ID that it expects the legitimate client to use next. When the legitimate client uses the slot ID with the same sequence number, the server returns the attacker's result from the reply cache which disrupts the legitimate client and thus denies service to it. Similarly an attacker could send a CREATE_SESSION with a forged client ID to create a new session associated with the client ID. The attacker could send requests using the new session that change locking state, such as LOCKU operations to release locks the legitimate client has acquired. Setting a security policy on the file which requires RPCSEC_GSS credentials when manipulating the file's state is one potential work around, but has the disadvantage of preventing a legitimate client from releasing state when RPCSEC_GSS is required to do so, but a GSS context cannot be obtained (possibly because the user has logged off the client).

NFSv4.1 provides three options to a client for state protection which are specified when a client creates a client ID via EXCHANGE_ID ([Section 18.35](#)).

The first (SP4_NONE) is to simply waive state protection.

The other two options (SP4_MACH_CRED and SP4_SSV) share several traits:

- o An RPCSEC_GSS-based credential is used to authenticate client ID and session maintenance operations, including creating and destroying a session, associating a connection with the session, and destroying the client ID.
- o Because RPCSEC_GSS is used to authenticate client ID and session maintenance, the attacker cannot associate a rogue connection with a legitimate session, or associate a rogue session with a legitimate client ID in order to maliciously alter the client ID's lock state via CLOSE, LOCKU, DELEGRETURN, LAYOUTRETURN, etc.
- o In cases where the server's security policies on a portion of its namespace require RPCSEC_GSS authentication, a client may have to use an RPCSEC_GSS credential to remove per-file state (e.g., LOCKU, CLOSE, etc.). The server may require that the principal that removes the state match certain criteria (e.g., the principal might have to be the same as the one that acquired the state). However, the client might not have an RPCSEC_GSS context for such a principal, and might not be able to create such a context (perhaps because the user has logged off). When the client establishes SP4_MACH_CRED or SP4_SSV protection, it can specify a list of operations that the server MUST allow using the machine credential (if SP4_MACH_CRED is used) or the SSV credential (if SP4_SSV is used).

The SP4_MACH_CRED state protection option uses a machine credential where the principal that creates the client ID, must also be the principal that performs client ID and session maintenance operations. The security of the machine credential state protection approach depends entirely on safe guarding the per-machine credential. Assuming a proper safe guard, using the per-machine credential for operations like CREATE_SESSION, BIND_CONN_TO_SESSION, DESTROY_SESSION, and DESTROY_CLIENTID will prevent an attacker from associating a rogue connection with a session, or associating a rogue session with a client ID.

There are at least three scenarios for the SP4_MACH_CRED option:

1. That the system administrator configures a unique, permanent per-machine credential for one of the mandated GSS mechanisms (e.g., if Kerberos V5 is used, a "keytab" containing a principal derived from a client host name could be used).
2. The client is used by a single user, and so the client ID and its sessions are used by just that user. If the user's credential

expires, then session and client ID maintenance cannot occur, but since the client has a single user, only that user is inconvenienced.

3. The physical client has multiple users, but the client implementation has a unique client ID for each user. This is effectively the same as the second scenario, but a disadvantage is that each user must be allocated at least one session each, so the approach suffers from lack of economy.

The SP4_SSV protection option uses a Secret State Verifier (SSV) which is shared between a client and server. The SSV serves as the secret key for an internal (that is, internal to NFSv4.1) GSS mechanism that uses the secret key for Message Integrity Code (MIC) and Wrap tokens ([Section 2.10.9](#)). The SP4_SSV protection option is intended for the client that has multiple users, and the system administrator does not wish to configure a permanent machine credential for each client. The SSV is established on the server via SET_SSV (see [Section 18.47](#)). To prevent eavesdropping, a client SHOULD send SET_SSV via RPCSEC_GSS with the privacy service. Several aspects of the SSV make it intractable for an attacker to guess the SSV, and thus associate rogue connections with a session, and rogue sessions with a client ID:

- o The arguments to and results of SET_SSV include digests of the old and new SSV, respectively.
- o Because the initial value of the SSV is zero, therefore known, the client that opts for SP4_SSV protection and opts to apply SP4_SSV protection to BIND_CONN_TO_SESSION and CREATE_SESSION MUST send at least one SET_SSV operation before the first BIND_CONN_TO_SESSION operation or before the second CREATE_SESSION operation on a

client ID. If it does not, the SSV mechanism will not generate tokens ([Section 2.10.9](#)). A client SHOULD send SET_SSV as soon as a session is created.

- o A SET_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. Each time a new principal uses a client ID for the first time, the client SHOULD send a SET_SSV with that principal's RPCSEC_GSS credentials, with RPCSEC_GSS service set to RPC_GSS_SVC_PRIVACY.

Here are the types of attacks that can be attempted by an attacker named Eve on a victim named Bob, and how SP4_SSV protection foils each attack:

- o Suppose Eve is the first user to log into a legitimate client. Eve's use of an NFSv4.1 file system will cause the legitimate client to create a client ID with SP4_SSV protection, specifying that the BIND_CONN_TO_SESSION operation MUST use the SSV credential. Eve's use of the file system also causes an SSV to be created. The SET_SSV operation that creates the SSV will be protected by the RPCSEC_GSS context created by the legitimate client which uses Eve's GSS principal and credentials. Eve can eavesdrop on the network while her RPCSEC_GSS context is created, and the SET_SSV using her context is sent. Even if the legitimate client sends the SET_SSV with RPC_GSS_SVC_PRIVACY, because Eve knows her own credentials, she can decrypt the SSV. Eve can compute an RPCSEC_GSS credential that BIND_CONN_TO_SESSION will accept, and so associate a new connection with the legitimate session. Eve can change the slot ID and sequence state of a legitimate session, and/or the SSV state, in such a way that when Bob accesses the server via the same legitimate client, the legitimate client will be unable to use the session.

The client's only recourse is to create a new client ID for Bob to use, and establish a new SSV for the client ID. The client will be unable to delete the old client ID, and will let the lease on the old client ID expire.

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 associating a rogue connection with the session.

The question is how did 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 sent a SET_SSV, which leads to following sub-scenarios:

- * Let us suppose that from the rogue connection, Eve sent a SET_SSV with the same slot ID and sequence ID that the legitimate client later uses. The server will assume the SET_SSV sent with Bob's credentials is a retry, 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 an indication to the client that the session has apparently been hijacked.

- * Alternatively, Eve sent a SET_SSV with a different slot ID than the legitimate client uses for its SET_SSV. Then the digest verification of the SET_SSV sent with Bob's credentials fails on the server, and the error returned to the client makes it apparent that the session has been hijacked.
- * Alternatively, Eve sent an operation other than SET_SSV, but with the same slot ID 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 a server bug, and either way destroys the old session.
- o Eve associates a rogue connection with the session as above, and then destroys the session. Again, Bob goes to use the server from

the legitimate client, which sends a SET_SSV using Bob's credentials. The client receives an error that indicates the session does not exist. When the client tries to create a new session, this will fail because the SSV it has does not match that the server has, and now the client knows the session was hijacked. The legitimate client establishes a new client ID.

- o 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 send a SET_SSV that will pass the digest verification check. However because the new connection has not been associated with the session, the SET_SSV is rejected for that reason.

In summary, an attacker's disruption of state when SP4_SSV protection is in use is limited to the formative period of a client ID, its first session, and the establishment of the SSV. Once a non-malicious user uses the client ID, the client quickly detects any hijack and rectifies the situation. Once a non-malicious user successfully modifies the SSV, the attacker cannot use NFSv4.1 operations to disrupt the non-malicious user.

Note that neither the SP4_MACH_CRED nor SP4_SSV protection approaches prevent hijacking of a transport connection that has previously been associated with a session. If the goal of a counter threat strategy is to prevent connection hijacking, the use of IPsec is RECOMMENDED.

If a connection hijack occurs, the hijacker could in theory change

locking state and negatively impact the service to legitimate clients. However if the server is configured to require the use of RPCSEC_GSS with integrity or privacy on the affected file objects, and if EXCHGID4_FLAG_BIND_PRINC_STATEID capability ([Section 18.35](#)), is in force, this will thwart unauthorized attempts to change locking state.

[2.10.9](#). The SSV GSS Mechanism

The SSV provides the secret key for a mechanism that NFSv4.1 uses for state protection. Contexts for this mechanism are not established

via the RPCSEC_GSS protocol. Instead, the contexts are automatically created when EXCHANGE_ID specifies SP4_SSV protection. The only tokens defined are the PerMsgToken (emitted by GSS_GetMIC) and the SealedMessage token (emitted by GSS_Wrap).

The mechanism OID for the SSV mechanism is:

iso.org.dod.internet.private.enterprise.Michael Eisler.nfs.ssv_mech (1.3.6.1.4.1.2882.1.1). While the SSV mechanism does not define any initial context tokens, the OID can be used to let servers indicate that the SSV mechanism is acceptable whenever the client sends a SECINFO or SECINFO_NO_NAME operation (see [Section 2.6](#)).

The SSV mechanism defines four subkeys derived from the SSV value. Each time SET_SSV is invoked the subkeys are recalculated by the client and server. The calculation of each of the four subkeys depends on each of the four respective ssv_subkey4 enumerated values. The calculation uses the HMAC [[11](#)], algorithm, using the current SSV as the key, the one way hash algorithm as negotiated by EXCHANGE_ID, and the input text as represented by the XDR encoded enumeration of type ssv_subkey4.

```
/* Input for computing subkeys */
enum ssv_subkey4 {
    SSV4_SUBKEY_MIC_I2T      = 1,
    SSV4_SUBKEY_MIC_T2I      = 2,
    SSV4_SUBKEY_SEAL_I2T     = 3,
    SSV4_SUBKEY_SEAL_T2I     = 4
};
```

The subkey derived from SSV4_SUBKEY_MIC_I2T is used for calculating message integrity codes (MICs) that originate from the NFSv4.1 client, whether as part of a request over the fore channel, or a response over the backchannel. The subkey derived from SSV4_SUBKEY_MIST2I is used for MICs originating from the NFSv4.1 server. The subkey derived from SSV4_SUBKEY_SEAL_I2T is used for encryption text

originating from the NFSv4.1 client and the subkey derived from SSV4_SUBKEY_SEAL_T2I is used for encryption text originating from the NFSv4.1 server.

The PerMsgToken description is based on an XDR definition:

```
/* Input for computing smt_hmac */
struct ssv_mic_plain_tkn4 {
    uint32_t      smpt_ssv_seq;
    opaque        smpt_orig_plain<>;
};
```

```
/* SSV GSS PerMsgToken token */
struct ssv_mic_tkn4 {
    uint32_t      smt_ssv_seq;
    opaque        smt_hmac<>;
};
```

The field `smt_hmac` is an HMAC calculated by using the subkey derived from `SSV4_SUBKEY_MIC_I2T` or `SSV4_SUBKEY_MIC_T2I` as the key, the one way hash algorithm as negotiated by `EXCHANGE_ID`, and the input text as represented by data of type `ssv_mic_plain_tkn4`. The field `smpt_ssv_seq` is the same as `smt_ssv_seq`. The field `smpt_orig_plain` is the "message" input passed to `GSS_GetMIC()` (see Section 2.3.1 of [7]). The caller of `GSS_GetMIC()` provides a pointer to a buffer containing the plain text. The SSV mechanism's entry point for `GSS_GetMIC()` encodes this into an opaque array, and the encoding will include an initial four byte length, plus any necessary padding. Prepend to this will be the XDR encoded value of `smpt_ssv_seq` thus making up an XDR encoding of a value of data type `ssv_mic_plain_tkn4`, which in turn is the input into the HMAC.

The token emitted by `GSS_GetMIC()` is XDR encoded and of XDR data type `ssv_mic_tkn4`. The field `smt_ssv_seq` comes from the SSV sequence number which is equal to 1 after `SET_SSV` ([Section 18.47](#)) is called the first time on a client ID. Thereafter, it is incremented on each `SET_SSV`. Thus `smt_ssv_seq` represents the version of the SSV at the time `GSS_GetMIC()` was called. As noted in [Section 18.35](#), the client and server can maintain multiple concurrent versions of the SSV. This allows the SSV to be changed without serializing all RPC calls that use the SSV mechanism with `SET_SSV` operations. Once the HMAC is calculated, it is XDR encoded into `smt_hmac`, which will include an initial four byte length, and any necessary padding. Prepend to

this will be the XDR encoded value of `smt_ssv_seq`.

The `SealedMessage` description is based on an XDR definition:

```
/* Input for computing ssct_encr_data and ssct_hmac */
struct ssv_seal_plain_tkn4 {
    opaque          sspt_confounder<>;
    uint32_t        sspt_ssv_seq;
    opaque          sspt_orig_plain<>;
    opaque          sspt_pad<>;
};
```

```
/* SSV GSS SealedMessage token */
struct ssv_seal_cipher_tkn4 {
    uint32_t        ssct_ssv_seq;
    opaque          ssct_iv<>;
    opaque          ssct_encr_data<>;
    opaque          ssct_hmac<>;
};
```

The token emitted by `GSS_Wrap()` is XDR encoded and of XDR data type `ssv_seal_cipher_tkn4`.

The `ssct_ssv_seq` field has the same meaning as `smt_ssv_seq`.

The `ssct_encr_data` field is the result of encrypting a value of the XDR encoded data type `ssv_seal_plain_tkn4`. The encryption key is the subkey derived from `SSV4_SUBKEY_SEAL_I2T` or `SSV4_SUBKEY_SEAL_T2I`, and the encryption algorithm is that negotiated by `EXCHANGE_ID`.

The `ssct_iv` field is the initialization vector (IV) for the encryption algorithm (if applicable) and is sent in clear text. The content and size of the IV MUST comply with specification of the encryption algorithm. For example, the `id-aes256-CBC` algorithm MUST use a 16 byte initialization vector (IV) which MUST be unpredictable for each instance of a value of type `ssv_seal_plain_tkn4` that is encrypted with a particular SSV key.

The `ssct_hmac` field is the result of computing an HMAC using value of the XDR encoded data type `ssv_seal_plain_tkn4` as the input text. The key is the subkey derived from `SSV4_SUBKEY_MIC_I2T` or `SSV4_SUBKEY_MIC_T2I`, and the one way hash algorithm is that

negotiated by EXCHANGE_ID.

The `sspt_confounder` field is a random value.

The `sspt_ssv_seq` field is the same as `ssvt_ssv_seq`.

The field `sspt_orig_plain` field is the original plaintext and is the "input_message" input passed to `GSS_Wrap()` (see Section 2.3.3 of [7]). As with the handling of the plaintext by the SSV mechanism's `GSS_GetMIC()` entry point, the entry point for `GSS_Wrap()` expects a pointer to the plaintext, and will XDR encode an opaque array into `sspt_orig_plain` representing the plain text, along with the other fields of an instance of data type `ssv_seal_plain_tkn4`.

The `sspt_pad` field is present to support encryption algorithms that require inputs to be in fixed sized blocks. The content of `sspt_pad` is zero filled except for the length. Beware that the XDR encoding of `ssv_seal_plain_tkn4` contains three variable length arrays, and so each array consumes four bytes for an array length, and each array that follows the length is always padded to a multiple of four bytes per the XDR standard.

For example suppose the encryption algorithm uses 16 byte blocks, and the `sspt_confounder` is three bytes long, and the `sspt_orig_plain` field is 15 bytes long. The XDR encoding of `sspt_confounder` uses eight bytes (4 + 3 + 1 byte pad), the XDR encoding of `sspt_ssv_seq` uses four bytes, the XDR encoding of `sspt_orig_plain` uses 20 bytes (4 + 15 + 1 byte pad), and the smallest XDR encoding of the `sspt_pad` field is four bytes. This totals 36 bytes. The next multiple of 16 is 48, thus the length field of `sspt_pad` needs to be set to 12 bytes, or a total encoding of 16 bytes. The total number of XDR encoded bytes is thus $8 + 4 + 20 + 16 = 48$.

`GSS_Wrap()` emits a token that is an XDR encoding of a value of data type `ssv_seal_cipher_tkn4`. Note that regardless whether the caller of `GSS_Wrap()` requests confidentiality or not, the token always has confidentiality. This is because the SSV mechanism is for `RPCSEC_GSS`, and `RPCSEC_GSS` never produces `GSS_wrap()` tokens without confidentiality.

There is one SSV per client ID. Effectively there is a single GSS context for a client ID / SSV pair. All SSV mechanism `RPCSEC_GSS`

handles of a client ID / SSV pair share the same GSS context. SSV GSS contexts do not expire except when the SSV is destroyed (causes would include the client ID being destroyed or a server restart). Since one purpose of context expiration is to replace keys that have been in use for "too long" hence vulnerable to compromise by brute force or accident, the client can replace the SSV key by sending periodic SET_SSV operations, by cycling through different users' RPCSEC_GSS credentials. This way the SSV is replaced without

destroying the SSV's GSS contexts.

SSV RPCSEC_GSS handles can be expired or deleted by the server at any time and the EXCHANGE_ID operation can be used to create more SSV RPCSEC_GSS handles. Expiration of SSV RPCSEC_GSS handles does not imply that the SSV or its GSS context have expired.

The client MUST establish an SSV via SET_SSV before the SSV GSS context can be used to emit tokens from GSS_Wrap() and GSS_GetMIC(). If SET_SSV has not been successfully called, attempts to emit tokens MUST fail.

The SSV mechanism does not support replay detection and sequencing in its tokens because RPCSEC_GSS does not use those features (See [Section 5.2.2](#) "Context Creation Requests" in [4]).

[2.10.10](#). Session Mechanics - Steady State

[2.10.10.1](#). 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 backchannel connections). If these resources vanish, the server takes action as specified in [Section 2.10.12.2](#).

[2.10.10.2](#). Obligations of the Client

The client SHOULD honor the following obligations in order to utilize the session:

- o 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 an inactive session. A server MAY consider a session to be inactive if the client has not used the session before the session inactivity timer ([Section 2.10.11](#)) has expired.

- o Destroy the session when not needed. If a client has multiple sessions, one of which has no requests waiting for replies, and has been idle for some period of time, it SHOULD destroy the session.
- o Maintain GSS contexts for the backchannel. 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.

- o Preserve a connection for a backchannel. The server requires a backchannel in order to gracefully recall recallable state, or notify the client of certain events. Note that if the connection is not being used for the fore channel, there is no way for the client tell if the connection is still alive (e.g., the server restarted without sending a disconnect). The onus is on the server, not the client, to determine if the backchannel's connection is alive, and to indicate in the response to a SEQUENCE operation when the last connection associated with a session's backchannel has disconnected.

[2.10.10.3](#). Steps the Client Takes To Establish a Session

If the client does not have a client ID, the client sends EXCHANGE_ID to establish a client ID. If it opts for SP4_MACH_CRED or SP4_SSV protection, in the spo_must_enforce list of operations, it SHOULD at minimum specify: CREATE_SESSION, DESTROY_SESSION, BIND_CONN_TO_SESSION, BACKCHANNEL_CTL, and DESTROY_CLIENTID. If opts for SP4_SSV protection, the client needs to ask for SSV-based RPCSEC_GSS handles.

The client uses the client ID to send a CREATE_SESSION on a connection to the server. The results of CREATE_SESSION indicate whether the server will persist the session reply cache through a

server restarted or not, and the client notes this for future reference.

If the client specified SP4_SSV state protection when the client ID was created, then it SHOULD send SET_SSV in the first COMPOUND after the session is created. Each time a new principal goes to use the client ID, it SHOULD send a SET_SSV again.

If the client wants to use delegations, layouts, directory notifications, or any other state that requires a backchannel, 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 associate the connection with 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 selected SP4_MACH_CRED or SP4_SSV protection when it called EXCHANGE_ID, then the client SHOULD specify that the backchannel use RPCSEC_GSS contexts for security.

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 fore channel, then it must call BIND_CONN_TO_SESSION if it specified SP4_SSV or SP4_MACH_CRED state protection when the client ID was created.

At this point the session has reached steady state.

[2.10.11](#). Session Inactivity Timer

The server MAY maintain a session inactivity timer for each session. If the session inactivity timer expires, then the server MAY destroy the session. To avoid losing a session due to inactivity, the client MUST renew the session inactivity timer. The length of session inactivity timer MUST NOT be less than the lease_time attribute ([Section 5.8.1.11](#)). As with lease renewal ([Section 8.3](#)), when the server receives a SEQUENCE operation, it resets the session inactivity timer, and MUST NOT allow the timer to expire while the rest of the operations in the COMPOUND procedure's request are still executing. Once the last operation has finished, the server MUST set

the session inactivity timer to expire no sooner than the sum of the current time and the value of the `lease_time` attribute.

[2.10.12.](#) Session Mechanics - Recovery

[2.10.12.1.](#) Events Requiring Client Action

The following events require client action to recover.

[2.10.12.1.1.](#) RPCSEC_GSS Context Loss by Callback Path

If all RPCSEC_GSS contexts granted 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 18.46.3](#)).

[2.10.12.1.2.](#) Connection Loss

If the client loses the last connection of the session, and if wants to retain the session, then it must create a new connection, and if, when the client ID was created, BIND_CONN_TO_SESSION was specified in the `spo_must_enforce` list, the client MUST use BIND_CONN_TO_SESSION to associate the connection with the session.

If there was a request outstanding at the time the connection loss, then if client wants to continue to use the session it MUST retry the request, as described in [Section 2.10.6.2](#). 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 lost connection. As long as the session ID, slot ID, and sequence ID in the retry match that of the original request, the server will recognize the request as a retry if it executed the request prior to disconnect.

If the connection that was lost was the last one associated with the backchannel, and the client wants to retain the backchannel and/or not put recallable state subject to revocation, the client must reconnect, and if it does, it MUST associate the connection to the session and backchannel via BIND_CONN_TO_SESSION. The server SHOULD indicate when it has no callback connection via the `sr_status_flags`

result from SEQUENCE.

[2.10.12.1.3.](#) 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. If the client wants to retain the backchannel and/or not put recallable state subject to revocation, the client must use `BACKCHANNEL_CTL` to assign new contexts.

[2.10.12.1.4.](#) Loss of Session

The replier might lose a record of the session. Causes include:

- o Replier failure and restart
- o A catastrophe that causes the reply cache to be corrupted or lost on the media it was stored on. This applies even if the replier indicated in the `CREATE_SESSION` results that it would persist the cache.
- o The server purges the session of a client that has been inactive for a very extended period of time.
- o As a result of configuration changes among a set of clustered servers, a network address previously connected to one server becomes connected to a different server which has no knowledge of the session in question. Such a configuration change will generally only happen when the original server ceases to function for a time.

Loss of reply cache is equivalent to loss of session. The replier indicates loss of session to the requester by returning `NFS4ERR_BADSESSION` on the next operation that uses the session ID that refers to the lost session.

After an event like a server restart, 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 connection association enforcement when the session was created, it invokes `BIND_CONN_TO_SESSION` using the session ID. Otherwise, it invokes

SEQUENCE. If BIND_CONN_TO_SESSION or SEQUENCE returns NFS4ERR_BADSESSION, the client knows the session is not available to it when communicating with that network address. If the connection survives session loss, then the next SEQUENCE operation the client sends over the connection will get back NFS4ERR_BADSESSION. The client again knows the session was lost.

Here is one suggested algorithm for the client when it gets NFS4ERR_BADSESSION. It is not obligatory in that, if a client does not want to take advantage of such features as trunking, it may omit parts of it. However, it is a useful example which draws attention to various possible recovery issues:

1. If the client has other connections to other server network addresses associated with the same session, attempt a COMPOUND with a single operation, SEQUENCE, on each of the other connections.
2. If the attempts succeed, the session is still alive, and this is a strong indicator the server's network address has moved. The client might send an EXCHANGE_ID on the connection that returned NFS4ERR_BADSESSION to see if there are opportunities for client ID trunking (i.e. the same client ID and so_major are returned). The client might use DNS to see if the moved network address was replaced with another, so that the performance and availability benefits of session trunking can continue.
3. If the SEQUENCE requests fail with NFS4ERR_BADSESSION then the session no longer exists on any of the server network addresses the client has connections associated with that session ID. It is possible the session is still alive and available on other network addresses. The client sends an EXCHANGE_ID on all the connections to see if the server owner is still listening on those network addresses. If the same server owner is returned, but a new client ID is returned, this is a strong indicator of a server restart. If both the same server owner and same client ID are returned, then this is a strong indication that the server did delete the session, and the client will need to send a CREATE_SESSION if it has no other sessions for that client ID. If a different server owner is returned, the client can use DNS to find other network addresses. If it does not, or if DNS does not find any other addresses for the server, then the client will be unable to provide NFSv4.1 service, and fatal errors should be

returned to processes that were using the server. If the client is using a "mount" paradigm, unmounting the server is advised.

4. If the client knows of no other connections associated with the session ID, and server network addresses that are, or have been associated with the session ID, then the client can use DNS to find other network addresses. If it does not, or if DNS does not find any other addresses for the server, then the client will be unable to provide NFSv4.1 service, and fatal errors should be returned to processes that were using the server. If the client is using a "mount" paradigm, unmounting the server is advised.

If there is a reconfiguration event which results in the same network being assigned to servers where the `eir_server_scope` value is different, it cannot be guaranteed that a session ID generated by the first will be recognized as invalid by the first. Therefore, in managing server reconfigurations among servers with different server scope values, it is necessary to make sure that all clients have disconnected from the first server before effecting the reconfiguration. Nonetheless, clients should not assume that servers will always adhere to this requirement; clients **MUST** be prepared to deal with unexpected effects of server reconfigurations. Even where a session ID is inappropriately recognized as valid, it is likely that either the connection will not be recognized as valid, or that a sequence value for a slot will not be correct. Therefore, when a client receives results indicating such unexpected errors, the use of `EXCHANGE_ID` to determine the current server configuration and present the client to the server is **RECOMMENDED**.

A variation on the above is that after a server's network address moves, there is no NFSv4.1 server listening. E.g. no listener on port 2049, the NFSv4 server returns `NFS4ERR_MINOR_VERS_MISMATCH`, the NFS server returns a `PROG_MISMATCH` error, the RPC listener on 2049 returns `PROG_MISMATCH`, or attempts to re-connect to the network address timeout. These **SHOULD** be treated as equivalent to `SEQUENCE` returning `NFS4ERR_BADSESSION` for these purposes.

When the client detects session loss, it must call `CREATE_SESSION` to recover. Any non-idempotent operations that were in progress might 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 because locks, opens, delegations, and layouts are tied to the client ID and depend on the client ID, not the session. Nor does loss of lock, open, delegation, or layout state imply loss of session state, because the session depends on the client ID; loss of client ID however does imply loss of session,

Internet-Draft

NFSv4.1

September 2008

lock, open, delegation, and layout state. See [Section 8.4.2](#). A session can survive a server restart, but lock recovery may still be needed.

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

[2.10.12.2](#). Events Requiring Server Action

The following events require server action to recover.

[2.10.12.2.1](#). Client Crash and Restart

As described in [Section 18.35](#), a restarted client sends EXCHANGE_ID in such a way it causes the server to delete any sessions it had.

[2.10.12.2.2](#). Client Crash with No Restart

If a client crashes and never comes back, it will never send 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 as well as locking state.

[2.10.12.2.3](#). Extended Network Partition

To the server, the extended network partition may be no different from a client crash with no restart (see [Section 2.10.12.2.2](#)). Unless the server can discern that there is a network partition, it is free to treat the situation as if the client has crashed permanently.

[2.10.12.2.4](#). Backchannel Connection Loss

If there were callback requests outstanding at the time of a connection loss, then the server MUST retry the request, as described in [Section 2.10.6.2](#). 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 lost connection. As long as the session ID, slot ID, and sequence ID in the retry match that of the original request, the callback target will recognize the request as a

retry even if it did see the request prior to disconnect.

If the connection lost is the last one associated with the backchannel, then the server MUST indicate that in the `sr_status_flags` field of every SEQUENCE reply until the backchannel is reestablished. There are two situations each of which use

different status flags: no connectivity for the session's backchannel, and no connectivity for any session backchannel of the client. See [Section 18.46](#) for a description of the appropriate flags in `sr_status_flags`.

[2.10.12.2.5](#). GSS Context Loss

The server SHOULD monitor when the number RPCSEC_GSS contexts assigned to the backchannel reaches one, and when that one context is near expiry (i.e. between one and two periods of lease time), indicate so in the `sr_status_flags` field of all SEQUENCE replies. The server MUST indicate when the all of the backchannel's assigned RPCSEC_GSS contexts have expired in the `sr_status_flags` field of all SEQUENCE replies.

[2.10.13](#). 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 Constants and Data Types

The syntax and semantics to describe the data types of the NFSv4.1 protocol are defined in the XDR [RFC4506](#) [2] and RPC [RFC1831](#) [3] documents. The next sections build upon the XDR data types to define constants, types and structures specific to this protocol. The full list of XDR data types is in [12].

Internet-Draft

NFSv4.1

September 2008

[3.1.](#) Basic Constants

```
const NFS4_FHSIZE           = 128;
const NFS4_VERIFIER_SIZE    = 8;
const NFS4_OPAQUE_LIMIT     = 1024;
const NFS4_SESSIONID_SIZE   = 16;

const NFS4_INT64_MAX        = 0x7fffffffffffffff;
const NFS4_UINT64_MAX       = 0xfffffffffffffff;
const NFS4_INT32_MAX        = 0x7fffffff;
const NFS4_UINT32_MAX       = 0xffffffff;

const NFS4_MAXFILELEN       = 0xfffffffffffffff;
const NFS4_MAXFILEOFF       = 0xfffffffffffffe;
```

Except where noted, all these constants are defined in bytes.

- o NFS4_FHSIZE is the maximum size of a filehandle.
- o NFS4_VERIFIER_SIZE is the fixed size of a verifier.
- o NFS4_OPAQUE_LIMIT is the maximum size of certain opaque information.
- o NFS4_SESSIONID_SIZE is the fixed size of a session identifier.
- o NFS4_INT64_MAX is the maximum value of a signed 64 bit integer.

- o NFS4_UINT64_MAX is the maximum value of an unsigned 64 bit integer.
- o NFS4_INT32_MAX is the maximum value of a signed 32 bit integer.
- o NFS4_UINT32_MAX is the maximum value of an unsigned 32 bit integer.
- o NFS4_MAXFILELEN is the maximum length of a regular file.
- o NFS4_MAXFILEOFF is the maximum offset into a regular file.

[3.2.](#) Basic Data Types

These are the base NFSv4.1 data types.

Data Type	Definition
int32_t	typedef int int32_t;

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

nfs_ftype4	enum nfs_ftype4; Various defined file types.
nfsstat4	enum nfsstat4; Return value for operations.
offset4	typedef uint64_t offset4; Various offset designations (READ, WRITE, LOCK, COMMIT).
qop4	typedef uint32_t qop4; Quality of protection designation in SECINFO.
sec_oid4	typedef opaque sec_oid4<>; Security Object Identifier. The sec_oid4 data type is not really opaque. Instead it contains an ASN.1 OBJECT IDENTIFIER as used by GSS-API in the mech_type argument to GSS_Init_sec_context. See [7] for details.
sequenceid4	typedef uint32_t sequenceid4; Sequence number used for various session operations (EXCHANGE_ID, CREATE_SESSION, SEQUENCE, CB_SEQUENCE).
seqid4	typedef uint32_t seqid4; Sequence identifier used for file locking.
sessionid4	typedef opaque sessionid4[NFS4_SESSIONID_SIZE]; Session identifier.
slotid4	typedef uint32_t slotid4; Sequencing artifact for various session operations (SEQUENCE, CB_SEQUENCE).
utf8string	typedef opaque utf8string<>;

utf8str_cis	UTF-8 encoding for strings. typedef utf8string utf8str_cis; Case-insensitive UTF-8 string.
utf8str_cs	typedef utf8string utf8str_cs; Case-sensitive UTF-8 string.
utf8str_mixed	typedef utf8string utf8str_mixed; UTF-8 strings with a case sensitive prefix and a case insensitive suffix.
component4	typedef utf8str_cs component4; Represents path name components.
linktext4	typedef utf8str_cs linktext4; Symbolic link contents.
pathname4	typedef component4 pathname4<>; Represents path name for fs_locations.

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

End of Base Data Types

Table 1

[3.3.](#) Structured Data Types

[3.3.1.](#) nfstime4

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

The nfstime4 data type 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 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.

[3.3.2.](#) time_how4

```
enum time_how4 {
```

```

        SET_TO_SERVER_TIME4 = 0,
        SET_TO_CLIENT_TIME4 = 1
};

```

[3.3.3.](#) settime4

```

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

```

The time_how4 and settime4 data types are used for setting timestamps in file object attributes. If set_it is SET_TO_SERVER_TIME4, then the server uses its local representation of time for the time value.

[3.3.4.](#) specdata4

```

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

```

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

[3.3.5.](#) fsid4

```

struct fsid4 {
    uint64_t      major;
    uint64_t      minor;
};

```

[3.3.6.](#) chg_policy4

```

struct change_policy4 {
    uint64_t      cp_major;
    uint64_t      cp_minor;
};

```

The `chg_policy4` data type is used for the `change_policy` RECOMMENDED attribute. It provides change sequencing indication analogous to the `change` attribute. To enable the server to present a value valid across server re-initialization without requiring persistent storage, two 64-bit quantities are used, allowing one to be a server instance ID and the second to be incremented non-persistently, within a given server instance.

[3.3.7.](#) `fattr4`

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

The `fattr4` data type 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 $(n \bmod 32)$.

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

[3.3.8.](#) `change_info4`

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

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

Internet-Draft

NFSv4.1

September 2008

[3.3.9.](#) netaddr4

```
struct netaddr4 {
    /* see struct rpcb in RFC 1833 */
    string na_r_netid<>; /* network id */
    string na_r_addr<>; /* universal address */
};
```

The netaddr4 data type is used to identify network transport endpoints. The r_netid and r_addr fields respectively contain a netid and uaddr. The netid and uaddr concepts are defined in [\[13\]](#). The netid and uaddr formats for TCP over IPv4 and TCP over IPv6 are defined in [\[13\]](#), specifically Tables 2 and 3 and Sections [3.2.3.3](#) and 3.2.3.4.

[3.3.10.](#) state_owner4

```
struct state_owner4 {
    clientid4      clientid;
    opaque         owner<NFS4_OPAQUE_LIMIT>;
};
```

```
typedef state_owner4 open_owner4;
typedef state_owner4 lock_owner4;
```

The state_owner4 data type is the base type for the open_owner4 [Section 3.3.10.1](#) and lock_owner4 [Section 3.3.10.2](#).

[3.3.10.1.](#) open_owner4

This data type is used to identify the owner of open state.

[3.3.10.2.](#) lock_owner4

This structure is used to identify the owner of byte-range locking state.

[3.3.11.](#) open_to_lock_owner4

```
struct open_to_lock_owner4 {
    seqid4      open_seqid;
    stateid4    open_stateid;
    seqid4      lock_seqid;
```

```

        lock_owner4    lock_owner;
};

```

This data type 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.

[3.3.12.](#) stateid4

```

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

```

This data type is used for the various state sharing mechanisms between the client and server. The client never modifies a value of data type stateid. The starting value of the seqid field is undefined. The server is required to increment the seqid field by one (1) 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.

[3.3.13.](#) layouttype4

```

enum layouttype4 {
    LAYOUT4_NFSV4_1_FILES    = 0x1,
    LAYOUT4 OSD2_OBJECTS     = 0x2,
    LAYOUT4_BLOCK_VOLUME     = 0x3
};

```

This data type indicates what type of layout is being used. The file server advertises the layout types it supports through the fs_layout_type file system attribute ([Section 5.12.1](#)). A client asks for layouts of a particular type in LAYOUTGET, and processes those layouts in its layout-type-specific logic.

The layouttype4 data type 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 22.4](#); 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, as defined in [Section 13](#), is to be used. The LAYOUT4_OSD2_OBJECTS enumeration specifies that the object layout, as defined in [\[29\]](#), is to be used. Similarly, the LAYOUT4_BLOCK_VOLUME enumeration specifies that the block/volume layout, as defined in [\[30\]](#), is to be used.

[3.3.14](#). deviceid4

```
const NFS4_DEVICEID4_SIZE = 16;
```

```
typedef opaque deviceid4[NFS4_DEVICEID4_SIZE];
```

Layout information includes device IDs that specify a storage device through a compact handle. Addressing and type information is obtained with the GETDEVICEINFO operation. Device IDs are not guaranteed to be valid across metadata server restarts. A device ID is unique per client ID and layout type. See [Section 12.2.10](#) for more details.

[3.3.15](#). 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 data types 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 (see [Section 13.3](#)), 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.

[3.3.16.](#) 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.

[3.3.17.](#) layout4

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

The layout4 data type defines a layout for a file. The layout type specific data is opaque within lo_content. Since layouts are sub-dividable, the offset and length together with the file's filehandle, the client ID, iomode, and layout type, identify the layout.

[3.3.18.](#) layoutupdate4

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

The layoutupdate4 data type is used by the client to return updated

layout information to the metadata server via the LAYOUTCOMMIT ([Section 18.42](#)) operation. This data type provides a channel to pass layout type specific information (in field `lou_body`) back to the metadata server. E.g., for the block/volume layout type 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. The NFSv4.1 file-based layout does not use this data type; if `lou_type` is `LAYOUT4_NFSV4_1_FILES`, the `lou_body` field MUST have a zero length.

[3.3.19.](#) layouthint4

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

The `layouthint4` data type 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 data type specified by the `layout_hint` attribute described in [Section 5.12.4](#). 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` data type as defined in [Section 13.3](#).

[3.3.20.](#) layoutiomode4

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

The `iomode` specifies whether the client intends to just read or both read and write the data represented by the layout. While the `LAYOUTIOMODE4_ANY` `iomode` MUST NOT be used in the arguments to the `LAYOUTGET` operation, it MAY be used in the arguments to the `LAYOUTRETURN` and `CB_LAYOUTRECALL` operations. The `LAYOUTIOMODE4_ANY` `iomode` specifies that layouts pertaining to both `LAYOUTIOMODE4_READ` and `LAYOUTIOMODE4_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.

[3.3.21.](#) nfs_impl_id4

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

This data type is used to identify client and server implementation details. 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.

[3.3.22.](#) threshold_item4

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

This data type contains a list of hints specific to a layout type for helping the client determine when it should send I/O directly through the metadata server versus the storage devices. The data type 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 content is determined by the hintset bitmap. See the `mdsthreshold` attribute for more details.

The `thi_hintset` field is a bitmap of the following values:

+-----+-----+-----+-----+

name	#	Data Type	Description
threshold4_read_size	0	length4	The file size below which it is RECOMMENDED to read data through the MDS.
threshold4_write_size	1	length4	The file size below which it is RECOMMENDED to write data through the MDS.
threshold4_read_iosize	2	length4	For read I/O sizes below this threshold it is RECOMMENDED to read data through the MDS
threshold4_write_iosize	3	length4	For write I/O sizes below this threshold it is RECOMMENDED to write data through the MDS

[3.3.23.](#) mdsthreshold4

```
struct mdsthreshold4 {
    threshold_item4 mth_hints<>;
};
```

This data type holds an array of elements of data type `threshold_item4`, each of which is valid for a particular layout type. An array is necessary because 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 NFSv3 protocol [RFC1813](#) [21], 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](#) [31] and [RFC2055](#) [32]. With the use of the public filehandle in combination with the LOOKUP operation in the NFSv3 protocol, it has been demonstrated that the MOUNT protocol is unnecessary for viable interaction between NFS client and server.

Therefore, the NFSv4.1 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 7](#).

[4.1.2.](#) Public Filehandle

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

[4.2.](#) Filehandle Types

In the NFSv3 protocol, there was one type of filehandle with a single set of semantics. This type of filehandle is termed "persistent" in NFSv4.1. The semantics of a persistent filehandle remain the same as before. A new type of filehandle introduced in NFSv4.1 is the "volatile" filehandle, which attempts to accommodate certain server environments.

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

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

[4.2.1.](#) General Properties of a Filehandle

The filehandle contains all the information the server needs to distinguish an individual file. To the client, the filehandle is opaque. The client stores filehandles for use in a later request and can compare two filehandles from the same server for equality by doing a byte-by-byte comparison. However, the client **MUST NOT** otherwise interpret the contents of filehandles. If two filehandles from the same server are equal, they **MUST** refer to the same file. 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 10.3.4](#).

As an example, in the case that two different path names when

traversed at the server terminate at the same file system object, the server SHOULD return the same filehandle for each path. This can

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

[4.2.2.](#) Persistent Filehandle

A persistent filehandle is defined as having a fixed value for the lifetime of the file system object to which it refers. Once the server creates the filehandle for a file system object, the server MUST accept the same filehandle for the object for the lifetime of the object. If the server restarts, 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 REQUIRED attribute "fh_expire_type" is used by the client to

determine what type of filehandle the server is providing for a particular file system. This attribute is a bitmask with the following values:

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

FH4_VOLATILE_ANY The filehandle may expire at any time, except as specifically excluded (i.e. FH4_NO_EXPIRE_WITH_OPEN).

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

FH4_VOL_MIGRATION The filehandle will expire as a result of a file system transition (migration or replication), in those case 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_NOEXPIRE_WITH_OPEN not set), should deny a RENAME or REMOVE that would affect an OPEN file of any of the components leading to the OPEN file. In addition, the server should deny all RENAME or REMOVE requests during the grace period upon server restart.

Servers which provide volatile filehandles that may expire while open require special care as regards handling of RENAMES and REMOVES. This situation can arise if FH4_VOL_MIGRATION or FH4_VOL_RENAME is set, if FH4_VOLATILE_ANY is set and FH4_NOEXPIRE_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.

Volatile filehandles are especially suitable for implementation of the pseudo file systems used to bridge exports. See [Section 7.5](#) for a discussion of this.

[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_FHEXPIRED. If slot is out of range, return NFS4ERR_BADHANDLE. If the generation number does not match, return NFS4ERR_FHEXPIRED.

When the server restarts, 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_FHEXPIRED error. The client must take on additional responsibility so that it may prepare itself to recover from the expiration of a volatile filehandle. If the server returns

persistent filehandles, the client does not need these additional steps.

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

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

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

like:

```
RENAME A B
LOOKUP B
GETFH
```

Note that the COMPOUND procedure does not provide atomicity. This example only reduces the overhead of recovering from an expired filehandle.

[5.](#) File Attributes

To meet the requirements of extensibility and increased interoperability with non-UNIX platforms, attributes must be handled in a flexible manner. The NFSv3 `fattnr3` structure contains a fixed list of attributes that not all clients and servers are able to support or care about. The `fattnr3` structure can not be extended as

new needs arise and it provides no way to indicate non-support. With the NFSv4.1 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: REQUIRED, RECOMMENDED, and named. Both REQUIRED and RECOMMENDED attributes are supported in the NFSv4.1 protocol by a specific and well-defined encoding and are identified by number. They are requested by setting a bit in the bit vector sent in the GETATTR request; the server response includes a bit vector to list what attributes were returned in the response. New REQUIRED or RECOMMENDED attributes may be added to the NFSv4 protocol as part of a new minor version by publishing a standards-track RFC which allocates a new attribute number value and defines the encoding for the attribute. See [Section 2.7](#) for further discussion.

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

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

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

by bringing them to the IETF standards-track process.

The set of attributes which are classified as REQUIRED 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 REQUIRED if the data is both needed by a large number of clients and is not otherwise reasonably computable by the client when support is not provided on the server.

Note that the hidden directory returned by OPENATTR is a convenience for protocol processing. The client should not make any assumptions about the server's implementation of named attributes and whether 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.](#) REQUIRED Attributes

These MUST be supported by every NFSv4.1 client and server in order to ensure a minimum level of interoperability. The server MUST store and return these attributes and the client MUST be able to function with an attribute set limited to these attributes. With just the REQUIRED attributes some client functionality may be impaired or limited in some ways. A client may ask for any of these attributes to be returned by setting a bit in the GETATTR request and the server must return their value.

[5.2.](#) RECOMMENDED Attributes

These attributes are understood well enough to warrant support in the NFSv4.1 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 NFSv4 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 "named attribute directory" and further perusal and modification of the name space may be done using operations that work on more typical directories. In particular, REaddir may be used to get a list of such named attributes and LOOKUP and OPEN may select a particular attribute. Creation of a new named attribute may be the result of an OPEN specifying file creation.

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

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

Named attributes and the named attribute directory may be the target of delegations (in the case of the named attribute directory these will be directory delegations). However, since granting of delegations or not is within the server's discretion, a server need not support delegations on named attributes or the named attribute directory.

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 metadata with complete transparency from one location to another; this would imply that names allowed for regular directory entries are valid for named attribute names as well.

In NFSv4.1, the structure of named attribute directories is restricted in a number of ways, in order to prevent the development of non-interoperable implementations in which some servers support a fully general hierarchical directory structure for named attributes while others support a limited set, but fully adequate to the feature's goals. In such an environment, clients or applications might come to depend on non-portable extensions. The restrictions are:

- o CREATE is not allowed in a named attribute directory. Thus, such objects as symbolic links and special files are not allowed to be named attributes. Further, directories may not be created in a named attribute directory so no hierarchical structure of named attributes for a single object is allowed.
- o If OPENATTR is done on a named attribute directory or on a named attribute, the server MUST return NFS4ERR_WRONG_TYPE.
- o Doing a RENAME of a named attribute to a different named attribute directory or to an ordinary (i.e. non-named-attribute) directory is not allowed.
- o Creating hard links between named attribute directories or between named attribute directories and ordinary directories is not allowed.

Names of attributes will not be controlled by this document or other IETF standards track documents. See [Section 22.1](#) for further discussion.

[5.4.](#) Classification of Attributes

Each of the REQUIRED and RECOMMENDED attributes can be classified in one of three categories: per server, 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.

Internet-Draft

NFSv4.1

September 2008

- o The per server attribute is:

- lease_time

- o The per file system attributes are:

- supported_attrs, suppattnr_exclcreat, fh_expire_type,
link_support, symlink_support, unique_handles, aclsupport,
cansettime, case_insensitive, case_preserving,
chown_restricted, files_avail, files_free, files_total,
fs_locations, homogeneous, maxfilesize, maxname, maxread,
maxwrite, no_trunc, space_avail, space_free, space_total,
time_delta, change_policy, fs_status, fs_layout_type,
fs_locations_info, fs_charset_cap

- o The per file system object attributes are:

- type, change, size, named_attr, fsid, rdattnr_error, filehandle,
acl, archive, fileid, hidden, maxlink, mime_type, 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.](#) Set-Only and Get-Only Attributes

Some REQUIRED and RECOMMENDED attributes are set-only, i.e. they can be set via SETATTR but not retrieved via GETATTR. Similarly, some REQUIRED and RECOMMENDED attributes are get-only, i.e. they can be retrieved GETATTR but not set via SETATTR. If a client attempts to set a get-only attribute or get a set-only attributes, the server MUST return NFS4ERR_INVALID.

[5.6.](#) REQUIRED Attributes - List and Definition References

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

- o Name: the name of attribute
- o Id: the number assigned to the attribute. In the event of conflicts between the assigned number and [12], the latter is

authoritative.

- o Data Type: The XDR data type of the attribute.
- o Acc: Access allowed to the attribute. R means read-only (GETATTR may retrieve, SETATTR may not set). W means write-only (SETATTR may set, GETATTR may not retrieve). R W means read/write (GETATTR may retrieve, SETATTR may set).
- o Defined in: the section of this specification that describes the attribute.

Name	Id	Data Type	Acc	Defined in:
supported_attrs	0	bitmap4	R	Section 5.8.1.1
type	1	nfs_ftype4	R	Section 5.8.1.2
fh_expire_type	2	uint32_t	R	Section 5.8.1.3
change	3	uint64_t	R	Section 5.8.1.4
size	4	uint64_t	R W	Section 5.8.1.5
link_support	5	bool	R	Section 5.8.1.6
symlink_support	6	bool	R	Section 5.8.1.7
named_attr	7	bool	R	Section 5.8.1.8
fsid	8	fsid4	R	Section 5.8.1.9
unique_handles	9	bool	R	Section 5.8.1.10
lease_time	10	nfs_lease4	R	Section 5.8.1.11
rdattr_error	11	enum	R	Section 5.8.1.12
filehandle	19	nfs_fh4	R	Section 5.8.1.13
suppattr_exclcreat	75	bitmap4	R	Section 5.8.1.14

Table 2

5.7. RECOMMENDED Attributes - List and Definition References

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

Name	Id	Data Type	Acc	Defined in:
acl	12	nfsace4<>	R W	Section 6.2.1
aclsupport	13	uint32_t	R	Section 6.2.1.2
archive	14	bool	R W	Section 5.8.2.1
cansettime	15	bool	R	Section 5.8.2.2
case_insensitive	16	bool	R	Section 5.8.2.3
case_preserving	17	bool	R	Section 5.8.2.4

change_policy	60	chg_policy4	R	Section 5.8.2.5
chown_restricted	18	bool	R	Section 5.8.2.6
dacl	58	nfsacl41	R W	Section 6.2.2
dir_notif_delay	56	nfstime4	R	Section 5.11.1
dirent_notif_delay	57	nfstime4	R	Section 5.11.2
fileid	20	uint64_t	R	Section 5.8.2.7
files_avail	21	uint64_t	R	Section 5.8.2.8
files_free	22	uint64_t	R	Section 5.8.2.9
files_total	23	uint64_t	R	Section 5.8.2.10
fs_charset_cap	76	uint32_t	R	Section 5.8.2.11
fs_layout_type	62	layouttype4<>	R	Section 5.12.1
fs_locations	24	fs_locations	R	Section 5.8.2.12
fs_locations_info	67	*	R	Section 5.8.2.13
fs_status	61	fs4_status	R	Section 5.8.2.14
hidden	25	bool	R W	Section 5.8.2.15
homogeneous	26	bool	R	Section 5.8.2.16
layout_alignment	66	uint32_t	R	Section 5.12.2
layout_blksize	65	uint32_t	R	Section 5.12.3
layout_hint	63	layouthint4	W	Section 5.12.4
layout_type	64	layouttype4<>	R	Section 5.12.5
maxfilesize	27	uint64_t	R	Section 5.8.2.17
maxlink	28	uint32_t	R	Section 5.8.2.18
maxname	29	uint32_t	R	Section 5.8.2.19
maxread	30	uint64_t	R	Section 5.8.2.20
maxwrite	31	uint64_t	R	Section 5.8.2.21
mdsthreshold	68	mdsthreshold4	R	Section 5.12.6

	mimetype		32		utf8<>		R W		Section 5.8.2.22	
	mode		33		mode4		R W		Section 6.2.4	
	mode_set_masked		74		mode_masked4		W		Section 6.2.5	
	mounted_on_fileid		55		uint64_t		R		Section 5.8.2.23	
	no_trunc		34		bool		R		Section 5.8.2.24	
	numlinks		35		uint32_t		R		Section 5.8.2.25	
	owner		36		utf8<>		R W		Section 5.8.2.26	
	owner_group		37		utf8<>		R W		Section 5.8.2.27	
	quota_avail_hard		38		uint64_t		R		Section 5.8.2.28	
	quota_avail_soft		39		uint64_t		R		Section 5.8.2.29	
	quota_used		40		uint64_t		R		Section 5.8.2.30	
	rawdev		41		specdata4		R		Section 5.8.2.31	
	retentevt_get		71		retention_get4		R		Section 5.13.3	
	retentevt_set		72		retention_set4		W		Section 5.13.4	
	retention_get		69		retention_get4		R		Section 5.13.1	
	retention_hold		73		uint64_t		R W		Section 5.13.5	
	retention_set		70		retention_set4		W		Section 5.13.2	
	sacl		59		nfsacl41		R W		Section 6.2.3	
	space_avail		42		uint64_t		R		Section 5.8.2.32	
	space_free		43		uint64_t		R		Section 5.8.2.33	
	space_total		44		uint64_t		R		Section 5.8.2.34	
	space_used		45		uint64_t		R		Section 5.8.2.35	

	system		46		bool		R W		Section 5.8.2.36	
	time_access		47		nfstime4		R		Section 5.8.2.37	
	time_access_set		48		settime4		W		Section 5.8.2.38	
	time_backup		49		nfstime4		R W		Section 5.8.2.39	
	time_create		50		nfstime4		R W		Section 5.8.2.40	
	time_delta		51		nfstime4		R		Section 5.8.2.41	
	time_metadata		52		nfstime4		R		Section 5.8.2.42	
	time_modify		53		nfstime4		R		Section 5.8.2.43	
	time_modify_set		54		settime4		W		Section 5.8.2.44	

+-----+-----+-----+-----+-----+-----+

Table 3

* fs_locations_info4

[5.8.](#) Attribute Definitions

[5.8.1.](#) Definitions of REQUIRED Attributes

[5.8.1.1.](#) Attribute 0: supported_attrs

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

[5.8.1.2.](#) Attribute 1: type

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

- o NF4REG designates a regular file.
- o NF4DIR designates a directory.
- o NF4BLK designates a block device special file.
- o NF4CHR designates a character device special file.
- o NF4LNK designates a symbolic link.
- o NF4SOCK designates a named socket special file.
- o NF4FIFO designates a fifo special file.
- o NF4ATTRDIR designates a named attribute directory.
- o NF4NAMEDATTR designates a named attribute.

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

- o The phrase "is a directory" means that the object is of type NF4DIR or of type NF4ATTRDIR.
- o The phrase "is a special file" means that the object is of one of the types NF4BLK, NF4CHR, NF4SOCK, or NF4FIFO.
- o The phrase "is an ordinary file" means that the object is of type NF4REG or of type NF4NAMEDATTR.

[5.8.1.3.](#) Attribute 2: fh_expire_type

Server uses this to specify filehandle expiration behavior to the client. See [Section 4](#) for additional description.

[5.8.1.4.](#) Attribute 3: change

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

[5.8.1.5.](#) Attribute 4: size

The size of the object in bytes.

[5.8.1.6.](#) Attribute 5: link_support

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

[5.8.1.7.](#) Attribute 6: symlink_support

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

[5.8.1.8.](#) Attribute 7: named_attr

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

[5.8.1.9.](#) Attribute 8: fsid

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

[5.8.1.10.](#) Attribute 9: unique_handles

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

[5.8.1.11.](#) Attribute 10: lease_time

Duration of leases at server in seconds.

[5.8.1.12.](#) Attribute 11: rgetattr_error

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

[5.8.1.13.](#) Attribute 19: filehandle

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

[5.8.1.14.](#) Attribute 75: suppattr_exclcreat

The bit vector which would set all REQUIRED and RECOMMENDED attributes that are supported by the EXCLUSIVE4_1 method of file creation via the OPEN operation. The scope of this attribute applies to all objects with a matching fsid.

[5.8.2.](#) Definitions of Uncategorized RECOMMENDED Attributes

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

[5.8.2.1.](#) Attribute 14: archive

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

[5.8.2.2.](#) Attribute 15: cansettime

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

[5.8.2.3.](#) Attribute 16: case_insensitive

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

[5.8.2.4.](#) Attribute 17: case_preserving

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

[5.8.2.5.](#) Attribute 60: change_policy

A value created by the server that the client can use to determine if some server policy related to the current file system has been subject to change. If the value remains the same then the client can be sure that the values of the attributes related to fs location and the fss_type field of the fs_status attribute have not changed. On the other hand, a change in this value does necessarily imply a change in policy. It is up to the client to interrogate the server to determine if some policy relevant to it has changed. See [Section 3.3.6](#) for details.

This attribute MUST change when the value returned by the fs_locations or fs_locations_info attribute changes, when a file system goes from read-only to writable or vice versa, or when the allowable set of security flavors for the file system or any part thereof is changed.

[5.8.2.6.](#) Attribute 18: chown_restricted

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

[5.8.2.7.](#) Attribute 20: fileid

A number uniquely identifying the file within the file system.

[5.8.2.8.](#) Attribute 21: files_avail

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

[5.8.2.9.](#) Attribute 22: files_free

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

[5.8.2.10.](#) Attribute 23: files_total

Total file slots on the file system containing this object.

Internet-Draft

NFSv4.1

September 2008

[5.8.2.11](#). Attribute 76: fs_charset_cap

Character set capabilities for this file system. See [Section 14.4](#).

[5.8.2.12](#). Attribute 24: fs_locations

Locations where this file system may be found. If the server returns NFS4ERR_MOVED as an error, this attribute MUST be supported. See [Section 11.9](#) for more details.

[5.8.2.13](#). Attribute 67: fs_locations_info

Full function file system location. See [Section 11.10](#) for more details.

[5.8.2.14](#). Attribute 61: fs_status

Generic file system type information. See [Section 11.11](#) for more details.

[5.8.2.15](#). Attribute 25: hidden

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

[5.8.2.16](#). Attribute 26: homogeneous

True, if this object's file system is homogeneous, i.e. are per file system attributes the same for all file system's objects.

[5.8.2.17](#). Attribute 27: maxfilesize

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

[5.8.2.18](#). Attribute 28: maxlink

Maximum number of links for this object.

[5.8.2.19](#). Attribute 29: maxname

Maximum file name size supported for this object.

[5.8.2.20.](#) Attribute 30: maxread

Maximum read size supported for this object.

[5.8.2.21.](#) Attribute 31: maxwrite

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.

[5.8.2.22.](#) Attribute 32: mimetype

MIME body type/subtype of this object.

[5.8.2.23.](#) Attribute 55: mounted_on_fileid

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

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

Unlike NFSv3, NFSv4.1 allows a client's LOOKUP request to cross other file systems. The client detects the file system crossing whenever the filehandle argument of LOOKUP has an `fsid` attribute different from that of the filehandle returned by LOOKUP. A UNIX-based client will consider this a "mount point crossing". UNIX has a legacy scheme for allowing a process to determine its current working

directory. This relies on `readdir()` of a mount point's parent and `stat()` of the mount point returning fileids as previously described. The `mounted_on_fileid` attribute corresponds to the fileid that `readdir()` would have returned as described previously.

While the NFSv4.1 client could simply fabricate a fileid corresponding to what `mounted_on_fileid` provides (and if the server does not support `mounted_on_fileid`, the client has no choice), there is a risk that the client will generate a fileid that conflicts with one that is already assigned to another object in the file system. Instead, if the server can provide the `mounted_on_fileid`, the potential for client operational problems in this area is eliminated.

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

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

[5.8.2.24](#). Attribute 34: `no_trunc`

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

[5.8.2.25](#). Attribute 35: `numlinks`

Number of hard links to this object.

[5.8.2.26](#). Attribute 36: owner

The string name of the owner of this object.

[5.8.2.27](#). Attribute 37: owner_group

The string name of the group ownership of this object.

[5.8.2.28](#). Attribute 38: 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.

[5.8.2.29](#). Attribute 39: 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.

[5.8.2.30](#). Attribute 40: quota_used

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

Note that there may be a number of distinct but overlapping sets of files or directories for which a quota_used value is maintained. E.g. "all files with a given owner", "all files with a given group owner". etc.

The server is at liberty to choose any of those sets but should do so in a repeatable way. The rule may be configured per file system or may be "choose the set with the smallest quota".

[5.8.2.31](#). Attribute 41: rawdev

Raw device identifier; the UNIX device major/minor node information. If the value of type is not NF4BLK or NF4CHR, the value returned SHOULD NOT be considered useful.

[5.8.2.32](#). Attribute 42: space_avail

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

[5.8.2.33](#). Attribute 43: space_free

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

[5.8.2.34](#). Attribute 44: space_total

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

[5.8.2.35](#). Attribute 45: space_used

Number of file system bytes allocated to this object.

[5.8.2.36](#). Attribute 46: system

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

[5.8.2.37](#). Attribute 47: time_access

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.

[5.8.2.38](#). Attribute 48: `time_access_set`

Set the time of last access to the object. `SETATTR` use only.

[5.8.2.39](#). Attribute 49: `time_backup`

The time of last backup of the object.

[5.8.2.40](#). Attribute 50: `time_create`

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

[5.8.2.41](#). Attribute 51: `time_delta`

Smallest useful server time granularity.

[5.8.2.42](#). Attribute 52: `time_metadata`

The time of last metadata modification of the object.

[5.8.2.43](#). Attribute 53: `time_modify`

The time of last modification to the object.

[5.8.2.44](#). Attribute 54: time_modify_set

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

[5.9](#). Interpreting owner and owner_group

The RECOMMENDED attributes "owner" and "owner_group" (and also users and groups within the "acl" attribute) are represented in terms of a UTF-8 string. To avoid a representation that is tied to a particular underlying implementation at the client or server, the use of the UTF-8 string has been chosen. Note that [section 6.1 of RFC2624](#) [33] 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 identifier to the user@dns_domain syntax. A name service may also be used to accomplish the translation. A server may provide a more general service, not limited by any particular translation (which would only translate a limited set of possible strings) by storing the owner and owner_group attributes in local storage without any translation or it may augment a translation method by storing the entire string for attributes for which no

translation is available while using the local representation for those cases in which a translation is available.

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

The "dns_domain" portion of the owner string is meant to be a DNS domain name. For example, user@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 NFSv3, which identified users and groups by 32-bit unsigned user identifiers and group identifiers, 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 an NFSv3 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

Internet-Draft

NFSv4.1

September 2008

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.

[5.10.](#) 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](#) [34] 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 [Section 14](#).

[5.11.](#) Directory Notification Attributes

As described in [Section 18.39](#), 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.

[5.11.1.](#) Attribute 56: 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.

[5.11.2.](#) Attribute 57: 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.

5.12.1. Attribute 62: fs_layout_type

The fs_layout_type attribute (see [Section 3.3.13](#)) applies to a file system and indicates what layout types are supported by the file system. When the client encounters a new fsid, the client SHOULD obtain the value for the fs_layout_type attribute associated with the new file system. This attribute is used by the client to determine if the layout types supported by the server match any of the client's supported layout types.

5.12.2. Attribute 66: layout_alignment

When a client holds layouts on files of a file system, the layout_alignment attribute indicates the preferred alignment for I/O to files on that file system. Where possible, the client should send READ and WRITE operations with offsets that are whole multiples of the layout_alignment attribute.

5.12.3. Attribute 65: layout_blksize

When a client holds layouts on files of a file system, the layout_blksize attribute indicates the preferred block size for I/O to files on that file system. Where possible, the client should send 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.

5.12.4. Attribute 63: layout_hint

The layout_hint attribute (see [Section 3.3.19](#)) may be set on newly created files to influence the metadata server's choice for the file's layout. If possible, this attribute is one of those set in the initial attributes within the OPEN operation. The metadata server may choose to ignore this attribute. The layout_hint attribute is a sub-set 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. The server implementation determines which fields within the layout will be used.

[5.12.5.](#) Attribute 64: layout_type

This attribute lists the layout type(s) available for a file. The value returned by the server is for informational purposes only. The client will use the LAYOUTGET operation to obtain the information needed in order to perform I/O. For example, the specific device information for the file and its layout.

[5.12.6.](#) Attribute 68: mdsthreshold

This attribute is a server provided hint used to communicate to the client when it is more efficient to send READ and WRITE operations to the metadata server or the data server. The two types of thresholds described are file size thresholds and I/O size thresholds. If a file's size is smaller than the file size threshold, data accesses SHOULD be sent to the metadata server. If an I/O request has a length that is below the I/O size threshold, the I/O SHOULD be sent to the metadata server. Each threshold type is specified separately for READ and WRITE.

The server MAY provide both types of thresholds for a file. If both file size and I/O size are provided, the client SHOULD reach or exceed both thresholds before issuing its READ or WRITE requests to the data server. Alternatively, if only one of the specified thresholds are reached or exceeded, the I/O requests are sent to the metadata server.

For each threshold type, a value of 0 indicates no READ or WRITE should be sent to the metadata server, while a value of all 1s indicates all READS or WRITES should be sent 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.

[5.13.](#) 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 to any other property of the file, including any subset of **REQUIRED**, **RECOMMENDED**, and named attributes, with the exceptions noted in this section.

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

The five retention attributes are explained in the next subsections.

[5.13.1.](#) Attribute 69: retention_get

If retention is enabled for the associated file, this attribute's value represents the retention begin time of the file object. This attribute's value is only readable with the GETATTR operation and **MUST NOT** be modified by the SETATTR operation ([Section 5.5](#)). 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 the 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.

If (as soon as) `rg_duration` is zero, then `rg_begin_time` will be of zero length, and again, retention is not (no longer) enabled.

[5.13.2](#). Attribute 70: `retention_set`

This attribute is used to set the retention duration and optionally enable retention for the associated file object. This attribute is only modifiable via the `SETATTR` operation and MUST NOT be retrieved by the `GETATTR` operation ([Section 5.5](#)). This attribute corresponds to `retention_get`. 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 starting 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 the 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 MUST NOT prevent the enabling of event-based retention nor the modification of the `retention_hold` attribute.

The following rules apply to both the `retention_set` and `retentevt_set` attributes.

- o As long as retention is not enabled, the client is permitted to decrease the duration.
- o The duration can always be set to an equal or higher value, even if retention is enabled. Note that once retention is enabled, the actual duration (as returned by the `retention_get` or `retentevt_get` attributes, see [Section 5.13.1](#) or [Section 5.13.3](#)), is constantly

counting down to zero (one unit per second), unless the duration was set to RET4_DURATION_INFINITE. Thus it will not be possible for the client to precisely extend the duration on a file that has retention enabled.

- o While retention is enabled, attempts to disable retention or decrease the retention's duration MUST fail with the error NFS4ERR_INVALID.
- o If the principal attempting to change retention_set or retentevt_set does not have ACE4_WRITE_RETENTION permissions, the attempt MUST fail with NFS4ERR_ACCESS.

[5.13.3.](#) Attribute 71: retentevt_get

Get the event-based retention duration, and if enabled, the event-based retention begin time of the file object. 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.

[5.13.4.](#) Attribute 72: retentevt_set

Set the event-based retention duration, and optionally enable event-based retention on the file object. 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 if 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 MUST NOT prevent the enabling of non-event-based retention nor the modification of the retention_hold attribute.

[5.13.5.](#) Attribute 73: retention_hold

Get or set administrative retention holds, one hold per bit position.

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.

If the principal attempting to change `retention_hold` does not have `ACE4_WRITE_RETENTION_HOLD` permissions, the attempt MUST fail with `NFS4ERR_ACCESS`.

[6.](#) Access Control Attributes

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

[6.1.](#) Goals

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

- o If a server supports the mode attribute, it should provide reasonable semantics to clients that only set and retrieve the mode attribute.
- o If a server supports ACL attributes, it should provide reasonable semantics to clients that only set and retrieve those attributes.
- o On servers that support the mode attribute, if ACL attributes have never been set on an object, via inheritance or explicitly, the behavior should be traditional UNIX-like behavior.

- o On servers that support the mode attribute, if the ACL attributes

have been previously set on an object, either explicitly or via inheritance:

- * Setting only the mode attribute should effectively control the traditional UNIX-like permissions of read, write, and execute on owner, owner_group, and other.
- * Setting only the mode attribute should provide reasonable security. For example, setting a mode of 000 should be enough to ensure that future opens for read or write by any principal fail, regardless of a previously existing or inherited ACL.
- o NFSv4.1 may introduce different semantics relating to the mode and ACL attributes, but it does not render invalid any previously existing implementations. Additionally, this chapter provides clarifications based on previous implementations and discussions around them.
- o On servers that support both the mode and the acl or dacl attributes, the server must keep the two consistent with each other. The value of the mode attribute (with the exception of the three high order bits described in [Section 6.2.4](#)), must be determined entirely by the value of the ACL, so that use of the mode is never required for anything other than setting the three high order bits. See [Section 6.4.1](#) for exact requirements.
- o When a mode attribute is set on an object, the ACL attributes may need to be modified so as to not conflict with the new mode. In such cases, it is desirable that the ACL keep as much information as possible. This includes information about inheritance, AUDIT and ALARM ACEs, and permissions granted and denied that do not conflict with the new mode.

[6.2.](#) File Attributes Discussion

[6.2.1.](#) Attribute 12: acl

The NFSv4.1 ACL attribute contains an array of access control entries (ACEs) that are associated with the file system object. Although the client can read and write the acl attribute, the server is responsible for using the ACL to perform access control. The client can use the OPEN or ACCESS operations to check access without modifying or reading data or metadata.

The NFS ACE structure is defined as follows:

```
typedef uint32_t          acetype4;
```

```
typedef uint32_t aceflag4;

typedef uint32_t      acemask4;

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

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

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

The NFSv4.1 ACL model is quite rich. Some server platforms may provide access control functionality that goes beyond the UNIX-style mode attribute, but which is not as rich as the NFS ACL model. So that users can take advantage of this more limited functionality, the server may support the `acl` attributes by mapping between its ACL model and the NFSv4.1 ACL model. Servers must ensure that the ACL they actually store or enforce is at least as strict as the NFSv4 ACL that was set. It is tempting to accomplish this by rejecting any ACL that falls outside the small set that can be represented accurately. However, such an approach can render ACLs unusable without special client-side knowledge of the server's mapping, which defeats the purpose of having a common NFSv4 ACL protocol. Therefore servers should accept every ACL that they can without compromising security. To help accomplish this, servers may make a special exception, in the case of unsupported permission bits, to the rule that bits not ALLOWED or DENIED by an ACL must be denied. For example, a UNIX-

style server might choose to silently allow read attribute permissions even though an ACL does not explicitly allow those

permissions. (An ACL that explicitly denies permission to read attributes should still be rejected.)

The situation is complicated by the fact that a server may have multiple modules that enforce ACLs. For example, the enforcement for NFSv4.1 access may be different from, but not weaker than, the enforcement for local access, and both may be different from the enforcement for access through other protocols such as SMB. So it may be useful for a server to accept an ACL even if not all of its modules are able to support it.

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

6.2.1.1. ACE Type

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

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

Only the ALLOWED and DENIED bits types may be used in the dacl attribute, and only the AUDIT and ALARM bits may be used in the sacl attribute. All four are permitted in the acl attribute.

Value	Abbreviation	Description
ACE4_ACCESS_ALLOWED_ACE_TYPE	ALLOW	Explicitly grants the access defined in acemask4 to the file or directory.
ACE4_ACCESS_DENIED_ACE_TYPE	DENY	Explicitly denies the access defined in acemask4 to the file or directory.

ACE4_SYSTEM_AUDIT_ACE_TYPE	AUDIT	LOG (in a system dependent way) any access attempt to a file or directory which uses any of the access methods specified in acemask4.
----------------------------	-------	---

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

+-----+-----+-----+

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

[6.2.1.2](#). Attribute 13: aclsupport

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

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

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

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

NFS4ERR_ATTRNOTSUPP. If the server receives a request to set an ACE that it can store but cannot enforce, the server SHOULD reject the request with NFS4ERR_ATTRNOTSUPP.

Support for any of the ACL attributes is optional (albeit, RECOMMENDED). However, a server that supports either of the new ACL attributes (dacl or sacl) MUST allow use of the new ACL attributes to access all of the ACE types which it supports. In other words, if such a server supports ALLOW or DENY ACEs, then it MUST support the dacl attribute, and if it supports AUDIT or ALARM ACEs, then it MUST support the sacl attribute.

[6.2.1.3](#). ACE Access Mask

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

```
const ACE4_READ_DATA           = 0x00000001;
const ACE4_LIST_DIRECTORY      = 0x00000001;
const ACE4_WRITE_DATA          = 0x00000002;
const ACE4_ADD_FILE            = 0x00000002;
const ACE4_APPEND_DATA         = 0x00000004;
const ACE4_ADD_SUBDIRECTORY    = 0x00000004;
const ACE4_READ_NAMED_ATTRS    = 0x00000008;
const ACE4_WRITE_NAMED_ATTRS   = 0x00000010;
const ACE4_EXECUTE             = 0x00000020;
const ACE4_DELETE_CHILD        = 0x00000040;
const ACE4_READ_ATTRIBUTES     = 0x00000080;
const ACE4_WRITE_ATTRIBUTES     = 0x00000100;
const ACE4_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;
```

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

ACE4_WRITE_DATA, and ACE4_APPEND_DATA are intended to be used with non-directory objects.

[6.2.1.3.1](#). Discussion of Mask Attributes

ACE4_READ_DATA

Operation(s) affected:

READ

OPEN

Discussion:

Permission to read the data of the file.

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

ACE4_LIST_DIRECTORY

Operation(s) affected:

READDIR

Discussion:

Permission to list the contents of a directory.

ACE4_WRITE_DATA

Operation(s) affected:

WRITE

OPEN

SETATTR of size

Discussion:

Permission to modify a file's data.

ACE4_ADD_FILE

Operation(s) affected:

CREATE

LINK

OPEN

RENAME

Discussion:

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

ACE4_APPEND_DATA

Operation(s) affected:

WRITE

OPEN

SETATTR of size

Discussion:

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

ACE4_ADD_SUBDIRECTORY

Operation(s) affected:

CREATE

RENAME

Discussion:

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

ACE4_READ_NAMED_ATTRS

Operation(s) affected:

OPENATTR

Discussion:

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

ACE4_WRITE_NAMED_ATTRS

Operation(s) affected:

OPENATTR

Discussion:

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

ACE4_EXECUTE

Operation(s) affected:

READ

OPEN

REMOVE

RENAME

LINK

CREATE

Discussion:

Permission to execute a file.

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

as that would make it impossible for the client to correctly enforce the distinction between read and execute permissions.

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

nfsuser:ACE4_EXECUTE:ALLOW

A subsequent GETATTR of ACL for that file SHOULD return:

nfsuser:ACE4_EXECUTE:ALLOW

Rather than:

nfsuser:ACE4_EXECUTE/ACE4_READ_DATA:ALLOW

ACE4_EXECUTE

Operation(s) affected:

LOOKUP

Discussion:

Permission to traverse/search a directory.

ACE4_DELETE_CHILD

Operation(s) affected:

REMOVE

RENAME

Discussion:

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

ACE4_READ_ATTRIBUTES

Operation(s) affected:

GETATTR of file system object attributes

Internet-Draft

NFSv4.1

September 2008

VERIFY

NVERIFY

READDIR

Discussion:

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

ACE4_WRITE_ATTRIBUTES**Operation(s) affected:**

SETATTR of time_access_set, time_backup,
time_create, time_modify_set, mimetype, hidden, system

Discussion:

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

ACE4_WRITE_RETENTION**Operation(s) affected:**

SETATTR 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 behave such that setting ACE4_WRITE_ATTRIBUTES allows ACE4_WRITE_RETENTION.

ACE4_WRITE_RETENTION_HOLD

Operation(s) affected:

SETATTR 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 6.2.1.3.2](#) for information on ACE4_DELETE and ACE4_DELETE_CHILD interact.

ACE4_READ_ACL

Operation(s) affected:

GETATTR of acl, dacl, or sacl

NVERIFY

VERIFY

Discussion:

Permission to read the ACL.

ACE4_WRITE_ACL

Operation(s) affected:

SETATTR of acl and mode

Shepler, et al.

Expires March 9, 2009

[Page 134]

Internet-Draft

NFSv4.1

September 2008

Discussion:

Permission to write the acl and mode attributes.

ACE4_WRITE_OWNER

Operation(s) affected:

SETATTR of owner and owner_group

Discussion:

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

ACE4_SYNCHRONIZE

Operation(s) affected:

NONE

Discussion:

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

[6.2.1.3.2](#). ACE4_DELETE vs. ACE4_DELETE_CHILD

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

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

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

If the ACLs in question neither explicitly ALLOW nor DENY either of the above, and if MODE4_SVTX is not set on the parent, then the server SHOULD allow the removal if and only if ACE4_ADD_FILE is

permitted. In the case where MODE4_SVTX is set, the server may also require the remover to own either the parent or the target, or may require the target to be writable.

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

[6.2.1.4.](#) ACE flag

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

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

For example, suppose a client tries to set an ACE with ACE4_FILE_INHERIT_ACE set but not ACE4_DIRECTORY_INHERIT_ACE. If the server does not support any form of ACL inheritance, the server should reject the request with NFS4ERR_ATTRNOTSUPP. If the server

supports a single "inherit ACE" flag that applies to both files and directories, the server may reject the request (i.e., requiring the client to set both the file and directory inheritance flags). The server may also accept the request and silently turn on the ACE4_DIRECTORY_INHERIT_ACE flag.

[6.2.1.4.1.](#) Discussion of Flag Bits

ACE4_FILE_INHERIT_ACE

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

ACE4_DIRECTORY_INHERIT_ACE

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

If this flag is set in an ACE in an ACL attribute to be set on a non-directory file system object, the operation attempting to set the ACL SHOULD fail with NFS4ERR_ATTRNOTSUPP.

ACE4_INHERIT_ONLY_ACE

Can be placed on a directory but does not apply to the directory; ALLOW and DENY ACEs with this bit set do not affect access to the directory, and AUDIT and ALARM ACEs with this bit set do not trigger log or alarm events. Such ACEs only take effect once they are applied (with this bit cleared) to newly created files and directories as specified by the above two flags.

If this flag is present on an ACE, but neither ACE4_DIRECTORY_INHERIT_ACE nor ACE4_FILE_INHERIT_ACE is present, then an operation attempting to set such an attribute SHOULD fail with NFS4ERR_ATTRNOTSUPP.

ACE4_NO_PROPAGATE_INHERIT_ACE

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

ACE4_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 may be set only on ACE4_SYSTEM_AUDIT_ACE_TYPE (AUDIT) and ACE4_SYSTEM_ALARM_ACE_TYPE (ALARM) ACE types. If during the processing of the file's ACL,

the server encounters an AUDIT or ALARM ACE that matches the principal attempting the OPEN, the server notes that fact, and the presence, if any, of the SUCCESS and FAILED flags encountered in the AUDIT or ALARM ACE. Once the server completes the ACL processing, it then notes if the operation succeeded or failed. If the operation succeeded, and if the SUCCESS flag was set for a matching AUDIT or ALARM ACE, then the appropriate AUDIT or ALARM event occurs. If the operation failed, and if the FAILED flag was set for the matching AUDIT or ALARM ACE, then the appropriate AUDIT or ALARM event occurs. Either or both of the SUCCESS or FAILED can be set, but if neither is set, the AUDIT or ALARM ACE is not useful.

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

ACE4_IDENTIFIER_GROUP

Indicates that the "who" refers to a GROUP as defined under UNIX or a GROUP ACCOUNT as defined under Windows. Clients and servers MUST ignore the ACE4_IDENTIFIER_GROUP flag on ACEs with a who value equal to one of the special identifiers outlined in [Section 6.2.1.5](#).

[6.2.1.5](#). ACE Who

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

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

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

Table 4

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

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

[6.2.1.5.1.](#) Discussion of EVERYONE@

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

[6.2.2.](#) Attribute 58: dacl

The dacl attribute is like the acl attribute, but dacl allows just ALLOW and DENY ACEs. The dacl attribute supports automatic inheritance (see [Section 6.4.3.2](#)).

[6.2.3.](#) Attribute 59: sacl

The sacl attribute is like the acl attribute, but sacl allows just AUDIT and ALARM ACEs. The sacl attribute supports automatic inheritance (see [Section 6.4.3.2](#)).

[6.2.4.](#) Attribute 33: mode

The NFSv4.1 mode attribute is based on the UNIX mode bits. The following bits are defined:

Internet-Draft

NFSv4.1

September 2008

```
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_OTH = 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_OTH, MODE4_WOTH, MODE4_XOTH apply to any principal that does not match that in the owner attribute, and does not have a group matching that of the owner_group attribute.

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

[6.2.5](#). Attribute 74: mode_set_masked

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.

In such instances that the nine low-order bits are left unmodified, then neither the acl nor the dacl attribute should be automatically modified as discussed in [Section 6.4.1](#).

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 NFS4ERR_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:

- o In the case of a file system exported as read-only, the server may deny write permissions even though an object's ACL grants it.
- o Server implementations MAY grant ACE4_WRITE_ACL and ACE4_READ_ACL permissions to prevent a situation from arising in which there is no valid way to ever modify the ACL.
- o All servers will allow a user the ability to read the data of the file when only the execute permission is granted (i.e. If the ACL denies the user the ACE4_READ_DATA access and allows the user ACE4_EXECUTE, the server will allow the user to read the data of the file).

- o Many servers have the notion of owner-override in which the owner of the object is allowed to override accesses that are denied by the ACL. This may be helpful, for example, to allow users continued access to open files on which the permissions have changed.
- o Many servers have the notion of a "superuser" that has privileges beyond an ordinary user. The superuser may be able to read or write data or metadata in ways that would not be permitted by the ACL.

[6.3.1.2](#). Client Considerations

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

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

[6.3.2](#). Computing a Mode Attribute from an ACL

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

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

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

1. Set the read bit (MODE4_RUSR, MODE4_RGRP, or MODE4_ROTH) if and only if ACE4_READ_DATA is set in the corresponding mask.
2. Set the write bit (MODE4_WUSR, MODE4_WGRP, or MODE4_WOTH) if and only if ACE4_WRITE_DATA and ACE4_APPEND_DATA are both set in the corresponding mask.
3. Set the execute bit (MODE4_XUSR, MODE4_XGRP, or MODE4_XOTH), if and only if ACE4_EXECUTE is set in the corresponding mask.

[6.3.2.1](#). Discussion

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

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

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

[6.4](#). Requirements

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

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

In the case where a server supports the `sacl` or `dacl` attribute, in addition to the `acl` attribute, the server **MUST** fail a request to set the `acl` attribute simultaneously with a `dacl` or `sacl` attribute. The error to be given is `NFS4ERR_ATTRNOTSUPP`.

[6.4.1.1.](#) Setting mode and not ACL

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

Note that any AUDIT or ALARM ACEs (hence any ACEs in the `sacl` attribute) are unaffected by changes to the mode.

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

1. If `MODE4_RGRP` is not set, entities explicitly listed in the ACL other than `OWNER@` and `EVERYONE@` **SHOULD NOT** be granted `ACE4_READ_DATA`.
2. If `MODE4_WGRP` is not set, entities explicitly listed in the ACL other than `OWNER@` and `EVERYONE@` **SHOULD NOT** be granted `ACE4_WRITE_DATA` or `ACE4_APPEND_DATA`.

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

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

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

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

[6.4.1.2](#). Setting ACL and not mode

When setting the `acl` or `dacl` 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` or `dacl` attributes 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 ACL attributes.

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

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

[6.4.3](#). Creating New Objects

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

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 the mode is given in the call:

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

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

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

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

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

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

In the case where an object is being created without any initial attributes at all, e.g. an OPEN operation with an opentype4 of OPEN4_CREATE and a createmode4 of EXCLUSIVE4, inheritance SHOULD NOT take place (note that EXCLUSIVE4_1 is a better choice of createmode4, since it does permit initial attributes). Instead, the server SHOULD set permissions to deny all access to the newly created object. It is expected that the appropriate client will set the desired attributes in a subsequent SETATTR operation, and the server SHOULD allow that operation to succeed, regardless of what permissions the object is created with. For example, an empty ACL denies all permissions, but the server should allow the owner's SETATTR to succeed even though WRITE_ACL is implicitly denied.

In other cases, inheritance SHOULD take place, and no modifications to the ACL will happen. The mode attribute, if supported, MUST be as computed in [Section 6.3.2](#), with the MODE4_SUID, MODE4_SGID and MODE4_SVTX bits clear. If no inheritable ACEs exist on the parent directory, the rules for creating acl, dacl or sacl attributes are implementation defined. If either the dacl or sacl attribute is supported, then the ACL4_DEFAULTED flag SHOULD be set on the newly created attributes.

[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

Internet-Draft

NFSv4.1

September 2008

have ACE4_FILE_INHERIT_ACE or ACE4_DIRECTORY_INHERIT_ACE flag set. If the inheritable ACE has ACE4_FILE_INHERIT_ACE set, but ACE4_DIRECTORY_INHERIT_ACE is clear, the inherited ACE on the newly created directory MUST have the ACE4_INHERIT_ONLY_ACE flag set to prevent the directory from being affected by ACEs meant for non-directories.

When a new directory is created, the server MAY split any inherited ACE which is both inheritable and effective (in other words, which has neither ACE4_INHERIT_ONLY_ACE nor ACE4_NO_PROPAGATE_INHERIT_ACE set), into two ACEs, one with no inheritance flags, and one with ACE4_INHERIT_ONLY_ACE set. (In the case of a dacl or sacl attribute, both of those ACEs SHOULD also have the ACE4_INHERITED_ACE flag set.) This makes it simpler to modify the effective permissions on the directory without modifying the ACE which is to be inherited to the new directory's children.

[6.4.3.2](#). Automatic Inheritance

The acl attribute consists only of an array of ACEs, but the sacl ([Section 6.2.3](#)) and dacl ([Section 6.2.2](#)) attributes also include an additional flag field.

```
struct nfsacl41 {
    aclflag4      na41_flag;
    nfsace4       na41_aces<>;
};
```

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 descendants.

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, and SHOULD set 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 descendants 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 descendants.)

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 descendants.

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 Namespace

This chapter describes the NFSv4 single-server namespace. 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 11](#).

[7.1.](#) Server Exports

On a UNIX server, the namespace describes all the files reachable by pathnames under the root directory or "/". On a Windows server the namespace constitutes all the files on disks named by mapped disk letters. NFS server administrators rarely make the entire server's file system namespace available to NFS clients. More often portions of the namespace 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 sent a string that identified the export name within the namespace and the server returned the root filehandle for that export. The MOUNT protocol also provided an EXPORTS procedure that enumerated server's exports.

[7.2.](#) Browsing Exports

The NFSv4.1 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 NFSv3 protocol. In NFSv3, 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 namespace paths that span exports.

In the case of NFSv3, an automounter on the client can obtain a snapshot of the server's namespace 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 namespace on the client. The parts of the namespace that are not exported by the server are filled in with directories that might be constructed similarly to an NFSv4.1 "pseudo file system" (see [Section 7.3](#)) that allows the user to browse from one mounted file system to another. There is a drawback to this representation of the server's namespace 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

NFSv4.1 servers avoid this namespace inconsistency by presenting all the exports for a given server within the framework of a single namespace, for that server. An NFSv4.1 client uses LOOKUP and REaddir operations to browse seamlessly from one export to another.

Where there are portions of the server namespace that are not exported, clients require some way of traversing those portions to reach actual exported file systems. A technique that servers may use to provide for this is to bridge unexported portion of the namespace 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 namespace, 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 is considered a separate entity and therefore MUST have its own fsid, unique among all the fsids for that server.

[7.4.](#) Multiple Roots

Certain operating environments are sometimes described as having "multiple roots". In such environments individual file systems are commonly represented by disk or volume names. NFSv4 servers for these platforms can construct a pseudo file system above these root names so that disk letters or volume names are simply directory names in the pseudo root.

[7.5.](#) Filehandle Volatility

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

Because it is quite likely that servers will implement pseudo file systems using volatile filehandles, clients need to be prepared for them, rather than assuming that all filehandles will be persistent.

[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:

Shepler, et al.

Expires March 9, 2009

[Page 151]

Internet-Draft

NFSv4.1

September 2008

/	fs1	(exported)
/a	fs2	(not exported)
/a/b	fs3	(exported)

Because fs2 is not exported, fs3 cannot be reached with simple LOOKUPS. The server must bridge the gap with a pseudo file system.

[7.7.](#) Mount Point Crossing

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

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

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

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

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

Because NFSv4 clients possess the ability to change the security mechanisms used, after determining what is allowed, by using SECINFO and SECINFO_NONAME, the server SHOULD NOT present a different view of the namespace based on the security mechanism being used by a client. Instead, it should present a consistent view and return NFS4ERR_WRONGSEC if an attempt is made to access data with an inappropriate security mechanism.

If security considerations make it necessary to hide the existence of a particular file system, as opposed to all of the data within it, the server can apply the security policy of a shared resource in the server's namespace to components of the resource's ancestors. For example:

/a/b	(file system 1)
/a/b/MySecretProject	(file system 2)

The /a/b/MySecretProject directory is a real file system and is the shared resource. Suppose the security policy for /a/b/MySecretProject is Kerberos with integrity and it is desired to limit knowledge of the existence of this file system. In this case, the server should apply the same security policy to /a/b. This allows for knowledge of the existence of a file system to be secured when desirable.

For the case of the use of multiple, disjoint security mechanisms in the server's resources, applying that sort of policy would result in the higher-level file system not being accessible using any security flavor, which would make the that higher-level file system inaccessible. Therefore, that sort of configuration is not compatible with hiding the existence (as opposed to the contents) from clients using multiple disjoint sets of security flavors.

In other circumstances, a desirable policy is for the security of a particular object in the server's namespace should include the union of all security mechanisms of all direct descendants. A common and convenient practice, unless strong security requirements dictate otherwise, is to make all of the pseudo file system accessible by all of the valid security mechanisms.

Where there is concern about the security of data on the network, clients should use strong security mechanisms to access the pseudo file system in order to prevent man-in-the-middle attacks.

8. State Management

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 byte-range locking, the protocol becomes substantially more dependent on proper management of state than the traditional combination of NFS and NLM [35]. These features include expanded locking facilities, which provide some measure of interclient exclusion, but the state also offers features not readily providable using a stateless model. There are three components to making this state manageable:

- o Clear division between client and server

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

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

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

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

[8.1.](#) 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, delegate, or obtain a layout for a file object. Each session ID is associated with a specific client ID, and thus serves as a shorthand reference to an NFSv4.1 client.

For some types of locking interactions, the client will represent some number of internal locking entities called "owners", which normally correspond to processes internal to the client. For other types of locking-related objects, such as delegations and layouts, no such intermediate entities are provided for, and the locking-related objects are considered to be transferred directly between the server and a unitary client.

[8.2.](#) Stateid Definition

When the server grants a lock of any type (including opens, byte-range 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

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

All stateids associated with a given client ID are associated with a common lease which represents the claim of those stateids and the objects they represent to be maintained by the server. See [Section 8.3](#) for a discussion of leases.

The server may assign stateids independently for different clients. 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.

[8.2.1](#). Stateid Types

With the exception of special stateids (see [Section 8.2.3](#)), each stateid represents locking objects of one of a set of types defined by the NFSv4.1 protocol. Note that in all these cases, where we speak of guarantee, it is understood there are situations such as a client restart, or lock revocation, that allow the guarantee to be voided.

- o Stateids may represent opens of files.

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

- o Stateids may represent sets of byte-range locks.

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

- o Stateids may represent file delegations, which are recallable guarantees by the server to the client, that other clients will

not reference, or will not modify a particular file, until the delegation is returned. In NFSv4.1, file delegations may be obtained on both regular and non-regular files.

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

- o Stateids may represent directory delegations, which are recallable guarantees by the server to the client, that other clients will not modify the directory, until the delegation is returned.

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

- o Stateids may represent layouts, which are recallable guarantees by the server to the client, that particular files may be accessed via an alternate data access protocol at specific locations. Such access is limited to particular sets of byte ranges and may proceed until those byte ranges are reduced or the layout is returned.

A stateid represents the set of all layouts held by a particular client for a particular filehandle with a given layout type. The seqid is updated as the layouts of that set changes with layout stateid changing operations such as LAYOUTGET and LAYOUTRETURN.

[8.2.2.](#) Stateid Structure

Stateids are divided into two fields, a 96-bit "other" field identifying the specific set of locks and a 32-bit "seqid" sequence value. Except in the case of special stateids (see [Section 8.2.3](#)), a particular value of the "other" field denotes a set of locks of the

same type (for example byte-range locks, opens, delegations, or layouts), for a specific file or directory, and sharing the same ownership characteristics. The seqid designates a specific instance of such a set of locks, and is incremented to indicate changes in such a set of locks, either by the addition or deletion of locks from the set, a change in the byte-range they apply to, or an upgrade or downgrade in the type of one or more locks.

When such a set of locks is first created the server returns a stateid with seqid value of one. On subsequent operations which modify the set of locks the server is required to increment the seqid field by one (1) whenever it returns a stateid for the same state-owner/file/type combination and there is some change in the set of locks actually designated. In this case the server will return a stateid with an other field the same as previously used for that state-owner/file/type combination, with an incremented seqid field.

This pattern continues until the seqid is incremented past NFS4_UINT32_MAX, and one (not zero) is the next seqid value.

The purpose of the incrementing of the seqid is to allow the server to communicate to the client the order in which operations that modified locking state associated with a stateid have been processed and to make it possible for the client to send requests that are conditional on the set of locks not having changed since the stateid in question was returned.

Except for layout stateids ([Section 12.5.3](#)) when a client sends a stateid to the server, it has two choices with regard to the seqid sent. It may set the seqid to zero to indicate to the server that it wishes the most up-to-date seqid for that stateid's "other" field to be used. This would be the common choice in the case of a stateid sent with a READ or WRITE operation. It also may set a non-zero value in which case the server checks if that seqid is the correct one. In that case the server is required to return NFS4ERR_OLD_STATEID if the seqid is lower than the most current value and NFS4ERR_BAD_STATEID if the seqid is greater than the most current value. This would be the common choice in the case of stateids sent with a CLOSE or OPEN_DOWNGRADE. Because OPENS may be sent in parallel for the same owner, a client might close a file without knowing that an OPEN upgrade had been done by the server, changing the lock in question. If CLOSE were sent with a zero seqid, the OPEN

upgrade would be canceled before the client even received an indication that an upgrade had happened.

When a stateid is sent by the server to client as part of a callback operation, it is not subject to checking for a current seqid and returning NFS4ERR_OLD_STATEID. This is because the client is not in a position to know the most up-to-date seqid and thus cannot verify it. Unless specially noted, the seqid value for a stateid sent by the server to the client as part of a callback is required to be zero with NFS4ERR_BAD_STATEID returned if it is not.

In making comparisons between seqids, both by the client in determining the order of operations and by the server in determining whether the NFS4ERR_OLD_STATEID is to be returned, the possibility of the seqid being swapped around past the NFS4_UINT32_MAX value needs to be taken into account. When two seqid values are being compared, the total count of slots for all sessions associated with the current client is used to do this. When one seqid value is less than this total slot count and another seqid value is greater than NFS4_UINT32_MAX minus the total slot count, the former is to be treated as lower than the later, despite the fact that it is numerically greater.

[8.2.3.](#) Special Stateids

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

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

- o When "other" and "seqid" are both zero, the stateid is treated as a special anonymous stateid, which can be used in READ, WRITE, and SETATTR requests to indicate the absence of any open state associated with the request. When an anonymous stateid value is used, and an existing open denies the form of access requested, then access will be denied to the request. This stateid MUST NOT be used on operations to data servers ([Section 13.6](#)).

- o When "other" and "seqid" are both all ones, the stateid is a special read bypass stateid. When this value is used in WRITE or SETATTR, it is treated like the anonymous value. When used in READ, the server MAY grant access, even if access would normally be denied to READ requests. This stateid MUST NOT be used on operations to data servers.
- o 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, except when the current stateid is used by the operation CLOSE or OPEN_DOWNGRADE. If there is no operation in the COMPOUND which has returned a stateid value, the server MUST return the error NFS4ERR_BAD_STATEID. As illustrated in Figure 6, if the value of a current stateid is a special stateid, and the stateid of an operation's arguments has "other" set to zero, and "seqid" set to one, then the server MUST return the error NFS4ERR_BAD_STATEID.
- o When "other" is zero and "seqid" is NFS4_UINT32_MAX, the stateid represents a reserved stateid value defined to be invalid. When this stateid is used, 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 IDs or filehandles and can be used with all valid client IDs and filehandles. In the case of a special stateid designating the current stateid, the current stateid value substituted for the special stateid is associated with a particular client ID and filehandle, and so, if it is used where current filehandle does not match that associated with the current stateid, the operation to which the stateid is passed will return NFS4ERR_BAD_STATEID.

[8.2.4.](#) Stateid Lifetime and Validation

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

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

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

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

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

And then store in each table entry,

- o The client ID with which the stateid is associated.
- o The current generation number for the (at most one) valid stateid sharing this index value.

- o The filehandle of the file on which the locks are taken.
- o An indication of the type of stateid (open, byte-range lock, file delegation, directory delegation, layout).
- o The last "seqid" value returned corresponding to the current "other" value.
- o An indication of the current status of the locks associated with this stateid. In particular, whether these have been revoked and if so, for what reason.

With this information, an incoming stateid can be validated and the appropriate error returned when necessary. Special and non-special stateids are handled separately. (See [Section 8.2.3](#) for a discussion of special stateids.)

Note that stateids are implicitly qualified by the current client ID, as derived from the client ID associated with the current session. Note however, that the semantics of the session will prevent stateids associated with a previous client or server instance from being analyzed by this procedure.

If server restart has resulted in an invalid client ID or a session ID which is invalid, SEQUENCE will return an error and the operation that takes a stateid as an argument will never be processed.

If there has been a server restart where there is a persistent session, and all leased state has been lost, then the session in question will, although valid, be marked as dead, and any operation not satisfied by means of the reply cache will receive the error NFS4ERR_DEADSESSION, and thus not be processed as indicated below.

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

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

- o If the special stateid is one designating the current stateid, and there is a current stateid, then the current stateid is substituted for the special stateid and the checks appropriate to non-special stateids is performed.
- o If the combination is valid in general but is not appropriate to the context in which the stateid is used (e.g. an all-zero stateid is used when an open stateid is required in a LOCK operation), the error NFS4ERR_BAD_STATEID is also returned.
- o Otherwise, the check is completed and the special stateid is accepted as valid.

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

- o If the table index field is outside the range of the associated table, return NFS4ERR_BAD_STATEID.
- o If the selected table entry is of a different generation than that specified in the incoming stateid, return NFS4ERR_BAD_STATEID.
- o If the selected table entry does not match the current filehandle, return NFS4ERR_BAD_STATEID.
- o If the client ID in the table entry does not match the client ID associated with the current session, return NFS4ERR_BAD_STATEID.
- o If the stateid represents revoked state, then return NFS4ERR_EXPIRED, NFS4ERR_ADMIN_REVOKED, or NFS4ERR_DELEG_REVOKED, as appropriate.
- o If the stateid type is not valid for the context in which the stateid appears, return NFS4ERR_BAD_STATEID. Note that a stateid may be valid in general, as would be reported by the TEST_STATEID operation, but be invalid for a particular operation, as, for example, when a stateid which doesn't represent byte-range locks is passed to the non-from_open case of LOCK or to LOCKU, or when a stateid which does not represent an open is passed to CLOSE or OPEN_DOWNGRADE. In such cases, the server MUST return NFS4ERR_BAD_STATEID.
- o If the "seqid" field is not zero, and it is greater than the current sequence value corresponding the current "other" field,

return NFS4ERR_BAD_STATEID.

- o If the "seqid" field is not zero, and it is less than the current sequence value corresponding the current "other" field, return NFS4ERR_OLD_STATEID.
- o Otherwise, the stateid is valid and the table entry should contain any additional information about the type of stateid and information associated with that particular type of stateid, such as the associated set of locks, such as open-owner and lock-owner information, as well as information on the specific locks, such as open modes and byte ranges.

[8.2.5.](#) Stateid Use for I/O Operations

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

The following rules, applied in order of decreasing priority, govern the selection of the appropriate stateid. In following these rules, the client will only consider locks of which it has actually received notification by an appropriate operation response or callback. Note that the rules are slightly different in the case of I/O to data servers when file layouts are being used (see [Section 13.9.1](#)).

- o If the client holds a delegation for the file in question, the delegation stateid SHOULD be used.
- o Otherwise, if the lock-owner corresponding entity (e.g. process) issuing the I/O has a lock stateid for the associated open file, then the lock stateid for that lock-owner and open file SHOULD be used.
- o If there is no lock stateid, then the open stateid for the open file in question SHOULD be used.
- o Finally, if none of the above apply, then a special stateid SHOULD be used.

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

The server however should not try to enforce these ordering rules and should use whatever information is available to properly process I/O

requests. In particular, when a client has a delegation for a given file, it SHOULD take note of this fact in processing a request, even if it is sent with a special stateid.

[8.2.6.](#) Stateid Use for SETATTR Operations

Because each operation is associated with a session ID and from that the clientid can be determined, operations do not need to include a stateid for the server to be able to determine whether they should cause a delegation to be recalled or are to be treated as done within the scope of the delegation.

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

[8.3.](#) Lease Renewal

Each client/server pair, as represented by a client ID, has a single lease. The purpose of the lease is to allow the client to indicate to the server, in a low-overhead way, that it is active, and thus that the server is to retain the client's locks. This arrangement allows the server to remove stale locking-related objects that are held by a client that has crashed or is otherwise unreachable, once the relevant lease expires. This in turn allows other clients to obtain conflicting locks without being delayed indefinitely by inactive or unreachable clients. 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 (identified by the client's client ID), any operation sent on that session is an indication that the associated client is reachable. When a request is sent for a given session, successful execution of a SEQUENCE operation (or successful retrieval of the result of SEQUENCE from the reply cache) on an unexpired lease will result in the lease being implicitly renewed, for the standard renewal period (equal to the lease_time attribute).

If the client ID's lease has not expired when the server receives a SEQUENCE operation, then the server MUST renew the lease. If the client ID's lease has expired when the server receives a SEQUENCE operation, the server MAY renew the lease; this depends on whether

any state was revoked as a result of the client's failure to renew the lease before expiration.

Absent other activity that would renew the lease, a COMPOUND consisting of a single SEQUENCE operation will suffice. The client should also take communication-related delays into account and take steps to ensure that the renewal messages actually reach the server in good time. For example:

- o When trunking is in effect, the client should consider issuing multiple requests on different connections, in order to ensure that renewal occurs, even in the event of blockage in the path used for one of those connections.
- o Transport retransmission delays might become so large as to approach or exceed the length of the lease period. This may be particularly likely when the server is unresponsive due to a restart; see [Section 8.4.2.1](#). If the client implementation is not careful, transport retransmission delays can result in the client failing to detect a server restart before the grace period ends. The scenario is that the client is using a transport with exponential back off, such that the maximum retransmission timeout exceeds the both the grace period and the lease_time attribute. A network partition causes the client's connection's retransmission interval to back off, and even after the partition heals, the next transport-level retransmission is sent after the server has

restarted and its grace period ends.

The client MUST either recover from the ensuing NFS4ERR_NO_GRACE errors, or it MUST ensure that despite transport level retransmission intervals that exceed the lease_time, nonetheless a SEQUENCE operation is sent that renews the lease before expiration. The client can achieve this by associating a new connection with the session, and sending a SEQUENCE operation on it. However, if the attempt to establish a new connection is delayed for some reason (e.g. exponential backoff of the connection establishment packets), the client will have to abort the connection establishment attempt before the lease expires, and attempt to re-connect.

If the server renews the lease upon receiving a SEQUENCE operation, the server MUST NOT allow the lease to expire while the rest of the operations in the COMPOUND procedure's request are still executing. Once the last operation has finished, and the response to COMPOUND has been sent, the server MUST set the lease to expire no sooner than the sum of current time and the value of the lease_time attribute.

A client ID's lease can expire when it has been at least the lease

interval (lease_time) since the last lease-renewing SEQUENCE operation was sent on any of the client ID's sessions and there are no active COMPOUND operations on any such sessions.

Because the SEQUENCE operation is the basic mechanism to renew a lease, and because it must be done at least once for each lease period, it is the natural mechanism whereby the server will inform the client of changes in the lease status that the client needs to be informed of. The client should inspect the status flags (sr_status_flags) returned by sequence and take the appropriate action (see [Section 18.46.3](#) for details).

- o The status bits SEQ4_STATUS_CB_PATH_DOWN and SEQ4_STATUS_CB_PATH_DOWN_SESSION indicate problems with the backchannel which the client may need to address in order to receive callback requests.
- o The status bits SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRING and SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRED indicate problems with GSS

contexts for the backchannel which the client may have to address to allow callback requests to be sent to it.

- o The status bits SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED, SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED, SEQ4_STATUS_ADMIN_STATE_REVOKED, and SEQ4_STATUS_RECALLABLE_STATE_REVOKED notify the client of lock revocation events. When these bits are set, the client should use TEST_STATEID to find what stateids have been revoked and use FREE_STATEID to acknowledge loss of the associated state.
- o The status bit SEQ4_STATUS_LEASE_MOVE indicates that responsibility for lease renewal has been transferred to one or more new servers.
- o The status bit SEQ4_STATUS_RESTART_RECLAIM_NEEDED indicates that due to server restart the client must reclaim locking state.
- o The status bit SEQ4_STATUS_BACKCHANNEL_FAULT indicates the server has encountered an unrecoverable fault with the backchannel (e.g. it has lost track of a sequence ID for a slot in the backchannel).

[8.4.](#) Crash Recovery

A critical 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. 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. Any that reach the server before the server can safely determine that the client has recovered enough locking state to be sure that such operations can be safely processed must be rejected. This will happen because either:

- o The state presented is no longer valid since it is associated with a now invalid client ID. In this case the client will receive either an NFS4ERR_BADSESSION or NFS4ERR_DEADSESSION error, and any attempt to attach a new session to that invalid client ID will result in an NFS4ERR_STALE_CLIENTID error.

- o Subsequent recovery of locks may make execution of the operation inappropriate (NFS4ERR_GRACE).

[8.4.1.](#) Client Failure and Recovery

In the event that a client fails, the server may release the client's locks when the associated lease has expired. Conflicting locks from another client may only be granted after this lease expiration. As discussed in [Section 8.3](#), when a client has not failed and re-establishes its 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 `client_owner4` sent in 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 (see [Section 18.36.3](#) for a description how this is done). All locks, including opens, byte-range locks, delegations, and layouts 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 (upon confirmation of the new client ID) 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 lease had not yet expired, can be granted. In addition, all stateids associated with the old client ID can also be freed, as they are no longer reference-able.

Note that the verifier must have the same uniqueness properties as

the verifier for the `COMMIT` operation.

[8.4.2.](#) Server Failure and Recovery

If the server loses locking state (usually as a result of a restart),

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, and that such locking state might make it invalid to perform READ or WRITE operations, for example through the establishment of mandatory locks, 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 (most common) or other operation returns NFS4ERR_BADSESSION, this may mean the session has been destroyed, but the client ID is still valid. The client sends 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](#)) and re-establish its lock state with the new client ID, after the CREATE_SESSION operation succeeds (see [Section 8.4.2.1](#)).
2. When a SEQUENCE (most common) or other operation on a persistent session returns NFS4ERR_DEADSESSION, this indicates that a session is no longer usable for new, i.e. not satisfied from the reply cache, operations. Once all pending operations are determined to be either performed before the retry or not performed, the client sends 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](#)) and re-establish its lock state after the CREATE_SESSION, with the new client ID, succeeds, ([Section 8.4.2.1](#)).
3. 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](#)) and re-establish its lock state ([Section 8.4.2.1](#)).

8.4.2.1. State Reclaim

When state information and the associated locks are lost as a result of a server restart, the protocol must provide a way to cause that state to be re-established. The approach used is to define, for most types of locking state (layouts are an exception), a request whose function is to allow the client to re-establish on the server a lock first obtained from a previous instance. Generally these requests are variants of the requests normally used to create locks of that type and are referred to as "reclaim-type" requests and the process of re-establishing such locks is referred to as "reclaiming" them.

Because each client must have an opportunity to reclaim all of the locks that it has without the possibility that some other client will be granted a conflicting lock, a special period called the "grace period" is devoted to the reclaim process. During this period, requests creating client IDs and sessions are handled normally, but locking requests are subject to special restrictions. Only reclaim-type locking requests are allowed, unless the server can reliably determine (through state persistently maintained across restart instances), that granting any such lock cannot possibly conflict with a subsequent reclaim. When a request is made to obtain a new lock (i.e. not a reclaim-type request) during the grace period and such a determination cannot be made, the server must return the error NFS4ERR_GRACE.

Once a session is established using the new client ID, the client will use reclaim-type locking requests (e.g. LOCK requests with reclaim set to TRUE and OPEN operations with a claim type of CLAIM_PREVIOUS; see [Section 9.11](#)) to re-establish its locking state. Once this is done, or if there is no such locking state to reclaim, the client sends a global RECLAIM_COMPLETE operation, i.e. one with the rca_one_fs argument set to FALSE, to indicate that it has reclaimed all of the locking state that it will reclaim. Once a client sends such a RECLAIM_COMPLETE operation, it may attempt non-reclaim locking operations, although it may get NFS4ERR_GRACE errors the operations until the period of special handling is over. See [Section 11.7.7](#) for a discussion of the analogous handling lock reclamation in the case of file systems transitioning from server to server.

During the grace period, the server must reject READ and WRITE operations and non-reclaim locking requests (i.e. other LOCK and OPEN operations) with an error of NFS4ERR_GRACE, unless it can guarantee that these may be done safely, as described below.

The grace period may last until all clients which are known to possibly have had locks have done a global RECLAIM_COMPLETE

Internet-Draft

NFSv4.1

September 2008

operation, indicating that they have finished reclaiming the locks they held before the server restart. This means that a client which has done a RECLAIM_COMPLETE must be prepared to receive an NFS4ERR_GRACE when attempting to acquire new locks. In order for the server to know that all clients with possible prior lock state have done a RECLAIM_COMPLETE, the server must maintain in stable storage a list of clients which may have such locks. The server may also terminate the grace period before all clients have done a global 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 restart, as a result of issuing requests on associated sessions with a frequency governed by the lease time. Note that when a client does not issue such requests (or they are issued by the client but not received by the server), it is possible for the grace period to expire before the client finds out that the server restart has occurred.

Some additional time in order to allow a client to establish a new client ID and session and to effect lock reclaims may be added to the lease time. Note that analogous rules apply to file system-specific grace periods discussed in [Section 11.7.7](#).

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 global 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 byte-range 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 byte-range 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. Another important case is where it is known that no mandatory byte-range locks exist, either because the server does not provide support for them, or because their absence is known from persistently recorded data. In this case, READ and WRITE operations specifying stateids derived from reclaim-type operation may be validly processed during the grace period because the fact of the valid reclaim ensures that no lock subsequently granted can prevent the I/O.

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 [\[36\]](#). The client must account for the server that can perform I/O and non-reclaim locking requests within the grace period as well as those that cannot do so.

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

The possibility exists, that because of server configuration events, the client will be communicating with a server different than the one on which the locks were obtained, as shown by the combination of `eir_server_scope` and `eir_server_owner`. This leads to the issue of if and when the client should attempt to reclaim locks previously obtained on what is being reported as a different server. The rules to resolve this question are as follows:

- o If the server scope is different the client should not attempt to reclaim locks. In this situation no lock reclaim is possible. Any attempt to re-obtain the locks with non-reclaim operations is problematic since there is no guarantee that the existing filehandles will be recognized by the new server, or that if recognized, they denote the same objects. It is best to treat the locks as having been revoked by the reconfiguration event.
- o If the server scope is the same, the client should attempt to reclaim locks, even if the `eir_server_owner` value is different. In this situation, it is the responsibility of the server to return `NFS4ERR_NO_GRACE` if it cannot provide correct support for lock reclaim operations, including the prevention of edge conditions.

The `eir_server_owner` field is not used in making this determination. Its function is to specify trunking possibilities for the client (see [Section 2.10.5](#)) and not to control lock reclaim.

[8.4.3](#). Network Partitions and Recovery

If the duration of a network partition is greater than the lease period provided by the server, the server will not have 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 an expired lease do not prevent such a conflicting lock from being

granted but MUST be 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, even if the conflict is with only a subset of locks.

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 obtain a conflicting a lock, the server may report the loss of lock state in a number of ways.

The server may choose to invalidate the session and the associated client ID. In this case, once the client can communicate with the server, it will receive an NFS4ERR_BADSESSION error. Upon attempting to create a new session, it would get an NFS4ERR_STALE_CLIENTID. Upon creating the new client ID and new session it would attempt to

reclaim locks not be allowed to do so by the server.

Another possibility is for the server to maintain the session and client ID but for all stateids held by the client to become invalid or stale. Once the client can reach the server after such a network partition, the status returned by the SEQUENCE operation will indicate a loss of locking state, i.e. the flag SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED will be set in sr_status_flags. 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 a client's lease has expired, only a subset of stateids will normally become invalid during a network partition. When the client can communicate with the server after such a network partition

heals, the status returned by the SEQUENCE operation will indicate a partial loss of locking state (SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED). 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 restart, there are edge conditions that place requirements on the server in order to avoid silent data corruption following the server restart. 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 the lock.
4. Client B acquires a lock that would have conflicted with that of Client A.
5. Client B releases its lock.
6. Server restarts.
7. Network partition between client A and server heals.
8. Client A connects to new server instance and finds out about server restart.

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 restarts.
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 restarts a second time.
8. Network partition between client A and server heals.
9. Client A connects to new server instance and finds out about server restart.
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 restarts 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:

- o the `co_ownerid` field from the `client_owner4` presented in the `EXCHANGE_ID` operation.
- o a boolean that indicates if the client's lease expired or if there was administrative intervention (see [Section 8.5](#)) to revoke a byte-range lock, share reservation, or delegation and there has been no acknowledgement, via `FREE_STATEID`, of such revocation.
- o 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 suitable `RECLAIM_COMPLETE` (global or file system-specific depending on the target of the lock request) 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 restarts, the record that client A's lease expired means that another client could have acquired a conflicting byte-range 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 restarts 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 restarts with no lock state, the client will send a global 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, byte-range 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 restart, including the two noted in this section, are detected. It is acceptable to erroneously recognize an edge condition and not allow a reclaim, when, with sufficient knowledge it would be allowed. The error the server would return in this case is NFS4ERR_NO_GRACE. Note it is not known if there are other edge conditions.

In the event that, after a server restart, 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 byte-range lock or share reservations (whether they were delegated or not) have been lost, such as via a UNIX signal, a GUI pop-up window, etc. See [Section 10.5](#) for a discussion of what the client should do for dealing with unreclaimed delegations on client state.

For further discussion of revocation of locks see [Section 8.5](#).

Internet-Draft

NFSv4.1

September 2008

[8.5.](#) 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 restart. Note that this includes situations in which sessions are persistent and locking state is lost. In this class of instances, the client will receive an error (NFS4ERR_STALE_CLIENTID) on an operation that takes client ID, usually as part of recovery in response to a problem with the current session) and the client will proceed with normal crash recovery as described in the [Section 8.4.2.1](#).

The second occasion of lock revocation is the inability to renew the lease before expiration, as discussed in [Section 8.4.3](#). While this is considered a rare or unusual event, the client must be prepared to recover. The server is responsible for determining the precise consequences of the lease expiration, informing the client of the scope of the lock revocation decided upon. The client then uses the status information provided by the server in the SEQUENCE results (field sr_status_flags, see [Section 18.46.3](#)) 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 locks revoked during the lease period 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.

[8.6.](#) 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 drops 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 global RECLAIM_COMPLETE. In the event of client failure, there can be a longer period for leases to expire thus forcing conflicting requests to wait.

Long leases are practical if the server can 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.

[8.7.](#) 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 lease. There is also the issue of propagation delay across the network which could easily be several hundred milliseconds as well as the possibility that requests will be lost and need to be retransmitted.

To take propagation delay into account, the client should subtract it from lease times (e.g. if the client estimates the one-way propagation delay as 200 milliseconds, then it can assume that the lease is already 200 milliseconds old when it gets it). In addition,

it will take another 200 milliseconds to get a response back to the server. So the client must send a lease renewal or write data back to the server at least 400 milliseconds before the lease would expire. If the propagation delay varies over the life of the lease (e.g. the client is on a mobile host), the client will need to continuously subtract the increase in propagation delay from the lease times.

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.

[8.8](#). Obsolete Locking Infrastructure From NFSv4.0

There are a number of operations and fields within existing operations that no longer have a function in NFSv4.1. In one way or another, these changes are all due to the implementation of sessions which provides client context and exactly once semantics as a base feature of the protocol, separate from locking itself.

The following NFSv4.0 operations MUST NOT be implemented in NFSv4.1. The server MUST return NFS4ERR_NOTSUPP if these operations are found in an NFSv4.1 COMPOUND.

- o SETCLIENTID since its function has been replaced by EXCHANGE_ID.
- o SETCLIENTID_CONFIRM since client ID confirmation now happens by means of CREATE_SESSION.
- o OPEN_CONFIRM because state-owner-based seqids have been replaced by the sequence ID in the SEQUENCE operation.
- o RELEASE_LOCKOWNER because lock-owners with no associated locks do not have any sequence-related state and so can be deleted by the server at will.
- o RENEW because every SEQUENCE operation for a session causes lease

renewal, making a separate operation superfluous.

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.

- o Sequence ids used to sequence requests for a given state-owner and to provide retry protection, now provided via sessions.
- o 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 have no function in NFSv4.1 and are ignored by the server. Note that client IDs in operations new to NFSv4.1 (such as CREATE_SESSION and DESTROY_CLIENTID) are not ignored.

[9.](#) File Locking and Share Reservations

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 NFSv4.1 protocol defines an OPEN operation which is capable of atomically looking up, creating, and locking a file on the server.

[9.1.](#) Opens and Byte-Range Locks

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

[9.1.1.](#) State-owner Definition

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

Each open is associated with a specific open-owner while each byte-range lock is associated with a lock-owner and an open-owner, the latter being the open-owner associated with the open file under which the LOCK operation was done. Delegations and layouts, on the other hand, are not associated with a specific owner but are associated with the client as a whole (identified by a client ID).

[9.1.2.](#) Use of the Stateid and Locking

All READ, WRITE and SETATTR operations contain a stateid. For the purposes of this section, SETATTR operations which change the size

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

If the state-owner performs a READ or WRITE in a situation in which it has established a byte-range lock or share reservation on the server (any OPEN constitutes a share reservation) the stateid (previously returned by the server) must be used to indicate what locks, including both byte-range locks and share reservations, are held by the state-owner. If no state is established by the client, either byte-range lock or share reservation, a special stateid for

anonymous state (zero as "other" and "seqid") is used. (See [Section 8.2.3](#) for a description of 'special' stateids in general.) 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 byte-range lock held on the file, the server MUST refuse to service the READ or WRITE operation.

Share reservations are established by OPEN operations and by their nature are mandatory in that when the OPEN denies READ or WRITE operations, that denial results in such operations being rejected with error NFS4ERR_LOCKED. Byte-range locks may be implemented by the server as either mandatory or advisory, or the choice of mandatory or advisory behavior may be determined by the server on the basis of the file being accessed (for example, some UNIX-based servers support a "mandatory lock bit" on the mode attribute such that if set, byte-range locks are required on the file before I/O is possible). When byte-range locks are advisory, they only prevent the granting of conflicting lock requests and have no effect on READs or WRITEs. Mandatory byte-range locks, however, prevent conflicting I/O operations. When they are attempted, they are rejected with NFS4ERR_LOCKED. When the client gets NFS4ERR_LOCKED on a file it knows it has the proper share reservation for, it will need to send 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 byte-range locks are exactly the same in so far as the APIs and requirements on implementation. If the mandatory lock attribute is set on the file, the server checks to see if the lock-owner has an appropriate shared (read) or exclusive (write) byte-range lock on the region it wishes to read or write to.

If there is no appropriate lock, the server checks if there is a conflicting lock (which can be done by attempting to acquire the conflicting lock on 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, byte-range locks are always mandatory, so the server always checks for byte-range locks during I/O requests.

Thus, the NFSv4.1 LOCK operation does not need to distinguish between advisory and mandatory byte-range locks. It is the NFSv4.1 server's processing of the READ and WRITE operations that introduces the distinction.

Every stateid which is validly passed to READ, WRITE or SETATTR, with the exception of special stateid values, defines an access mode for the file (i.e. READ, WRITE, or READ-WRITE)

- o For stateids associated with opens, this is the mode defined 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.
- o For stateids returned by byte-range lock requests, the appropriate mode is the access mode for the open stateid associated with the lock set represented by the stateid.
- o For delegation stateids the access mode is 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 stateid 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 MUST 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.

to one) 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 scope of the lock to be granted would conflict with the READ or WRITE operation. This can occur when:

- o A mandatory byte range lock is requested with range that 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.
- o A share reservation is requested which denies reading and or writing and the corresponding operation is being performed.
- o A delegation is to be granted and the delegation type would prevent the I/O operation, i.e. READ and WRITE conflict with a write delegation and WRITE conflicts with a read delegation.

When a client holds a delegation, it needs to ensure that the stateid sent conveys the association of operation with the delegation, to avoid the delegation from being avoidably recalled. When the delegation stateid, or a stateid open associated with that delegation, or a stateid representing byte-range locks derived from such an open is used, the server knows that the READ, WRITE, or SETATTR does not conflict with the delegation, but is sent under the aegis of the delegation. Even though it is possible for the server to determine from the client ID (via the session ID) that the client does in fact have a delegation, the server is not obliged to check this, so using a special stateid can result in avoidable recall of the delegation.

[9.2.](#) Lock Ranges

The protocol allows a lock-owner to request a lock with a byte range and then either upgrade, downgrade, or unlock a sub-range of the initial lock, or a range that consists of a range which overlaps, fully or partially, that initial lock or a combination of a set of existing locks for the same lock-owner. It is expected that this will be an uncommon type of request. In any case, servers or server

file systems may not be able to support sub-range lock semantics. In the event that a server receives a locking request that represents a sub-range of current locking state for the lock-owner, the server is allowed to return the error NFS4ERR_LOCK_RANGE to signify that it does not support sub-range lock operations. Therefore, the client should be prepared to receive this error and, if appropriate, report the error to the requesting application.

The client is discouraged from combining multiple independent locking ranges that happen to be adjacent into a single request since the server may not support sub-range requests and for reasons related to the recovery of file locking state in the event of server failure. As discussed in [Section 8.4.2](#), the server may employ certain optimizations during recovery that work effectively only when the client's behavior during lock recovery is similar to the client's locking behavior prior to server failure.

[9.3.](#) Upgrading and Downgrading Locks

If a client has a write lock on a byte-range, 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 byte-range, 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 sent the LOCK request with the type set to WRITEW_LT and the server has detected a deadlock. The client should be prepared to receive such errors and if appropriate, report the error to the requesting application.

[9.4.](#) Stateid Seqid Values and Byte-Range Locks

When a lock or unlock request is done, passing a stateid, the stateid returned has the same "other" value and a "seqid" value that is incremented to reflect the occurrence of the lock or unlock request. The server MUST increment the value of the "seqid" field whenever there is any change to the locking status of any byte offset as

described by any of locks covered by the stateid. A change in locking status includes a change from locked to unlocked or the

reverse or a change from being locked for read to being locked for write or the reverse.

When there is no such change, as, for example when a range already locked for write is locked again for write, the server MAY increment the "seqid" value.

[9.5.](#) Issues with Multiple Open-Owners

When the same file is opened by multiple open-owners, a client will have multiple open stateids for that file, each associated with a different open-owner. In that case, there can be multiple LOCK and LOCKU requests for the same lock-owner issued using the different open stateids, and so a situation may arise in which there are multiple stateids, each representing byte-range locks on the same file and held by the same lock-owner but each associated with a different open-owner.

In such a situation, the locking status of each byte (i.e. whether it is locked, the read or write mode of the lock and the lock-owner holding the lock) MUST reflect the last LOCK or LOCKU operation done for the lock-owner in question, independent of the stateid through which the request was issued.

When a byte is locked by the lock-owner in question, the open-owner to which that lock is assigned SHOULD be that of the open-owner associated with the stateid through which the last LOCK of that byte was done. When there is a change in the open-owner associated with locks for the stateid through which a LOCK or LOCKU was done, the "seqid" field of the stateid MUST be incremented, even if the locking, in terms of lock-owners has not changed. When there is a change to the set of locked bytes associated with a different stateid for the same lock-owner, i.e. associated with a different open-owner, the "seqid" value for that stateid MUST NOT be incremented.

[9.6.](#) Blocking Locks

Some clients require the support of blocking locks. While NFSv4.1 provides a callback when a previously unavailable lock becomes

available, this is an OPTIONAL feature and clients cannot depend on its presence. Clients need to be prepared to continually poll for the lock. This presents a fairness problem. Two of the lock types, READW and WRITEW, are used to indicate to the server that the client is requesting a blocking lock. When the callback is not used, the server should maintain an ordered list of pending blocking locks. When the conflicting lock is released, the server may wait for the period of time equal to lease_time 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.

When server indicates, via the flag OPEN4_RESULT_MAY_NOTIFY_LOCK, that CB_NOTIFY_LOCK callbacks will be done for the current open file, the client should take notice of this, but, since this is a hint, cannot rely on a CB_NOTIFY_LOCK always being done. A client may

reasonably reduce the frequency with which it polls for a denied lock, since the greater latency that might occur is likely to be eliminated given a prompt callback, but it still needs to poll. When it receives a CB_NOTIFY_LOCK it should promptly try to obtain the lock, but it should be aware that other clients may be polling and the server is under no obligation to reserve the lock for that particular client.

[9.7.](#) Share Reservations

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

Pseudo-code definition of the semantics:

```
if (request.access == 0) {
    return (NFS4ERR_INVAL)
} else {
    if ((request.access & file_state.deny) ||
        (request.deny & file_state.access)) {
        return (NFS4ERR_SHARE_DENIED)
    }
    return (NFS4ERR_OK);
}
```

When doing this checking of share reservations on OPEN, the current file_state used in the algorithm includes bits that reflect all current opens, including those for the open-owner making the new OPEN request.

The constants used for the OPEN and OPEN_DOWNGRADE operations for the access and deny fields are as follows:

```
const OPEN4_SHARE_ACCESS_READ    = 0x00000001;
const OPEN4_SHARE_ACCESS_WRITE   = 0x00000002;
const OPEN4_SHARE_ACCESS_BOTH    = 0x00000003;

const OPEN4_SHARE_DENY_NONE      = 0x00000000;
const OPEN4_SHARE_DENY_READ      = 0x00000001;
```

```
const OPEN4_SHARE_DENY_WRITE    = 0x00000002;
const OPEN4_SHARE_DENY_BOTH     = 0x00000003;
```

[9.8.](#) 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 byte-range locks are held, the client **SHOULD** release all locks before issuing a CLOSE. The server **MAY** free all outstanding locks on CLOSE but some servers may not support the CLOSE of a file that still has byte-range locks held. The server **MUST**

return failure, NFS4ERR_LOCKS_HELD, if any locks would exist after the CLOSE.

The LOOKUP operation will return a filehandle without establishing any lock state on the server. Without a valid stateid, the server will assume the client has the least access. For example, 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.

[9.9.](#) 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. The open is represented by a single stateid whose "other" values matches that of the original open, and whose "seqid" value is incremented to reflect the occurrence of the upgrade. The increment is required in cases in which the "upgrade" results in no change to the open mode (e.g. an OPEN is done for read when the existing open file is opened for read-write). Only a single CLOSE will be done to reset the effects of both OPENS. The client may use the stateid returned by the OPEN effecting the upgrade or with a stateid sharing the same "other" field and a seqid of zero, although care needs to be taken as far as upgrades which happen while the CLOSE is pending. Note that the client, when issuing the OPEN, may not know that the same file is in fact being opened. The above only applies if both OPENS result in the OPENed object being designated by the same filehandle.

When the server chooses to export multiple filehandles corresponding to the same file object and returns different filehandles on two different OPENS of the same file object, the server MUST NOT "OR" together the access and deny bits and coalesce the two open files. Instead the server must maintain separate OPENS with separate stateids and will require separate CLOSEs to free them.

When multiple open files on the client are merged into a single open file object on the server, the close of one of the open files (on the client) may necessitate change of the access and deny status of the open file on the server. This is because the union of the access and deny bits for the remaining opens may be smaller (i.e. a proper subset) than previously. The OPEN_DOWNGRADE operation is used to make the necessary change and the client should use it to update the

server so that share reservation requests by other clients are handled properly. The stateid returned has the same "other" field as that passed to the server. The "seqid" value in the returned stateid MUST be incremented, even in situation in which there is no change the access and deny bits for the file.

[9.10.](#) Parallel OPENS

Unlike the case of NFSv4.0, in which OPEN operations for the same open-owner are inherently serialized because of the owner-based seqid, multiple OPENS for the same open-owner may be done in

parallel. When clients do this, they may encounter situations in which, because of the existence of hard links, two OPEN operations may turn out to open the same file, with a later OPEN performed being an upgrade of the first, with this fact only visible to the client once the operations complete.

In this situation, clients may determine the order in which the OPENs were performed by examining the stateids returned by the OPENs. Stateids that share a common value of the "other" field can be recognized as having opened the same file, with the order of the operations determinable from the order of the "seqid" fields, mod any possible wraparound of the 32-bit field.

When the possibility exists that the client will send multiple OPENs for the same open-owner in parallel, it may be the case that an open upgrade may happen without the client knowing beforehand that this could happen. Because of this possibility, CLOSEs and OPEN_DOWNGRADEs, should generally be sent with a non-zero seqid in the stateid, to avoid the possibility that the status change associated with an open upgrade is not inadvertently lost.

[9.11.](#) Reclaim of Open and Byte-Range Locks

Special forms of the LOCK and OPEN operations are provided when it is necessary to re-establish byte-range locks or opens after a server failure.

- o To reclaim existing opens, an OPEN operation is performed using a CLAIM_PREVIOUS. Because the client, in this type of situation, will have already opened the file and have the filehandle of the target file, this operation requires that the current filehandle be the target file, rather than a directory and no file name is specified.
- o To reclaim byte-range locks, a LOCK operation with the reclaim parameter set to true is used.

10. Client-Side Caching

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

The NFSv4.1 protocol uses many techniques similar to those that have been used in previous protocol versions. The NFSv4.1 protocol does not provide distributed cache coherence. However, it defines a more limited set of caching guarantees to allow locks and share reservations to be used without destructive interference from client side caching.

In addition, the NFSv4.1 protocol introduces a delegation mechanism which allows many decisions normally made by the server to be made locally by clients. This mechanism provides efficient support of the common cases where sharing is infrequent or where sharing is read-only.

10.1. Performance Challenges for Client-Side Caching

Caching techniques used in previous versions of the NFS protocol have been successful in providing good performance. However, several scalability challenges can arise when those techniques are used with very large numbers of clients. This is particularly true when clients are geographically distributed which classically increases the latency for cache 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 [Section 10.3.2](#)) can severely limit performance. When locking is used to provide protection against infrequent conflicts, a large penalty is incurred. This penalty may discourage the use of file locking by applications.

The NFSv4.1 protocol provides more aggressive caching strategies with the following design goals:

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

The appropriate requirements for the server are discussed in later sections in which specific forms of caching are covered (see [Section 10.4](#)).

[10.2](#). Delegation and Callbacks

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

A delegation is passed from the server to the client, specifying the object of the delegation and the type of delegation. There are different types of delegations but each type contains a stateid to be used to represent the delegation when performing operations that depend on the delegation. This stateid is similar to those associated with locks and share reservations but differs in that the stateid for a delegation is associated with a client ID and may be used on behalf of all the open-owners for the given client. A delegation is made to the client as a whole and not to any specific process or thread of control within it.

The backchannel is established by CREATE_SESSION and

Because the backchannel may be down, even temporarily, correct protocol operation does not depend on them. Preliminary testing of backchannel 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. For individual operations, we will describe, under IMPLEMENTATION, when such operations are required to effect a recall. A number of points should be noted, however.

- o The server is free to recall a delegation whenever it feels it is desirable and may do so even if no operations requiring recall are being done.
- o Operations done outside the NFSv4 protocol, due to, for example, access by other protocols, or by local access, also need to result in delegation recall when they make analogous changes to file system data. What is crucial is if the change would invalidate the guarantees provided by the delegation. When this is possible, the delegation needs to be recalled and must be returned or revoked before allowing the operation to proceed.
- o The semantics of the file system are crucial in defining when delegation recall is required. If a particular change within a specific implementation causes change to a file attribute, then delegation recall is required, whether that operation has been specifically listed as requiring delegation recall. Again, what is critical is whether the guarantees provided by the delegation

are being invalidated.

Despite those caveats, the implementation sections for a number of operations describe situations in which delegation recall would be required under some common circumstances:

- o For GETATTR, see [Section 18.7.4](#).

- o For OPEN, see [Section 18.16.4](#).
- o For READ, see [Section 18.22.4](#).
- o For REMOVE, see [Section 18.25.4](#).
- o For RENAME, see [Section 18.26.4](#).
- o For SETATTR, see [Section 18.30.4](#).
- o For WRITE, see [Section 18.32.4](#).

On recall, the client holding the delegation must flush modified state (such as modified data) to the server and return the delegation. The conflicting request will not be acted on until the recall is complete. The recall is considered complete when the client returns the delegation or the server times its wait for the delegation to be returned and revokes the delegation as a result of the timeout. In the interim, the server will either delay responding to conflicting requests or respond to them with NFS4ERR_DELAY. Following the resolution of the recall, the server has the information necessary to grant or deny the second client's request.

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

An example of this is when responsibility to mediate opens on a given

file is delegated to a client (see [Section 10.4](#)). The server will not know what opens are in effect on the client. Without this knowledge the server will be unable to determine if the access and deny state for the file allows any particular open until the delegation for the file has been returned.

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

[10.2.1](#). Delegation Recovery

There are three situations that delegation recovery must deal with:

- o Client restart
- o Server restart
- o Network partition (full or backchannel-only)

In the event the client restarts, the failure to renew the lease will result in the revocation of byte-range locks and share reservations. Delegations, however, may be treated a bit differently.

There will be situations in which delegations will need to be reestablished after a client 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 restart 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 or CLAIM_DELEG_PREV_FH (See [Section 10.5](#) and [Section 18.16](#) for discussion of open delegation and the details of OPEN respectively).

A server MAY support claim types of CLAIM_DELEGATE_PREV and CLAIM_DELEG_PREV_FH, and 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 send CLAIM_DELEGATE_PREV requests. The server that supports CLAIM_DELEGATE_PREV and/or CLAIM_DELEG_PREV_FH MUST support the DELEGPURGE operation.

When the server restarts, delegations are reclaimed (using the OPEN operation with CLAIM_PREVIOUS) in a similar fashion to byte-range locks and share reservations. However, there is a slight semantic difference. In the normal case if the server decides that a delegation should not be granted, it performs the requested action (e.g. OPEN) without granting any delegation. For reclaim, the server grants the delegation but a special designation is applied so that the client treats the delegation as having been granted but recalled by the server. Because of this, the client has the duty to

write all modified state to the server and then return the delegation. This process of handling delegation reclaim reconciles three principles of the NFSv4.1 protocol:

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

When a client needs to reclaim a delegation and there is no associated open, the client may use the CLAIM_PREVIOUS variant of the WANT_DELEGATION operation. However, since the server is not required to support this operation, an alternative is to reclaim via a dummy open together with the delegation using an OPEN of type

CLAIM_PREVIOUS. The dummy open file can be released using a CLOSE to re-establish the original state to be reclaimed, a delegation without an associated open.

When a client has more than a single open associated with a delegation, state for those additional opens can be established using OPEN operations of type CLAIM_DELEGATE_CUR. When these are used to establish opens associated with reclaimed delegations, the server MUST allow them when made within the grace period.

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 backchannel (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 one of the errors NFS4ERR_EXPIRED, NFS4ERR_ADMIN_REVOKED, or NFS4ERR_DELEG_REVOKED. It also may find out about delegation revocation after a client restart 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 [Section 10.5.1](#) for a discussion of

such issues. Note also that when delegations are revoked, information about the revoked delegation will be written by the server to stable storage (as described in [Section 8.4.3](#)). This is done to deal with the case in which a server restarts after revoking a delegation but before the client holding the revoked delegation is notified about the revocation.

[10.3.](#) Data Caching

When applications share access to a set of files, they need to be implemented so as to take account of the possibility of conflicting access by another application. This is true whether the applications

in question execute on different clients or reside on the same client.

Share reservations and byte-range locks are the facilities the NFSv4.1 protocol provides to allow applications to coordinate access by using mutual exclusion facilities. The NFSv4.1 protocol's data caching must be implemented such that it does not invalidate the assumptions that those using these facilities depend upon.

[10.3.1](#). Data Caching and OPENS

In order to avoid invalidating the sharing assumptions that applications rely on, NFSv4.1 clients should not provide cached data to applications or modify it on behalf of an application when it would not be valid to obtain or modify that same data via a READ or WRITE operation.

Furthermore, in the absence of open delegation (see [Section 10.4](#)), two additional rules apply. Note that these rules are obeyed in practice by many NFSv3 clients.

- o First, cached data present on a client must be revalidated after doing an OPEN. Revalidating means that the client fetches the change attribute from the server, compares it with the cached change attribute, and if different, declares the cached data (as well as the cached attributes) as invalid. This is to ensure that the data for the OPENed file is still correctly reflected in the client's cache. This validation must be done at least when the client's OPEN operation includes DENY=WRITE or BOTH thus terminating a period in which other clients may have had the opportunity to open the file with WRITE access. Clients may choose to do the revalidation more often (i.e. at OPENS specifying DENY=NONE) to parallel the NFSv3 protocol's practice for the benefit of users assuming this degree of cache revalidation.

Since the change attribute is updated for data and metadata

modifications, some client implementors may be tempted to use the time_modify attribute and not the change attribute to validate cached data, so that metadata changes do not spuriously invalidate clean data. The implementor is cautioned in this approach. The change attribute is guaranteed to change for each update to the

file, whereas `time_modify` is guaranteed to change only at the granularity of the `time_delta` attribute. Use by the client's data cache validation logic of `time_modify` and not change runs the risk of the client incorrectly marking stale data as valid.

- o Second, modified data must be flushed to the server before closing a file OPENed for write. This is complementary to the first rule. If the data is not flushed at CLOSE, the revalidation done after 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 restart and a CLOSEd file, it may not be possible to retransmit the data to be written to the file. Hence, this requirement.

10.3.2. Data Caching and File Locking

For those applications that choose to use file locking instead of share reservations to exclude inconsistent file access, there is an analogous set of constraints that apply to client side data caching. These rules are effective only if the file locking is used in a way that matches in an equivalent way the actual READ and WRITE operations executed. This is as opposed to file locking that is based on pure convention. For example, it is possible to manipulate a two-megabyte file by dividing the file into two one-megabyte regions and protecting access to the two regions by file locks on bytes zero and one. A lock for write on byte zero of the file would represent the right to do READ and WRITE operations on the first region. A lock for write on byte one of the file would represent the right to do READ and WRITE operations on the second region. As long as all applications manipulating the file obey this convention, they will work on a local file system. However, they may not work with the NFSv4.1 protocol unless clients refrain from data caching.

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

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

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

Note that flushing data to the server and the invalidation of cached data must reflect the actual byte ranges locked or unlocked. Rounding these up or down to reflect client cache block boundaries will cause problems if not carefully done. For example, writing a modified block when only half of that block is within an area being unlocked may cause invalid modification to the region outside the unlocked area. This, in turn, may be part of a region locked by another client. Clients can avoid this situation by synchronously performing portions of write operations that overlap that portion (initial or final) that is not a full block. Similarly, invalidating a locked area which is not an integral number of full buffer blocks would require the client to read one or two partial blocks from the server if the revalidation procedure shows that the data which the client possesses may not be valid.

The data that is written to the server as a prerequisite to the unlocking of a region must be written, at the server, to stable storage. The client may accomplish this either with synchronous writes or by following asynchronous writes with a COMMIT operation. This is required because retransmission of the modified data after a server restart might conflict with a lock held by another client.

A client implementation may choose to accommodate applications which use byte-range locking in non-standard ways (e.g. using a byte-range lock as a global semaphore) by flushing to the server more data upon 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 byte-range locks which the application holds. For example, an application locks a single byte of a file and proceeds to write that single byte. A client that chose to handle a LOCKU by flushing all modified data to the server could validly write that single byte in response to an unrelated unlock. However, it would not be valid to write the entire block in which that single written byte was located since it includes an area that is not locked and might be locked by another client. Client implementations can avoid this problem by dividing files with modified data into those for which all modifications are done to areas covered by an appropriate byte-range lock and those for which there are modifications not covered by a byte-range lock. Any writes done for the former class of files must

Internet-Draft

NFSv4.1

September 2008

not include areas not locked and thus not modified on the client.

[10.3.3.](#) Data Caching and Mandatory File Locking

Client side data caching needs to respect mandatory file locking when it is in effect. The presence of mandatory file locking for a given file is indicated when the client gets back NFS4ERR_LOCKED from a READ or WRITE on a file it has an appropriate share reservation for. When mandatory locking is in effect for a file, the client must check for an appropriate file lock for data being read or written. If a lock exists for the range being read or written, the client may satisfy the request using the client's validated cache. If an appropriate file lock is not held for the range of the read or write, the read or write request must not be satisfied by the client's cache and the request must be sent to the server for processing. When a read or write request partially overlaps a locked region, the request should be subdivided into multiple pieces with each region (locked or not) treated appropriately.

[10.3.4.](#) Data Caching and File Identity

When clients cache data, the file data needs to be organized according to the file system object to which the data belongs. For NFSv3 clients, the typical practice has been to assume for the purpose of caching that distinct filehandles represent distinct file system objects. The client then has the choice to organize and maintain the data cache on this basis.

In the NFSv4.1 protocol, there is now the possibility to have significant deviations from a "one filehandle per object" model because a filehandle may be constructed on the basis of the object's pathname. Therefore, clients need a reliable method to determine if two filehandles designate the same file system object. If clients were simply to assume that all distinct filehandles denote distinct objects and proceed to do data caching on this basis, caching inconsistencies would arise between the distinct client side objects which mapped to the same server side object.

By providing a method to differentiate filehandles, the NFSv4.1 protocol alleviates a potential functional regression in comparison with the NFSv3 protocol. Without this method, caching inconsistencies within the same client could occur and this has not

been present in previous versions of the NFS protocol. Note that it is possible to have such inconsistencies with applications executing on multiple clients but that is not the issue being addressed here.

For the purposes of data caching, the following steps allow an NFSv4.1 client to determine whether two distinct filehandles denote

the same server side object:

- o If GETATTR directed to two filehandles returns different values of the fsid attribute, then the filehandles represent distinct objects.
- o If GETATTR for any file with an fsid that matches the fsid of the two filehandles in question returns a unique_handles attribute with a value of TRUE, then the two objects are distinct.
- o If GETATTR directed to the two filehandles does not return the fileid attribute for both of the handles, then it cannot be determined whether the two objects are the same. Therefore, operations which depend on that knowledge (e.g. client side data caching) cannot be done reliably. Note that if GETATTR does not return the fileid attribute for both filehandles, it will return it for neither of the filehandles, since the fsid for both filehandles is the same.
- o If GETATTR directed to the two filehandles returns different values for the fileid attribute, then they are distinct objects.
- o Otherwise they are the same object.

[10.4.](#) Open Delegation

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

set of conditions that servers might use in deciding whether OPEN should be delegated:

- o The client must be able to respond to the server's callback requests. If a backchannel has been established, the server will send a CB_COMPOUND request, containing a single operation, CB_SEQUENCE, for a test of backchannel availability.
- 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 is assured that neither the contents, the attributes (with the exception of `time_access`), nor the names of any links to the file will change without its knowledge, so long as the delegation is held. When a client has a write open delegation, it may modify the file data locally since no other client will be accessing the file's data. The client holding a write delegation may only locally affect file attributes which are intimately connected with the file data: `size`, `change`, `time_access`, `time_metadata`, and `time_modify`. All other attributes must be

reflected on the server.

When a client has an open delegation, it does not need to send OPENS or CLOSEs to the server. Instead the client may update 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 reply 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 [Section 10.4.1](#)).
- o an nfsace4 specifying read and write permissions.
- o a stateid to represent the delegation for READ and WRITE.

The delegation stateid is separate and distinct from the stateid for the OPEN proper. The standard stateid, unlike the delegation stateid, is associated with a particular 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 an open delegation is in effect, it will be accepted or rejected solely on the basis of the following conditions. Any requirement for other checks to be made by the delegate should result in open delegation being denied so that the checks can be made by the server itself.

- o The access and deny bits for the request and the file as described in [Section 9.7](#).
- 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 a delegation together with various other forms of caching creates the possibility that no server authentication and authorization 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 and authorization, the client should be sure authentication and authorization occurs for each user by use of the ACCESS operation. This should be the case even if an ACCESS operation would not be required otherwise. As mentioned before, the server may enforce frequent authentication by returning an nfsace4 denying all access with every open delegation.

[10.4.1.](#) Open Delegation and Data Caching

An 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 written 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. Although client SHOULD NOT use special stateids when an open exists, delegation handling on the server can use the client ID associated with the current session to determine if the operation has been done by the holder of the delegation, in which case, no recall is necessary, or by another client, in which case the delegation must be recalled and I/O not proceed until the delegation is recalled or revoked.

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 write modified 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.

[10.4.2.](#) 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.

[10.4.3.](#) Handling of CB_GETATTR

The server needs to employ special handling for a GETATTR where the target is a file that has a 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:

- o The value of the change attribute will be obtained from the server and cached. Let this value be represented by *c*.
- o The client will create a value greater than *c* that will be used for communicating modified data is held at the client. Let this value be represented by *d*.
- o When the client is queried via CB_GETATTR for the change attribute, it checks to see if it holds modified data. If the file is modified, the value *d* is returned for the change attribute value. If this file is not currently modified, the client returns the value *c* for the change attribute.

For simplicity of implementation, the client MAY for each CB_GETATTR return the same value *d*. This is true even if, between successive CB_GETATTR operations, the client again modifies in the file's data or metadata in its cache. The client can return the same value because the only requirement is that the client be able to indicate to the server that the client holds modified data. Therefore, the value of *d* may always be *c* + 1.

While the change attribute is opaque to the client in the sense that it has no idea what units of time, if any, the server is counting change with, it is not opaque in that the client has to treat it as an unsigned integer, and the server has to be able to see the results of the client's changes to that integer. Therefore, the server MUST encode the change attribute in network order when sending it to the client. The client MUST decode it from network order to its native order when receiving it and the client MUST encode it network order when sending it to the server. For this reason, change is defined as an unsigned integer rather than an opaque array of bytes.

For the server, the following steps will be taken when providing a write delegation:

Internet-Draft

NFSv4.1

September 2008

- o 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`.
- o When a second client sends a GETATTR operation on the same file to the server, the server obtains the change attribute from the first client. Let this value be `cc`.
- o If the value `cc` is equal to `sc`, the file is not modified and the server returns the current values for `change`, `time_metadata`, and `time_modify` (for example) to the second client.
- o If the value `cc` is NOT equal to `sc`, the file is currently modified at the first client and most likely will be modified at the server at a future time. The server then uses its current time to construct attribute values for `time_metadata` and `time_modify`. A new value of `sc`, which we will call `nsc`, is computed by the server, such that $nsc \geq sc + 1$. 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 != sc`, and would stay TRUE until the delegation is returned or revoked. The processing for constructing `nsc`, `time_modify`, and `time_metadata` would use this pseudo code:

```
if (!modified) {
    do CB_GETATTR for change and size;

    if (cc != sc)
        modified = TRUE;
} else {
    do CB_GETATTR for size;
}

if (modified) {
    sc = sc + 1;
    time_modify = time_metadata = current_time;
    update sc, time_modify, time_metadata into file's metadata;
}
```

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

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

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

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

[10.4.4.](#) Recall of Open Delegation

The following events necessitate recall of an open delegation:

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

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

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

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

- o If the file associated with the delegation is no longer open and no previous CLOSE operation has been sent to the server, a CLOSE operation must be sent to the server.
- o If a file has other open references at the client, then OPEN operations must be sent to the server. The appropriate stateids will be provided by the server for subsequent use by the client since the delegation stateid will no 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 18.16](#) which describes the OPEN" operation for

details.)

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

[10.4.5.](#) 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 backchannel from server to the client. The client may be unaware of a failure in the backchannel. 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.

Status bits returned by SEQUENCE operations help to provide an alternate way of informing the client of issues regarding the status of the backchannel and of recalled delegations. When the backchannel is not available, the server returns the status bit SEQ4_STATUS_CB_PATH_DOWN on SEQUENCE operations. The client can react by attempting to re-establish the backchannel and by returning recallable objects if a backchannel cannot be successfully re-established.

Whether the backchannel is functioning or not, it may be that the recalled delegation is not returned. Note that the client's lease might still be renewed, even though the recalled delegation is not returned. In this situation, servers SHOULD revoke delegations that are not returned in a period of time equal to the lease period. This period of time should allow the client time to note the backchannel-down status and re-establish the backchannel.

When delegations are revoked, the server will return with the SEQ4_STATUS_RECALLABLE_STATE_REVOKED status bit set on subsequent SEQUENCE operations. The client should note this and then use TEST_STATEID to find which delegations have been recalled.

[10.4.6.](#) Delegation Revocation

At the point a delegation is revoked, if there are associated opens on the client, these opens may or may not be revoked. If no lock or open is granted that is inconsistent with the existing open, the stateid for the open may remain valid, and be disconnected from the revoked delegation, just as would be the case if the delegation were returned.

For example, if an OPEN for read-write with DENY=NONE is associated with the delegation, granting of another such OPEN to a different client will revoke the delegation but need not revoke the OPEN, since no lock inconsistent with that OPEN has been granted. On the other hand, if an OPEN denying write is granted, then the existing open must be revoked.

When opens and/or locks are revoked, the applications holding these opens or locks need to be notified. This notification usually occurs by returning errors for READ/WRITE operations or when a close is attempted for the open file.

If no opens exist for the file at the point the delegation is revoked, then notification of the revocation is unnecessary. However, if there is modified data present at the client for the file, the user of the application should be notified. Unfortunately, it may not be possible to notify the user since active applications may not be present at the client. See [Section 10.5.1](#) for additional details.

[10.4.7.](#) Delegations via WANT_DELEGATION

In addition to providing delegations as part of the reply to OPEN operations, servers MAY provide delegations separate from open, via the OPTIONAL WANT_DELEGATION operation. This allows delegations to be obtained in advance of an OPEN that might benefit from them, for

objects which are not a valid target of OPEN, or to deal with cases in which a delegation has been recalled and the client wants to make an attempt to re-establish it if the absence of use by other clients allows that.

The WANT_DELEGATION operation may be performed on any type of file object other than a directory.

When a delegation is obtained using WANT_DELEGATION, any open files for the same filehandle held by that client are to be treated as subordinate to the delegation, just as if they had been created using an OPEN of type CLAIM_DELEGATE_CUR. They are otherwise unchanged as to seqid, access and deny modes, and the relationship with byte-range locks. Similarly, existing byte-range locks subordinate to an open which becomes subordinate to a delegation, become indirectly subordinate to that new delegation.

The WANT_DELEGATION operation provides for delivery of delegations via callbacks, when the delegations are not immediately available. When a requested delegation is available, it is delivered to the client via a CB_PUSH_DELEG operation. When this happens, open files for the same filehandle become subordinate to the new delegation at the point at which the delegation is delivered, just as if they had been created using an OPEN of type CLAIM_DELEGATE_CUR. Similarly, for existing byte-range locks subordinate to an open.

10.5. Data Caching and Revocation

When locks and delegations are revoked, the assumptions upon which successful caching depend are no longer guaranteed. For any locks or share reservations that have been revoked, the corresponding state-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 state-owner will in many cases consist of simply returning an error on the next and all subsequent READs/WRITEs to the open file or on the close. Where the methods available to a client

make such notification impossible because errors for certain

operations may not be returned, more drastic action such as signals or process termination may be appropriate. The justification for this is that an invariant for which an application depends on may be violated. Depending on how errors are typically treated for the client operating environment, further levels of notification including logging, console messages, and GUI pop-ups may be appropriate.

[10.5.1.](#) Revocation Recovery for Write Open Delegation

Revocation recovery for a 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.

[10.6.](#) Attribute Caching

This section pertains to the caching of a file's attributes on a client when that client does not hold a delegation on the file.

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 reaction 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.

- o All attributes for a given file (per-fsid attributes excepted) are cached as a unit at the client so that no non-serializability can arise within the context of a single file.
- o An upper time boundary is maintained on how long a client cache entry can be kept without being refreshed from the server.
- o When operations are performed that 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.

10.7. Data and Metadata Caching and Memory Mapped Files

Some operating environments include the capability for an application to map a file's content into the application's address space. Each time the application accesses a memory location that corresponds to a

Shepler, et al.

Expires March 9, 2009

[Page 213]

Internet-Draft

NFSv4.1

September 2008

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:

- o If there is an application on the server that has memory mapped a file that a client is also accessing, the client may not be able to get a consistent value of the `change` attribute to determine whether its cache is stale or not. A server that knows that the file is memory mapped could always pessimistically return updated values for `change` so as to force the application to always get the most up to date data and metadata for the file. However, due to the negative performance implications of this, such behavior is OPTIONAL.
- o If the memory mapped file is not being modified on the server, and instead is just being read by an application via the memory mapped interface, the client will not see an updated `time_access` attribute. However, in many operating environments, neither will any process running on the server. Thus NFS clients are at no disadvantage with respect to local processes.

- o If there is another client that is memory mapping the file, and if that client is holding a 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 reply from CB_GETATTR will not necessarily be accurate. As discussed earlier, the client's obligation is to report that the file has been modified since the delegation was granted, not whether it has been modified again between successive CB_GETATTR calls, and the server MUST assume that any file the client has modified in cache has been modified again between successive CB_GETATTR calls. Depending on the nature of the client's memory management system, this weak obligation may not be possible. A client MAY return stale information in CB_GETATTR whenever the file is memory mapped.

- o The mixture of memory mapping and file locking on the same file is problematic. Consider the following scenario, where a page size on each client is 8192 bytes.
 - * Client A memory maps first page (8192 bytes) of file X
 - * Client B memory maps first page (8192 bytes) of file X
 - * Client A write locks first 4096 bytes
 - * Client B write locks second 4096 bytes
 - * Client A, via a STORE instruction modifies part of its locked region.
 - * Simultaneous to client A, client B executes 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 execute the STORE instructions, the resulting page faults require a byte-range lock on the entire page. Each client then tries to extend their locked range to the entire page, which results in a deadlock. Communicating the NFS4ERR_DEADLOCK error to a STORE instruction is difficult at best.

If a client is locking the entire memory mapped file, there is no problem with advisory or mandatory byte-range locking, at least until the client unlocks a region in the middle of the file.

Given the above issues the following are permitted:

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

- o A client MAY deny memory mapping a file that it knows requires mandatory locking for I/O. If mandatory locking is enabled after the file is opened and mapped, the client MAY deny the application further access to its mapped file.

[10.8.](#) Name and Directory Caching without Directory Delegations

The NFSv4.1 directory delegation facility (described in [Section 10.9](#) below) is OPTIONAL for servers to implement. Even where it is implemented, it may not always be functional because of resource availability issues or other constraints. Thus, it is important to understand how name and directory caching are done in the absence of directory delegations. Those topics are discussed in the next in [Section 10.8.1](#).

[10.8.1.](#) 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 has a basis for determining, given proper care, whether other clients are modifying the directory is question.

The simplest way to enable the client to make this determination is for the client to serialize all changes made to a specific directory. When this is done, and the server provides before and after values of the change attribute atomically, the client can simply compare the

after value of the change attribute from one operation on a directory with the before value on the next subsequent operation modifying that directory. When these are equal, the client is assured that no other client is modifying the directory in question.

When such serialization is not used, and there may be multiple simultaneous outstanding operations modifying a single directory sent from a single client, making this sort of determination can be more complicated, since two such operations which are recognized as complete in a different order than they were actually performed, might give an appearance consistent with modification being made by

another client. Where this appears to happen, the client needs to await the completion of all such modifications that were started previously, to see if the outstanding before and after change numbers can be sorted into a chain such that the before value of one change number matches the after value of a previous one, in a chain consistent with this client being the only one modifying the directory.

In either of these cases, the client is able to determine whether the directory is being modified by another client. If the comparison indicates that the directory was updated by another client, the name cache associated with the modified directory is purged from the client. If the comparison indicates no modification, the name cache can be updated on the client to reflect the directory operation and the associated timeout extended. The post-operation change value needs to be saved as the basis for future change_info4 comparisons.

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

[10.8.2.](#) Directory Caching

The results of READDIR operations may be used to avoid subsequent READDIR operations. Just as in the cases of attribute and name caching, inconsistencies may arise among the various client caches. To mitigate the effects of these inconsistencies, and given the

context of typical file system APIs, the following rules should be followed:

- o Cached READDIR information for a directory which is not obtained

in a single READDIR operation must always be a consistent snapshot of directory contents. This is determined by using a GETATTR before the first READDIR and after the last of READDIR that contributes to the cache.

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

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

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

[10.9.](#) Directory Delegations

[10.9.1.](#) Introduction to Directory Delegations

Directory caching for the NFSv4.1 protocol, as previously described, is similar to file caching in 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 a 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 sends requests 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 an OPTIONAL 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. It can also simplify the task of determining whether other clients are making changes to the directory when the client itself is making many changes to the directory and changes are not serialized.

Directory delegations allow improved namespace cache consistency to be achieved through delegations and synchronous recalls, in the absence of 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.

10.9.2. Directory Delegation Design

NFSv4.1 introduces the GET_DIR_DELEGATION ([Section 18.39](#)) 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 the directory's 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 NFSv4.1 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, if the client has requested notifications, notify the client holding the delegation of the changes made as a result. This is to avoid any need for 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 (i.e. the changes are made via operations sent though a session associated with the client ID holding the delegation), 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, unless those clients have made provisions to be notified of that sort of modification.

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. If another client removes the directory for which a delegation has been granted, the server will recall the delegation.

[10.9.3.](#) Attributes in Support of Directory Notifications

See [Section 5.11](#) for a description of the attributes associated with directory notifications.

[10.9.4.](#) Directory 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 need not recall the delegation if attributes of an entry within the directory change.

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 delegations for that directory, or recall those already granted. If a client tries to remove the directory for which a delegation has been granted, the server will recall all associated delegations.

The implementation sections for a number of operations describe situations in which notification or delegation recall would be required under some common circumstances. In this regard, a similar set of caveats to those listed in [Section 10.2](#) apply.

- o For CREATE, see [Section 18.4.4](#).
- o For LINK, see [Section 18.9.4](#).
- o For OPEN, see [Section 18.16.4](#).
- o For REMOVE, see [Section 18.25.4](#).
- o For RENAME, see [Section 18.26.4](#).
- o For SETATTR, see [Section 18.30.4](#).

[10.9.5.](#) Directory Delegation Recovery

Recovery from client or server restart 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 goals applies to directories protected by read delegations and notifications. Thus, no provision is made for reclaiming directory delegations in the event of client or server restart. The client can simply establish a directory delegation in the same fashion as was done initially.

11. Multi-Server Namespace

NFSv4.1 supports attributes that allow a namespace to extend beyond the boundaries of a single server. It is RECOMMENDED that clients and servers support construction of such multi-server namespaces. Use of such multi-server namespaces is OPTIONAL however, and for many purposes, single-server namespace are perfectly acceptable. Use of

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

11.1. Location Attributes

NFSv4.1 contains RECOMMENDED attributes that allow file systems on one server to be associated with one or more instances of that file system on other servers. These attributes specify such file system instances by specifying a server address target (either as a DNS name representing one or more IP addresses or as a literal IP address) together with the path of that file system within the associated single-server namespace.

The `fs_locations_info` RECOMMENDED attribute allows specification of one or more file system 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, significant information about the various file system instance choices (e.g. priority for use, writability, currency, etc.). It also includes 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.

[11.2.](#) File System Presence or Absence

A given location in an NFSv4.1 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. Any

reference to it, except to access a small set of attributes useful in determining alternate locations, will result in an error, NFS4ERR_MOVED. Note that if the server ever returns the error NFS4ERR_MOVED, it MUST support the `fs_locations` attribute and SHOULD support the `fs_locations_info` and `fs_status` 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 set of file system(s) designated by the location attributes being the only realization. The name NFS4ERR_MOVED reflects an earlier, more limited conception of its function, but this error will be returned whenever the referenced file system is absent, whether it has moved or not.

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

absent on the current server.

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

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

The RECOMMENDED file system attribute `fs_status` can be used to interrogate the present/absent status of a given file system.

[11.3.](#) 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`.

[11.3.1.](#) 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_status`. 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 REQUIRED, will not be available on an absent file system. In addition to the attributes mentioned above (fs_locations, fs_locations_info, fs_status), 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_policy: 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. This value must be different from any other fsid on the current server and need have no particular relationship to fsids on any particular destination to which the client might be directed.

mounted_on_fileid: For objects at the top of an absent file system this attribute needs to be available. Since the fileid is 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 fs_status, 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_status, the error NFS4ERR_MOVED will result. It differs in that any appearance in the attribute mask of an attribute not supported for an absent file system (and note that this will include some normally REQUIRED attributes), will also cause an NFS4ERR_MOVED

result.

[11.3.2.](#) READDIR and Absent File Systems

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

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

- o If the attribute set requested includes one of the attributes `fs_locations`, `fs_locations_info`, or `fs_status`, then fetching of attributes proceeds normally and no NFS4ERR_MOVED indication is returned, even when the `rdattr_error` attribute is requested.
- o If the attribute set requested does not include one of the attributes `fs_locations`, `fs_locations_info`, or `fs_status`, then if the `rdattr_error` attribute is requested, each directory entry for the root of an absent file system, will report NFS4ERR_MOVED as the value of the `rdattr_error` attribute.
- o If the attribute set requested does not include any of the attributes `fs_locations`, `fs_locations_info`, `fs_status`, or `rdattr_error` then the occurrence of the root of an absent file system within the directory will result in the READDIR failing with an NFS4ERR_MOVED error.
- o The unavailability of an attribute because of a file system's absence, even one that is ordinarily REQUIRED, does not result in any error indication. The set of attributes returned for the root directory of the absent file system in that case is simply restricted to those actually available.

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

When a file system is present and becomes absent, clients can be given the opportunity to have continued access to their data, at an alternate location. In this case, a continued attempt to use the data in the now-absent file system will result in an NFS4ERR_MOVED error and at that point the successor locations (typically only one 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 namespace defined by another server, thus allowing a general multi-server namespace facility. A designation of such a location, in place of an absent file system, is called a "referral".

Because client support for location-related attributes is OPTIONAL, a server may (but is not required to) take action to hide migration and referral events from such clients, by acting as a proxy, for example. The server can determine the presence of client support from the arguments of the EXCHANGE_ID operation (see [Section 18.35.3](#)).

[11.4.1](#). 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 addition to the current file system instance. On first access to a file system, the client should obtain the value of the set of alternate locations by interrogating the fs_locations 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

Internet-Draft

NFSv4.1

September 2008

locations as a way to get continued access to its 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 forms of server clustering in which multiple servers provide alternate ways of accessing the same physical file system. How these different modes of file system transition are represented within the `fs_locations` and `fs_locations_info` attributes and how the client deals with file system transition issues will be discussed in detail below.

Multiple server addresses, whether they are derived from a single entry with a DNS name representing a set of IP addresses, or from multiple entries each with its own server address may correspond to the same actual server. The fact that two addresses correspond to the same server is shown by a common `so_major_id` field within the `eir_server_owner` field returned by `EXCHANGE_ID` (see [Section 18.35.3](#)). For a detailed discussion of how server address targets interact with the determination of server identity specified by the server owner field, see [Section 11.5](#).

[11.4.2](#). 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 the `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. These can be used to access the file system in question at those addresses and when it is 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.

[11.4.3.](#) Referrals

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

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

fs_locations 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 be 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 variables so that different clients are referred to different file systems (with different data contents) based on client attributes such as CPU architecture.

When the fs_locations_info attribute indicates that there are multiple possible targets listed, the relationships among them may be important to the client in selecting the one to use. The same rules specified in [Section 11.4.1](#) defining the appropriate standards for the data propagation, apply to these multiple replicas as well. For example, the client might prefer a writable target on a server that has additional writable replicas to which it subsequently might switch. Note that, as distinguished from the case of replication, there is no need to deal with the case of propagation of updates made by the current client, since the current client has not accessed the file system in question.

Use of multi-server namespaces is enabled by NFSv4.1 but is not required. The use of multi-server namespaces and their scope will depend on the applications used, and system administration preferences.

Multi-server namespaces can be established by a single server providing a large set of referrals to all of the included file systems. Alternatively, a single multi-server namespace may be administratively segmented with separate referral file systems (on separate servers) for each separately-administered portion of the namespace. 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 clients to be directed to different sets of data, so as to adapt to such client characteristics as CPU architecture.

[11.5.](#) Location Entries and Server Identity

As mentioned above, a single location entry may have a server address target in the form of a DNS name which may represent multiple IP addresses, while multiple location entries may have their own server address targets, that reference the same server. Whether two IP addresses designate the same server is indicated by the existence of a common `so_major_id` field within the `eir_server_owner` field returned by `EXCHANGE_ID` (see [Section 18.35.3](#)), subject to further verification, for details of which see [Section 2.10.5](#).

When multiple addresses for the same server exist, the client may assume that for each file system in the namespace of a given server network address, there exist file systems at corresponding namespace locations for each of the other server network addresses. It may do this even in the absence of explicit listing in `fs_locations` 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 addresses are explicitly 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 client's use of these alternate locations.

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

[11.6.](#) 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 continues to see a coherent picture of that user-side file system despite the fact that it contains a number of server-side file systems 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), in the case in which it transitions to that file system as a result of referral, migration, or a transition as a result of replication.

When the client is at such a point, and it needs to ascend to the parent, it must go back to the parent as seen within the multi-server namespace rather issuing a LOOKUPP call to the server, which would result in the parent within that server's single-server namespace. In order to do this, the client needs to remember the filehandles that represent such file system roots, and use these instead of issuing a LOOKUPP to the current server. This will allow the client to present to applications a consistent namespace, where upward navigation and downward navigation are consistent.

Another issue concerns refresh of referral locations. When referrals are used extensively, they may change as server configurations change. It is expected that clients will cache information related to traversing referrals so that future client side requests are

resolved locally without server communication. This is usually rooted in client-side name lookup caching. Clients should periodically purge this data for referral points in order to detect changes in location information. When the `change_policy` 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.

[11.7.](#) Effecting File System Transitions

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

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 11.10](#) for details). When only `fs_locations` is available, default assumptions with regard to such classifications have to be inferred (see [Section 11.9](#) for details).

In cases in which one server is expected to accept opaque values from the client that originated from another server, the servers SHOULD encode the "opaque" values in big endian byte 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 byte order is different from that of other servers cooperating in the replication and migration of the file system.

[11.7.1](#). 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 system 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:

- o The client accesses multiple instances simultaneously, as representing alternate paths to the same data and metadata.
- o 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 system instances in the same location within a multiple single-server namespace as indicated by the `so_major_id` field in the `eir_server_owner` field returned by `EXCHANGE_ID`.

11.7.2. Simultaneous Use and Transparent Transitions

When two file system instances have the same location within their respective single-server namespaces and those two server network addresses designate the same server (as indicated by the same `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. In this case we refer to the transition as "transparent" and the client in transferring access from to the other is acting as it would in the event that communication is interrupted, with a new connection and possibly a new session being established to continue access to the same file system.

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. See [Section 11.10.1](#) for information about the definition of the various use classes, including the `_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_`, `_write-verifier_`, `_change_`, `_readdir_` classes, indicates a serious problem and the client, if it allows transition to the file system instance at all, must not treat this as a transparent transition. The server SHOULD NOT indicate that these instances belong to different `_handle_`, `_fileid_`, `_write-verifier_`, `_change_`, `_readdir_` classes, whether the

Internet-Draft

NFSv4.1

September 2008

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_`, `_write-verifier_`, `_change_`, `_readdir_` classes of the two file system instances and whether the two servers address in question have the same `eir_server_scope` value as reported by `EXCHANGE_ID`.

[11.7.2.1](#). Simultaneous Use of File System Instances

When the conditions in [Section 11.7.2](#) 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-network-address entries for file systems instances at the distinct network addresses. This includes the case in which the `fs_locations_info` attribute is unavailable. In this case, the fact that the two server addresses connect to the same server (as indicated by the two addresses sharing the same the `so_major_id` value and subsequently confirmed as described in [Section 2.10.5](#)) justifies simultaneous use and there is no `fs_locations_info` attribute information contradicting that.
- 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 network 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.

[11.7.2.2](#). Transparent File System Transitions

When the conditions in [Section 11.7.2.1](#) hold and the `fs_locations_info` attribute explicitly shows the file system instances for these distinct network 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 of 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:

- o If filehandles are persistent they stay the same. If filehandles are volatile, they either stay the same, or if they expire, the reason for expiration is not due to the file system transition.
- o Fileid values do not change across the transition.
- o The file system will have the same `fsid` in both the old and new locations.
- o 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.
- o Client and state identifiers 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. Any such staleness is an indication that the server should be considered to have restarted and is reported as discussed in [Section 8.4.2](#).
- o Write verifiers are presumed to retain their validity and can be used to compare with verifiers returned by `COMMIT` on the new

server, with the expectation that if COMMIT on the new server returns an identical verifier, then that server has all of the data unstably written to the original server and has committed it to stable storage as requested.

- o Readdir cookies are presumed to retain their validity and can be presented to subsequent READDIR requests together with the readdir verifier with which they are associated. When the verifier is accepted as valid, the cookie will continue the READDIR operation so that the entire directory can be obtained by the client.

[11.7.3.](#) 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 filehandles remain valid after the file system transition, while volatile filehandles (excluding those that are only volatile due to the `FH4_VOL_MIGRATION` bit) are subject to expiration on the target server.

[11.7.4.](#) Fileids and File System Transitions

In NFSv4.0, the issue of continuity of fileids 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,

fileids 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 transition mechanisms adopted by the server. Note that although fileid is not a REQUIRED attribute, many servers support fileids and many clients provide API's that depend on fileids.

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 fileids 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 a file system between vendors where assigning the same index to a given file may not be possible. Note here that a fileid is not required to be useful to find the file in question, only that it is unique within the given file system. 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 fileids across a file system transition is represented by specifying whether the file systems in question are of the same `_fileid_` class.

Note that when consistent fileids do not exist across a transition (either because there is no continuity of fileids or because fileid is not a supported attribute on one of instances involved), and there

are no reliable filehandles across a transition event (either because there is no filehandle continuity or because the filehandles are volatile), the client is in a position where it cannot verify that files it was accessing before the transition are the same objects. It is forced to assume that no object has been renamed, and, unless there are guarantees that provide this (e.g. the file system is read-only), problems for applications may occur. Therefore, use of such configurations should be limited to situations where the problems that this may cause can be tolerated.

[11.7.5.](#) Fsid and File System Transitions

Since fsids are generally only unique within a per-server basis, it is likely that they will change during a file system transition. One exception is the case of transparent transitions, but in that case we have multiple network addresses that are defined as the same server (as specified by a common value of the `so_major_id` field of `eir_server_owner`). Clients should not make the fsids received from the server visible to applications 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.

Although normally, a single source file system will transition to a single target file system, there is a provision for splitting a single source file system into multiple target file systems, by

specifying the `FSLI4F_MULTI_FS` flag.

[11.7.5.1.](#) File System Splitting

When a file system transition is made and the `fs_locations_info` indicates that the file system in question may be split into multiple file systems (via the `FSLI4F_MULTI_FS` flag), the client SHOULD do GETATTRs to determine the fsid attribute on all known objects within the file system undergoing transition to determine the new file system boundaries.

Clients may maintain the fsids passed to existing applications by mapping all of the fsids for the descendant file systems to the common fsid used for the original file system.

Splitting a file system may be done on a transition between file systems of the same `_fileid_` class, since the fact that fileids are unique within the source file system ensure they will be unique in each of the target file systems.

[11.7.6.](#) 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 retrieved on refetch no conclusions can be 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.

[11.7.7.](#) 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 client IDs. 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 client IDs to the point that they will reject client IDs that refer to state they do not know about. See [Section 2.10.4](#) for more information about the use of server scope.

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 is 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` (as modified by the validation procedures described in [Section 2.10.4](#)) 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 occurs. 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 stateids associated with that client ID for the old file system instance in connection with that same client ID in connection with the transitioned file system instance. If the client in question already had a client ID on the target system, it may interrogate the stateid values from the source system under that new client ID, with the assurance that if they are accepted as valid, then they represent validly transferred lock state for the source file system, transferred to the target server.

When the two servers belong to the same server scope, it does not 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 its current client ID when establishing communication with the new server and the new server will either recognize the 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 identifier space so as to recognize (and reject as stale) each other's stateids and client IDs. 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 system 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 locks if allowed by the server. In this case, old stateids and client IDs should not be presented to the new server since there is no assurance that they will not conflict with IDs valid on that server. Note that in this case lock reclaim may be attempted even when the servers involved in the transfer have different server scope values (see [Section 8.4.2.1](#) for the contrary case of reclaim after server reboot. Servers with different server scope values may co-operate to allow reclaim for locks associated with the transfer of a filesystem even if they do not co-operate sufficiently to share a server scope.

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 restart (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). Unless the destination server can guarantee that locks will not be incorrectly granted, the destination server should not allow lock reclaims and avoid establishing a grace period.

Once all locks have been reclaimed, or there were no locks to reclaim, the client indicates that there are no more reclaims to be done for the file system in question by issuing a RECLAIM_COMPLETE operation with the `rca_one_fs` parameter set to true. Once this has been done, non-reclaim locking operations may be done, and any subsequent request to do reclaims will be rejected with the error NFS4ERR_NO_GRACE.

Information about client identity 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 its LOCK or OPEN request denied due to a conflicting lock.

The consequences of having no facilities available to reclaim locks on the new server will depend on the type of environment. 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 its original lock had been revoked. Note that when the lock is granted, the client cannot assume that no conflicting lock 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.

[11.7.7.1](#). 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 the lease associated with 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 where there is locking state 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_STATUS_LEASE_MOVED, if responsibility for any of the locking state renewed has been transferred to a new server. This will continue until the client receives an NFS4ERR_MOVED error for each of the file systems for which there has been locking state relocation.

When a client receives an SEQ4_STATUS_LEASE_MOVED indication, it should perform an operation on each file system associated with the server where there is locking state for the current client associated with the file system in question. The client may choose to reference all file systems in the interests of simplicity but what is important is that it must reference all file systems for which there was locking state where that state moved. Once the client receives an

NFS4ERR_MOVED error for each file system, the SEQ4_STATUS_LEASE_MOVED indication is cleared. The client can terminate the process of

checking file systems once this indication is cleared (but only if the client has received a reply for all outstanding SEQUENCE requests on all sessions it has with the server), since there are no others for which locking state has moved.

A client may use GETATTR of the `fs_status` (or `fs_locations_info`) attribute on all of the file systems to get absence indications in a single (or a few) request(s), since absent file systems will not cause an error in this context. However, it still must do an operation which receives NFS4ERR_MOVED on each file system, in order to clear the SEQ4_STATUS_LEASE_MOVED indication is cleared.

Once the set of file systems with transferred locking state has been determined, 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 NFS4ERR_STALE_CLIENTID from the new server, as described above, and the client can then reclaim locks as is done in the event of server failure.

[11.7.7.2.](#) 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 shown as having 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.

[11.7.8.](#) 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 `_write-verifier_` class, write verifiers returned from one system may be compared to those returned by the other and superfluous writes avoided.

When two file systems belong to different `_write-verifier_` classes, any verifier generated by one must not be compared to one provided by the other. Instead, it should be treated as not equal even when the values are identical.

[11.7.9.](#) Readdir Cookies and Verifiers and File System Transitions

In a file system transition, the two file systems may be consistent in their handling of READDIR cookies and verifiers. When this is the case, and the two file systems belong to the same `_readdir_` class, READDIR cookies and verifiers from one system may be recognized by the other and READDIR operations started on one server may be validly continued on the other, simply by presenting the cookie and verifier returned by a READDIR operation done on the first file system to the second.

When two file systems belong to different `_readdir_` classes, any READDIR cookie and verifier generated by one is not valid on the second, and must not be presented to that server by the client. The client should act as if the verifier was rejected.

[11.7.10.](#) File System Data and File System Transitions

When multiple replicas exist and are used simultaneously or in

succession by a client, applications using them will normally expect that they contain data the same data or data which is consistent with the normal sorts of changes that are made by other clients updating the data of the file system. (with metadata being the same to the degree indicated by the `fs_locations_info` attribute). However, when multiple file systems are presented as replicas of one another, the precise relationship between the data of one and the data of another is not, as a general matter, specified by the NFSv4.1 protocol. It is quite possible to present as replicas file systems where the data of those file systems is sufficiently different that some applications have problems dealing with the transition between replicas. The namespace will typically be constructed so that applications can choose an appropriate level of support, so that in one position in the namespace a varied set of replicas will be listed while in another only those that are up-to-date may be considered replicas. The protocol does define three special cases of the

relationship among replicas to be specified by the server and relied upon by clients:

- o When multiple server addresses correspond to the same actual server, as indicated by a common `so_major_id` field within the `eir_server_owner` field returned by `EXCHANGE_ID`, the client may depend on the fact that changes to data, metadata, or locks made on one file system are immediately reflected on others.
- o When multiple replicas exist and are used simultaneously by a client (see the `FSLIB4_CLSIMUL` definition within `fs_locations_info`), they must designate the same data. 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 requester or the visibility of that change on any of the associated replicas. This allows a client to use these replicas simultaneously without any special adaptation to the fact that there are multiple replicas. In this case, locks, whether shared or byte-range, and delegations obtained on one replica are immediately reflected on all replicas, even though these locks will be managed under a set of client IDs.
- o When one replica is designated as the successor instance to another existing instance after return `NFS4ERR_MOVED` (i.e. the case of migration), the client may depend on the fact that all

changes securely made to data (uncommitted writes are dealt with in [Section 11.7.8](#)) on the original instance are made to the successor image.

- o Where a file system is not writable but represents a read-only copy (possibly periodically updated) of a writable file system, clients have similar requirements with regard to the propagation of updates. They may need a guarantee 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, in order 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 11.11](#)). Since these file systems are presumed not to be suitable for simultaneous use, there is no specification of how locking is handled and it generally will be the case that locks obtained on one file system will be separate from those on others. Since these are going to be read-only file systems, this is not expected to pose an issue for clients or applications.

[11.8.](#) 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 a different server, even though it retains its logical position within the original namespace. Referrals differ from migration events in that they happen only when the client has not previously referenced the file system in question (so there is nothing to transition). Referrals can only come into effect when an absent file system is encountered at its root.

The examples given in the sections below are somewhat artificial in that an actual client will not typically do a multi-component 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.

[11.8.1.](#) Referral Example (LOOKUP)

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

- o PUTROOTFH
- o LOOKUP "this"
- o LOOKUP "is"
- o LOOKUP "the"
- o LOOKUP "path"
- o GETFH
- o GETATTR fsid,fileid,size,time_modify

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

- o PUTROOTFH --> NFS_OK. The current fh is now the root of the pseudo-fs.
- o LOOKUP "this" --> NFS_OK. The current fh is for `/this` and is within the pseudo-fs.
- o LOOKUP "is" --> NFS_OK. The current fh is for `/this/is` and is within the pseudo-fs.
- o LOOKUP "the" --> NFS_OK. The current fh is for `/this/is/the` and is within the pseudo-fs.

- o LOOKUP "path" --> NFS_OK. The current fh is for /this/is/the/path and is within a new, absent file system, but ... the client will never see the value of that fh.
- o GETFH --> NFS4ERR_MOVED. Fails because current fh is in an absent file system at the start of the operation and the spec makes no exception for GETFH.
- o GETATTR fsid,fileid,size,time_modify. Not executed because the failure of the GETFH stops processing of the COMPOUND.

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

In order to get the necessary information, let us re-send the chain of LOOKUPS with GETFHs and GETATTRs to at least get the fsids so we can be sure where the appropriate file system boundaries are. The client could choose to get fs_locations_info at the same time but in most cases the client will have a good guess as to where file system boundaries are (because of where and where not NFS4ERR_MOVED was received) 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 file system boundaries are. The fsid will be that for the pseudo-fs in this example, so no boundary.

OP05: GETFH --> NFS_OK

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

OP06: LOOKUP "is" --> NFS_OK

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

OP07: GETATTR(fsid) --> NFS_OK

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

OP08: GETFH --> NFS_OK

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

OP09: LOOKUP "the" --> NFS_OK

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

OP10: GETATTR(fsid) --> NFS_OK

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

OP11: GETFH --> NFS_OK

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

OP12: LOOKUP "path" --> NFS_OK

- Current fh is for /this/is/the/path and is within a new, absent file system, but ...
- The client will never see the value of that fh

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

- We are getting the fsid to know where the file system boundaries are. In this operation the fsid will be different than that of the parent directory (which in turn was retrieved in OP10). Note that the fsid we are given will not necessarily be preserved at the new location. That fsid might be different and in fact the fsid we have for this file system might be a valid fsid of a different file system on that new server.
- In this particular case, we are pretty sure anyway that what has moved is /this/is/the/path rather than /this/is/the since we have the fsid of the latter and it is that of the pseudo-fs, which presumably cannot move. However, in other examples, we might not have this kind of information to rely on (e.g. /this/is/the might be a non-pseudo file system separate from /this/is/the/path), so we need to have another reliable source information on the boundary of the file system 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 file system at the start of the operation and the spec makes no exception for GETFH. Note that this means the server will never send the client a filehandle from within an absent file system.

Given the above, the client knows where the root of the absent file

system is (/this/is/the/path), by noting where the change of fsid

occurred (between "the" and "path"). The `fs_locations_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.

[11.8.2](#). Referral Example (READDIR)

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

Suppose such a directory is read as follows:

- o PUTROOTFH
- o LOOKUP "this"
- o LOOKUP "is"
- o LOOKUP "the"
- o READDIR (fsid, size, time_modify, mounted_on_fileid)

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

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

- o LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.
- o LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.
- o READDIR (fsid, size, time_modify, mounted_on_fileid) --> NFS4ERR_MOVED. Note that the same error would have been returned

if /this/is/the had migrated, when in fact it is because the directory contains the root of an absent file system.

So now suppose that we re-send with rdattn_error:

- o PUTROOTFH
- o LOOKUP "this"
- o LOOKUP "is"
- o LOOKUP "the"
- o READDIR (rdattr_error, fsid, size, time_modify, mounted_on_fileid)

The results will be:

- o PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.
- o LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.
- o LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.
- o LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.
- o READDIR (rdattr_error, fsid, size, time_modify, mounted_on_fileid) --> NFS_OK. The attributes for directory entry with the component named "path" will only contain rdattr_error with the value NFS4ERR_MOVED, together with an fsid value and a value for

mounted_on_fileid.

So suppose we do another READDIR to get fs_locations_info (although we could have used a GETATTR directly, as in [Section 11.8.1](#)).

- o PUTROOTFH
- o LOOKUP "this"
- o LOOKUP "is"
- o LOOKUP "the"
- o READDIR (rdattr_error, fs_locations_info, mounted_on_fileid, fsid, size, time_modify)

The results would be:

- o PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.
- o LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.
- o LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.
- o LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.
- o READDIR (rdattr_error, fs_locations_info, mounted_on_fileid, fsid, size, time_modify) --> NFS_OK. The attributes will be as shown below.

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

- o rdattr_error (value: NFS_OK)
- o fs_locations_info
- o mounted_on_fileid (value: unique fileid within referring file)

system)

- o fsid (value: unique value within referring server)

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

[11.9](#). The Attribute fs_locations

The fs_locations attribute is structured in the following way:

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

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

```
};
```

The fs_location4 data type is used to represent the location of a file system by providing a server name and the path to the root of the file system within that server's namespace. When a set of servers have corresponding file systems at the same path within their namespaces, an array of server names may be provided. An entry in the server array is a UTF-8 string and represents one of a traditional DNS host name, IPv4 address, or IPv6 address, or a zero-length string. An IPv4 or IPv6 address is represented as a universal address (see [Section 3.3.9](#) and [13]), minus the netid, and either with or without the trailing ".p1.p2" suffix that represents the port number. If the suffix is omitted, then the default port, 2049, SHOULD be assumed. A zero-length string SHOULD be used to indicate the current address being used for the RPC call. It is not a requirement that all servers that share the same rootpath be listed in one fs_location4 instance. The array of server names is provided for convenience. Servers that share the same rootpath may also be listed in separate fs_location4 entries in the fs_locations

attribute.

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

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

As an example, suppose there is a replicated file system located at two servers (`servA` and `servB`). At `servA`, the file system is located at path `"/a/b/c"`. At `servB` the file system is located at path `"/x/y/z"`. If the client were to obtain the `fs_locations` value for the directory at `"/a/b/c/d"`, it might not necessarily know that the file system's root is located in `servA`'s namespace at `"/a/b/c"`. When the client switches to `servB`, it will need to determine that the directory it first referenced at `servA` is now represented by the path `"/x/y/z/d"` on `servB`. To facilitate this, the `fs_locations` attribute

provided by `servA` would have 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.

Note that: there is no requirement that the number of components in each rootpath be the same; there is no relation between the number of components in rootpath or `fs_root`; and the none of the components in each rootpath and `fs_root` have to be the same. In the above example, we could have had a third element in the locations array, with server

equal to "servC", and rootpath equal to "/I/II", and a fourth element in locations with server equal to "servD", and rootpath equal to "/aleph/beth/gimel/dalet/he".

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

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

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

Since the 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 of file system transitions depends on the reasons for the transition:

- o When the transition is due to migration, that is the client was directed to new file system after receiving an `NFS4ERR_MOVED` error, the target should be treated as being of the same `_write-verifier_class` as the source.
- o When the transition is due to failover to another replica, that is, the client selected another replica without receiving and `NFS4ERR_MOVED` error, the target should be treated as being of a different `_write-verifier_class` from the source.

The specific choices reflect typical implementation patterns for failover and controlled migration respectively. Since other choices are possible and useful, this information is better obtained by using `fs_locations_info`. When a server implementation needs to communicate other choices, it MUST support the `fs_locations_info` attribute.

See [Section 21](#) for a discussion on the recommendations for the security flavor to be used by any `GETATTR` operation that requests the "fs_locations" attribute.

[11.10](#). 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 to 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`:

- o 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).
- o File System identity information which indicates when multiple replicas, from the client's 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.
- o 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.
- o Server-derived preference information for replicas, which can be used to implement load-balancing while giving the client the entire file system list to be used in case the primary fails.

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 file system 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`) contains 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. This per-server-replica information includes a nominally opaque array, `fls_info`, in which specific pieces of information are located at the specific indices listed below.

The attribute will always contains at least a single `fs_locations_server` entry. Typically, this will be an entry with the `FS4LIGF_CUR_REQ` flag set, although in the case of a referral there will be no entry with that flag set.

It should be noted that `fs_locations_info` attributes returned by servers for various replicas may differ for various reasons. One server may know about a set of replicas that are not know to other servers. Further, compatibility attributes may differ. Filehandles might be of the same class going from replica A to replica B but not

Internet-Draft

NFSv4.1

September 2008

going in the reverse direction. This might happen because the filehandles are the same but replica B's server implementation might not have provision to note and report that equivalence.

The `fs_locations_info` attribute consists of a root pathname (`fli_fs_root`, just like `fs_root` in the `fs_locations` attribute), together with an array of `fs_location_item4` structures. The `fs_location_item4` structures in turn consist of a root pathname (`fli_rootpath`) together with an array (`fli_entries`) of elements of data type `fs_locations_server4`, all defined as follows.

```
/*
 * Defines an individual server replica
 */
struct fs_locations_server4 {
    int32_t      fls_currency;
    opaque       fls_info<>;
    utf8str_cis  fls_server;
};

/*
 * Byte indices of items within
 * fls_info: flag fields, class numbers,
 * bytes indicating ranks and orders.
 */
const FSLI4BX_GFLAGS          = 0;
const FSLI4BX_TFLAGS          = 1;

const FSLI4BX_CLSIMUL         = 2;
const FSLI4BX_CLHANDLE        = 3;
const FSLI4BX_CLFILEID        = 4;
const FSLI4BX_CLWRITEEVER     = 5;
const FSLI4BX_CLCHANGE        = 6;
const FSLI4BX_CLREADDIR       = 7;

const FSLI4BX_READRANK        = 8;
const FSLI4BX_WRITERANK       = 9;
const FSLI4BX_READORDER       = 10;
const FSLI4BX_WRITEORDER      = 11;

/*
 * Bits defined within the general flag byte.
```

```

*/
const FSLI4GF_WRITABLE          = 0x01;
const FSLI4GF_CUR_REQ          = 0x02;
const FSLI4GF_ABSENT           = 0x04;
const FSLI4GF_GOING            = 0x08;
const FSLI4GF_SPLIT            = 0x10;

```

```

/*
 * Bits defined within the transport flag byte.
 */
const FSLI4TF_RDMA              = 0x01;

/*
 * Defines a set of replicas sharing
 * a common value of the root path
 * with in the corresponding
 * single-server namespaces.
 */
struct fs_locations_item4 {
    fs_locations_server4      fli_entries<>;
    pathname4                 fli_rootpath;
};

/*
 * Defines the overall structure of
 * the fs_locations_info attribute.
 */
struct fs_locations_info4 {
    uint32_t                  fli_flags;
    int32_t                   fli_valid_for;
    pathname4                 fli_fs_root;
    fs_locations_item4        fli_items<>;
};

/*
 * Flag bits in fli_flags.
 */
const FSLI4IF_VAR_SUB         = 0x00000001;

typedef fs_locations_info4 fattr4_fs_locations_info;

```

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 previous location of an absent file system (the one just referenced) and its successor location. Servers are strongly urged to support this attribute on all file systems if they support it on any file system.

The data presented in the `fs_locations_info` attribute may be obtained by the server in any number of ways, including specification by the administrator or by current protocols for transferring data among

replicas and protocols not yet developed. NFSv4.1 only defines how this information is presented by the server to the client.

[11.10.1](#). The `fs_locations_server4` Structure

The `fs_locations_server4` structure consists of the following items:

- o An indication of file system up-to-date-ness (`fls_currency`) in terms of approximate seconds before the present. This value is relative to the master copy. 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 the copy of the data would be expected to be behind the most up-to-date copy.
- o A counted array of one-byte 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.
- o The server string (`fls_server`). For the case of the replica currently being accessed (via `GETATTR`), a zero-length string MAY be used to indicate the current address being used for the RPC

call. The `fls_server` field can also be an IPv4 or IPv6 address, formatted the same way as an IPv4 or IPv6 address in the "server" field of the `fs_location4` data type (see [Section 11.9](#)).

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.

- o 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.
- o 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.

- o 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-byte numeric values, even though none are currently defined. If extensions are made via standards-track RFC's, multi-byte quantities will be encoded as a range of bytes with a range of indices with the byte interpreted in big endian byte order. Further any such index assignments are constrained so that the relevant quantities will not cross XDR word boundaries.

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:

- o Two 8-bit flag fields, one devoted to general file-system characteristics and a second reserved for transport-related capabilities.
- o Six 8-bit class values which define various file system equivalence classes as explained below.
- o Four 8-bit priority values which govern file system selection as explained below.

The general file system characteristics flag (at byte index FSLI4BX_GFLAGS) has the following bits defined within it:

- o FSLI4GF_WRITABLE indicates that this file system 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, and is defined as of the same simultaneous use class (as specified by the value at index FSLI4BX_CLSIMUL) to which the client was previously writing, then it must incorporate within its data any committed write made on the source file system instance. See [Section 11.7.8](#) which discusses the write-verifier class. 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 which select a migration or replication target based on it and then find themselves unable to write.

- o 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.
- o 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_status 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 has subsequently been migrated.

- o 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.

When this flag is seen as part of a transition into a new file system, a client might choose to transfer immediately to another replica, or it may reference the current file system and only transition when a migration event occurs. Similarly, when this flag appears as a replica in the referral, clients would likely to avoid being referred to this instance whenever there is another choice.

- o 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 file system before migration will be on different ones after. Note that FSLI4GF_SPLIT is not incompatible with the file systems belonging to the same `_fileid_` class since, if one has a set of fileids that are unique within a file system, each subset assigned to a smaller file system after migration would not have any conflicts internal to that file system.

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 file system boundaries. Note that in the event of a referral, there will not be any such files and so these action will not be performed.

Instead, a reference to a portion of the original file system now split off into other file systems will encounter an fsid change and possibly a further referral.

Once the client recognizes that one file system has been split into two, it can prevent the disruption of running applications by presenting the two file systems as a single one until a convenient point to recognize the transition, such as a restart. This would require a mapping 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 file systems will not conflict. Providing non-conflicting fileids for newly-created files on the split file systems is the responsibility of the server (or servers working in concert). The server can encode filehandles such that filehandles generated before the split event can be discerned from those generated after the split, allowing the server to determine when the need for emulating two file systems as one is over.

Although it is possible for this flag to be present in the event of referral, it would generally be of little interest to the client, since the client is not expected to have information regarding the current contents of the absent file system.

The transport-flag field (at byte index FSLI4BX_TFLAGS) contains the following bits related to the transport capabilities of the specific file system.

- o 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 11.7](#) for details about how this information is to be used. Servers may assign these values as they wish, so long as file system instances that share the same value have the specified relationship to one another, conversely file systems which have the specified relationship to one another share a common class value. As each instance entry is added, the relationships of this instance to previously entered instances can be consulted and if one is found that bears the specified relationship, that entry's class value can be copied to the new entry. When no such previous entry exists, a new value for that byte index, not previously used can be selected, most likely by increment the value of the last class value assigned for that index.

- o The field with byte index FSLI4BX_CLSIMUL defines the simultaneous-use class for the file system.
- o The field with byte index FSLI4BX_CLHANDLE defines the handle class for the file system.
- o The field with byte index FSLI4BX_CLFILEID defines the fileid class for the file system.
- o The field with byte index FSLI4BX_CLWRITEVER defines the write-verifier class for the file system.
- o The field with byte index FSLI4BX_CLCHANGE defines the change class for the file system.
- o The field with byte index FSLI4BX_CLREADDIR defines the readdir 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. Because specifying a rank will override client preferences, servers

Internet-Draft

NFSv4.1

September 2008

should be conservative about using this mechanism, particularly when the environment is one in which client communication characteristics are not tightly controlled and visible to the server.

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.

- o The field at byte index FSLI4BX_READRANK gives the rank value to be used for read-only access.
- o The field at byte index FSLI4BX_READORDER gives the order value to be used for read-only access.
- o The field at byte index FSLI4BX_WRITERANK gives the rank value to be used for writable access.
- o The field at byte index FSLI4BX_WRITEORDER 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.

[11.10.2](#). The fs_locations_info4 Structure

The fs_locations_info4 structure, encoding the fs_locations_info attribute, contains the following:

- o 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.

- o 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 in the `fs_locations4` structure.

- o 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 `NFS4ERR_MOVED`.
- o 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. See [Section 11.10.3](#) for an explanation of variable substitution.

[11.10.3](#). 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_location4 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 `${ietf.org:OS_VERSION}` is used to denote the operating system version and thus the specific details of versioned interfaces for which code might be compiled. This specification does not limit the acceptable values (except that they must be valid UTF-8 strings). However, combinations of numbers and letters with interspersed dots would be expected to be used in line with industry practice, with the details of the version format depending on the specific value of the variable `${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.

Because namespace administration is affected by the values selected to substitute for various variables, clients should provide convenient means of determining what variable substitutions a client will implement, as well as, where appropriate, providing means to control the substitutions to be used. The exact means by which this will be done is outside the scope of this specification.

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 `FSLI4IF_VAR_SUB` flag within the associated `fs_locations_info4` structure is set or not.

[11.11](#). 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_UPDATED = 2,
    STATUS4_VERSIONED = 3,
    STATUS4_WRITABLE = 4,
    STATUS4_REFERRAL = 5
};

struct fs4_status {
    bool                fss_absent;
    fs4_status_type     fss_type;
    utf8str_cs          fss_source;
    utf8str_cs          fss_current;
    int32_t             fss_age;
    nfstime4            fss_version;
```

};

The boolean `fss_absent` indicates whether the file system is currently absent. This value will be set if the file system was previously present and becomes absent, or if the file system has never been present and the type is `STATUS4_REFERRAL`. When this boolean is set and the type is not `STATUS4_REFERRAL`, the remaining information in the `fs4_status` reflects that last valid when the file system was present.

The `fss_type` field 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. When `fss_absent` is set, and the file system was previously present, the value of `fss_type` reflected is that when the file was last present. Five values are distinguished:

- o `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 cache aggressively.
- o `STATUS4_VERSIONED` which indicates that the image, like the `STATUS4_UPDATED` case, is updated externally, 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.

- o `STATUS4_UPDATED` which indicates an image that cannot be updated by the user writing to it but may be changed externally, 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_REFERRAL which indicates that the file system is question is absent and has never been present on this server.

Note that in the STATUS4_UPDATED and STATUS4_VERSIONED cases, the server is responsible for the appropriate handling of locks that are inconsistent with external changes to delegations. If a server gives out delegations, they SHOULD be recalled before an inconsistent change made to data, and MUST be revoked if this is not possible. Similarly, if an open is inconsistent with data that is changed (the open denies WRITE and the data is changed), that lock SHOULD be considered administratively revoked.

The opaque strings fss_source and fss_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 diagnostic 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, clones 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 fss_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, it is desirable to associate with it data regarding how the file system was created, where it was created, by whom, etc. Making this information available in this attribute in a human-readable string form will be helpful for applications and system administrators and also serves to make it available when the original file system is used to make subsequent copies.

The opaque string `fss_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 field `fss_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 11.10](#)) 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 value in `fss_age` gives a bound on how out of date that data actually is. Negative values imply that 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. Using this value, the client may be able to decide that a data copy is too old, so that it may search for a newer version to use.

The `fss_version` field provides a version identification, in the form of a time value, such that successive versions always have later time values. When the `fs_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 `fss_source` and `fss_current`.

When `fss_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 the fs_status attribute 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 does not have an fss_type of STATUS4_VERSIONED or whose fss_version field is earlier than the last one obtained from the predecessor file system instance.

[12.](#) Parallel NFS (pNFS)

[12.1.](#) Introduction

pNFS is an OPTIONAL feature within NFSv4.1; the pNFS feature set allows direct client access to the storage devices containing 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:

Internet-Draft

NFSv4.1

September 2008

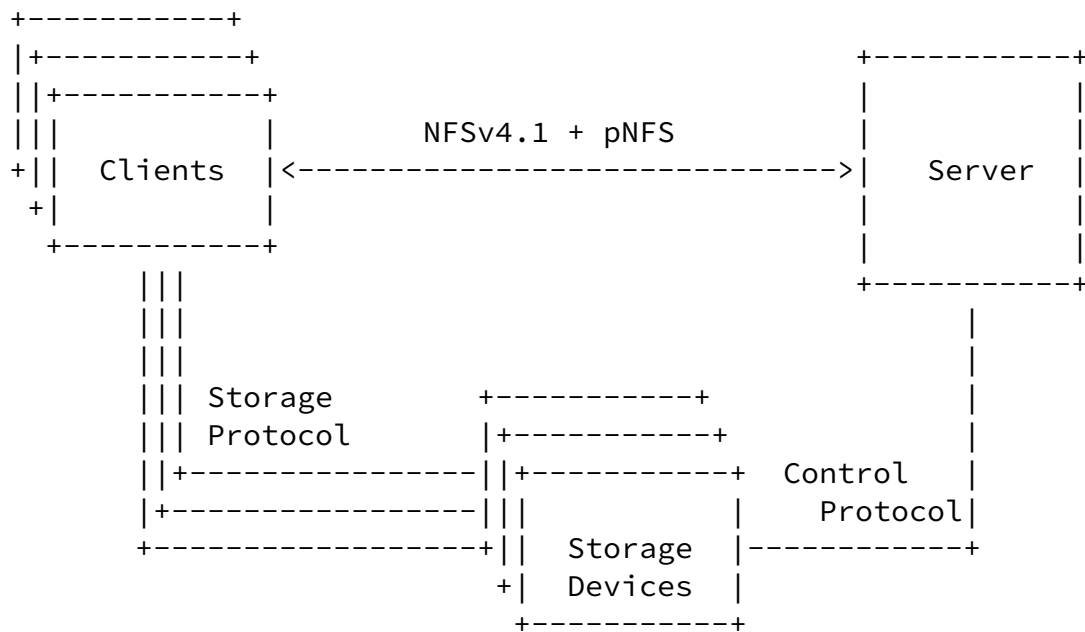


Figure 1

In this model, the clients, server, and storage devices are responsible for managing file access. This is in contrast to NFSv4 without pNFS where it is primarily the server's responsibility; some of this responsibility may be delegated to the client under strictly specified conditions.

pNFS takes the form of OPTIONAL operations that manage protocol objects called 'layouts' which contain a byte-range and storage location information. The layout is managed in a similar fashion as NFSv4.1 data delegations. For example, the layout is leased, recallable and revocable. However, layouts are distinct abstractions and are manipulated with new operations. When a client holds a layout, it is granted the ability to directly access the byte-range at the storage location specified in the layout.

There are interactions between layouts and other NFSv4.1 abstractions such as data delegations and byte-range locking. Delegation issues are discussed in [Section 12.5.5](#). Byte range locking issues are discussed in [Section 12.2.9](#) and [Section 12.5.1](#).

The NFSv4.1 pNFS feature has been structured to allow for a variety of storage protocols to be defined and used. As noted in the diagram above, the storage protocol is the method used by the client to store and retrieve data directly from the storage devices. The NFSv4.1 protocol directly defines one storage protocol, the NFSv4.1 storage type, and its use.

Examples of other storage protocols that could be used with NFSv4.1's

pNFS are:

- o Block/volume protocols such as iSCSI ([\[37\]](#)), and FCP ([\[38\]](#)). 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.
- o Object protocols such as OSD over iSCSI or Fibre Channel [\[39\]](#).
- o Other storage protocols, including PVFS and other file systems that are in use in HPC environments.

It is possible that various storage protocols are available to both client and server and it may be possible that a client and server do not have a matching storage protocol available to them. Because of this, the pNFS server MUST support normal NFSv4.1 access to any file accessible by the pNFS feature; this will allow for continued interoperability between an NFSv4.1 client and server.

[12.2.](#) pNFS Definitions

NFSv4.1's pNFS feature provides parallel data access to a file system that stripes its content across multiple storage servers. The first instantiation of pNFS, as part of NFSv4.1, separates the file system protocol processing into two parts: metadata processing and data processing. Data consist of the contents of regular files which are striped across storage servers. Data striping occurs in at least two ways: on a file-by-file basis, and within sufficiently large files, on a block-by-block basis. In contrast, striped access to metadata by pNFS clients is not provided in NFSv4.1, even though the file system back end of a pNFS server might stripe metadata. Metadata

consist of everything else, including the contents of non-regular files (e.g. directories); see [Section 12.2.1](#). The metadata functionality is implemented by an NFSv4.1 server that supports pNFS and the operations described in ([Section 18](#)); such a server is called a metadata server ([Section 12.2.2](#)).

The data functionality is implemented by one or more storage devices, each of which are accessed by the client via a storage protocol. A subset (defined in [Section 13.6](#)) of NFSv4.1 is one such storage protocol. New terms are introduced to the NFSv4.1 nomenclature and existing terms are clarified to allow for the description of the pNFS feature.

[12.2.1](#). Metadata

Information about a file system object, such as its name, location within the namespace, owner, ACL and other attributes. Metadata may also include storage location information and this will vary based on the underlying storage mechanism that is used.

[12.2.2](#). Metadata Server

An NFSv4.1 server which supports the pNFS feature. A variety of architectural choices exists for the metadata server and its use of file system information held at the server. Some servers may contain metadata only for file objects residing at the metadata server while the file data resides on associated storage devices. Other metadata servers may hold both metadata and a varying degree of file data.

[12.2.3](#). pNFS Client

An NFSv4.1 client that supports pNFS operations and supports at least one storage protocol for performing I/O to storage devices.

[12.2.4](#). Storage Device

A storage device stores a regular file's data, but leaves metadata management 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.](#) Storage Protocol

A storage protocol is the protocol used between the pNFS client and the storage device to access the file data.

[12.2.6.](#) 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 the NFSv4.1 protocol. Such control protocols would be used to control activities such as the allocation and deallocation of storage and the management of state required by the storage devices to perform client access control.

A particular control protocol is not REQUIRED by NFSv4.1 but requirements are placed on the control protocol for maintaining attributes like modify time, the change attribute, and the end-of-file (EOF) position.

[12.2.7.](#) 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 3.3.13](#)). The layout type allows for variants to handle different storage protocols, such as those associated with block/volume [\[30\]](#), object [\[29\]](#), 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 MAY be used for internal testing or experimentation.

As an example, the organization of the file layout type could be an array of tuples (e.g., device ID, filehandle), 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 <device ID, block_number, block count> along with information

about block size and the associated file offset of the block number. An object layout might be an array of tuples <device ID, object ID> and an additional structure (i.e., the aggregation map) that defines how the logical byte sequence of the file data is serialized into the different objects. Note that the actual layouts are typically more complex than these simple expository examples.

Requests for pNFS-related operations will often specify a layout type. Examples of such operations are GETDEVICEINFO and LAYOUTGET. The response for these operations will include structures such as device_addr4 or a layout4, each of which includes a layout type within it. The layout type sent by the server MUST always be the same one requested by the client. When a server sends a response that includes a different layout type, the client SHOULD ignore the response and behave as if the server had returned an error response.

[12.2.8.](#) Layout

A layout defines how a file's data is organized on one or more storage devices. There are many potential layout types; each of the layout types are differentiated by the storage protocol used to access data and in the aggregation scheme that lays out the file data on the underlying storage devices. A layout is precisely identified by the following tuple: <client ID, filehandle, layout type, iomode, range>; where filehandle refers to the filehandle of the file on the metadata server.

It is important to define when layouts overlap and/or conflict with each other. For two layouts with overlapping byte ranges to actually overlap each other, both layouts must be of the same layout type,

correspond to the same filehandle, and have the same iomode. Layouts conflict when they overlap and differ in the content of the layout (i.e., the storage device/file mapping parameters differ). Note that differing iomodes do not lead to conflicting layouts. It is permissible for layouts with different iomodes, pertaining to the same byte range, to be held by the same client. An example of this would be copy-on-write functionality for a block/volume layout type.

[12.2.9.](#) Layout Iomode

The layout iomode (data type layoutiomode4, see [Section 3.3.20](#))

indicates to the metadata server the client's intent to perform either just read operations or a mixture of I/O possibly containing read and write operations. For certain layout types, it is useful for a client to specify this intent at the time it sends LAYOUTGET ([Section 18.43](#)). For example, 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 CB_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 regard to the iomode; this is dependent upon storage device implementation and layout type. Thus, if the client's layout iomode is inconsistent with 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 iomode should be obtained via LAYOUTGET. For example, 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 use of the layout iomode does not conflict with OPEN share modes or byte-range lock requests; open mode and lock conflicts are enforced as they are without the use of pNFS, and are logically separate from the pNFS layout level. Open modes and locks are the preferred method for restricting user access to data files. For example, 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 byte-range locking to serialize their accesses.

[12.2.10](#). Device IDs

The device ID (data type deviceid4, see [Section 3.3.14](#)) identifies a group of storage devices. The scope of a device ID is the pair <client ID, layout type>. In practice, a significant amount of information may be required to fully address a storage device. Rather than embedding all such information in a layout, layouts embed

device IDs. The NFSv4.1 operation GETDEVICEINFO ([Section 18.40](#)) is used to retrieve the complete address information (including all device addresses for the device ID) regarding the storage device according to its layout type and device ID. 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.

Clients cannot expect the mapping between a device ID and its storage device address(es) to persist across metadata server restart. See [Section 12.7.4](#) for a description of how recovery works in that situation.

A device ID lives as long as there is a layout referring to the device ID. If there are no layouts referring to the device ID, the server is free to delete the device ID any time. Once a device ID is deleted by the server, the server MUST NOT reuse the device ID for the same layout type and client ID again. This requirement is feasible because the device ID is 16 bytes long, leaving sufficient room to store a generation number if server's implementation requires most of the rest of the device ID's content to be reused. This requirement is necessary because otherwise the race conditions between asynchronous notification of device ID addition and deletion would be too difficult to sort out.

Device ID to device address mappings are not leased, and can be changed at any time. (Note that while device ID to device address mappings are likely to change after the metadata server restarts, the server is not required to change the mappings.) A server has two choices for changing mappings. It can recall all layouts referring to the device ID or it can use a notification mechanism.

The NFSv4.1 protocol has no optimal way to recall all layouts that referred to a particular device ID (unless the server associates a single device ID with a single fsid or a single client ID; in which case, CB_LAYOUTRECALL has options for recalling all layouts associated with the fsid, client ID pair or just the client ID).

Via a notification mechanism (see [Section 20.12](#)), device ID to device address mappings can change over the duration of server operation without recalling or revoking the layouts that refer to device ID. The notification mechanism can also delete a device ID, but only if the client has no layouts referring to the device ID. A notification of a change to a device ID to device address mapping will immediately or eventually invalidate some or all of the device ID's mappings. The server MUST support notifications and the client must request them before they can be used. For further information about the notification types [Section 20.12](#).

[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 sent to a metadata server and summarized here. While pNFS is an OPTIONAL feature, if pNFS is implemented, some operations are REQUIRED in order to comply with pNFS. See [Section 17](#).

These are the fore channel pNFS operations:

GETDEVICEINFO. As noted previously ([Section 12.2.10](#)), GETDEVICEINFO ([Section 18.40](#)) returns the mapping of device ID to storage device address.

GETDEVICELIST ([Section 18.41](#)), allows clients to fetch all device IDs for a specific file system.

LAYOUTGET ([Section 18.43](#)) is used by a client to get a layout for a file.

LAYOUTCOMMIT ([Section 18.42](#)) is used to inform the metadata server of the client's intent to commit data which has been written to the storage device; the storage device as originally indicated in the return value of LAYOUTGET.

LAYOUTRETURN ([Section 18.44](#)) is used to return layouts for a file, an FSID and for client ID.

These are the backchannel pNFS operations:

CB_LAYOUTRECALL ([Section 20.3](#)) recalls a layout or all layouts belonging to a file system, or all layouts belonging to a client ID.

CB_RECALL_ANY ([Section 20.6](#)), tells a client that it needs to return some number of recallable objects, including layouts, to the metadata server.

CB_RECALLABLE_OBJ_AVAIL ([Section 20.7](#)) 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.

CB_NOTIFY_DEVICEID ([Section 20.12](#)) Notifies the client of changes to device IDs.

Internet-Draft

NFSv4.1

September 2008

[12.4.](#) pNFS Attributes

A number of attributes specific to pNFS are listed and described in [Section 5.12](#)

[12.5.](#) Layout Semantics

[12.5.1.](#) Guarantees Provided by Layouts

Layouts grant to the client the ability to access data located at a storage device with the appropriate storage protocol. The client is guaranteed the layout will be recalled when one of two things occur; either a conflicting layout is requested or the state encapsulated by the layout becomes invalid and this can happen when an event directly or indirectly modifies the layout. When a layout is recalled and returned by the client, the client continues with the ability to access file data with normal NFSv4.1 operations through the metadata server. Only the ability to access the storage devices is affected.

The requirement of NFSv4.1, that all user access rights MUST be obtained through the appropriate open, lock, and access operations, is not modified with the existence of layouts. Layouts are provided to NFSv4.1 clients and user access still follows the rules of the protocol as if they did not exist. It is a requirement that for a client to access a storage device, a layout must be held by the client. If a storage device receives an I/O for a byte range for which the client does not hold a layout, the storage device SHOULD reject that I/O request. Note that 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. Therefore, it is the requirement of the storage protocol or layout type that determines the necessary behavior. For example, block/volume layout types require that the layout's iomode 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. For an example of storage device access permissions see an object based protocol such as [\[39\]](#). If access permissions are encoded within the layout, the metadata

server SHOULD 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 with open, lock and access as described above. The degree to which it is possible for the client to circumvent these access operations and the consequences of doing so must be clearly specified by the individual layout type specifications. In addition, these specifications must be clear

about the requirements and non-requirements for the checking performed by the server.

In the presence of pNFS functionality, mandatory file locks MUST behave as they would without pNFS. Therefore, if mandatory file locks and layouts are provided simultaneously, the storage device MUST be able to enforce the mandatory file locks. For example, if one client obtains a mandatory lock and a second client accesses the storage device, the storage device MUST appropriately restrict I/O for the byte range of the mandatory file lock. If the storage device is incapable of providing this check in the presence of mandatory file locks, the metadata server then MUST NOT grant layouts and mandatory file locks simultaneously.

[12.5.2](#). Getting a Layout

A client obtains a layout with the LAYOUTGET operation. The metadata server will grant layouts of a particular type (e.g., block/volume, object, or file). The client selects an appropriate layout type that the server supports and the client is prepared to use. The layout returned to the client might not exactly match the requested byte range as described in [Section 18.43.3](#). As needed a client may make multiple LAYOUTGET requests; these might result in multiple overlapping, non-conflicting layouts (see [Section 12.2.8](#)).

In order to get a layout, the client must first have opened the file via the OPEN operation. When a client has no layout on a file, it MUST present a stateid as returned by OPEN, a delegation stateid, or a byte-range lock stateid in the loga_stateid argument. A successful LAYOUTGET result includes a layout stateid. The first successful LAYOUTGET processed by the server using a non-layout stateid as an argument MUST have the "seqid" field of the layout stateid in the response set to one. Thereafter, the client MUST use a layout

stateid (see [Section 12.5.3](#)) on future invocations of LAYOUTGET on the file, and the "seqid" MUST NOT be set to zero. Once the layout has been retrieved, it can be held across multiple OPEN and CLOSE sequences. Therefore, a client may hold a layout for a file that is not currently open by any user on the client. This allows for the caching of layouts beyond CLOSE.

The storage protocol used by the client to access the data on the storage device is determined by the layout's type. The client is responsible for matching the layout type with an available method to interpret and use the layout. The method for this layout type selection is outside the scope of the pNFS functionality.

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 5.12.4](#)) 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 might make it difficult, or impossible, for the server implementation to comply.

Because the EXCLUSIVE4 createmode4 does not allow the setting of attributes at file creation time, NFSv4.1 introduces the EXCLUSIVE4_1 createmode4, which does allow attributes to be set at file creation time. In addition, if the session is created with persistent reply caches, EXCLUSIVE4_1 is neither necessary nor allowed. Instead, GUARDED4 both works better and is prescribed. Table 10 in [Section 18.16.3](#), summarizes how a client is allowed to send an exclusive create.

[12.5.3](#). Layout Stateid

As with all other stateids, the layout stateid consists of a "seqid" and "other" field. Once a layout stateid is changed, the "other" field will stay constant unless the stateid is revoked, or the client returns all layouts on the file and the server disposes of the stateid. The "seqid" field is initially set to one, and is never zero on any NFSv4.1 operation that uses layout stateids, whether it is a fore channel or backchannel operation. After the layout stateid is established, the server increments by one the value of the "seqid"

in each subsequent LAYOUTGET and LAYOUTRETURN response, and in each CB_LAYOUTRECALL request.

Given the design goal of pNFS to provide parallelism, the layout stateid differs from other stateid types in that the client is expected to send LAYOUTGET and LAYOUTRETURN operations in parallel. The "seqid" value is used by the client to properly sort responses to LAYOUTGET and LAYOUTRETURN. The "seqid" is also used to prevent race conditions between LAYOUTGET and CB_LAYOUTRECALL. Given the processing rules differ from layout stateids and other stateid types, only the pNFS sections of this document should be considered to determine proper layout stateid handling.

Once the client receives a layout stateid, it MUST use the correct "seqid" for subsequent LAYOUTGET or LAYOUTRETURN operations. The correct "seqid" is defined as the highest "seqid" value from responses of fully processed LAYOUTGET or LAYOUTRETURN operations or arguments of a fully processed CB_LAYOUTRECALL operation. Since the server is incrementing the "seqid" value on each layout operation, the client may determine the order of operation processing by inspecting the "seqid" value. In the case of overlapping layout ranges, the ordering information will provide the client the

knowledge of which layout ranges are held. Note that overlapping layout ranges may occur because of the client's specific requests or because the server is allowed to expand the range of a requested layout and notify the client in the LAYOUTRETURN results. Additional layout stateid sequencing requirements are provided in [Section 12.5.5.2](#).

The client's receipt of a "seqid" is not sufficient for subsequent use. The client must fully process the operations before the "seqid" can be used. For LAYOUTGET results, if the client is not using the forgetful model ([Section 12.5.5.1](#)), it MUST first update its record of what ranges of the file's layout it has before using the seqid. For LAYOUTRETURN results, the client MUST delete the range from its record of what ranges of the file's layout it had before using the seqid. For CB_LAYOUTRECALL arguments, the client MUST send a response to the recall before using the seqid. The fundamental requirement in client processing is that the "seqid" is used to provide the order of processing. LAYOUTGET results may be processed in parallel. LAYOUTRETURN results may be processed in parallel.

LAYOUTGET and LAYOUTRETURN responses may be processed in parallel as long as the ranges do not overlap. CB_LAYOUTRECALL request processing MUST be processed in "seqid" order at all times.

Once a client has no more layouts on a file, the layout stateid is no longer valid, and MUST NOT be used. Any attempt to use such a layout stateid will result in NFS4ERR_BAD_STATEID.

[12.5.4](#). Committing a Layout

Allowing for varying storage protocols capabilities, the pNFS protocol does not require the metadata server and storage devices to have a consistent view of file attributes and data location mappings. Data location mapping refers to aspects such as which offsets store data as opposed to storing holes (see [Section 13.4.4](#) for a discussion). Related issues arise for storage protocols where a layout may hold provisionally allocated blocks where the allocation of those blocks does not survive a complete restart of both the client and server. Because of this inconsistency, it is necessary to re-synchronize the client with the metadata server and its storage devices and make any potential changes available to other clients. This is accomplished by use of the LAYOUTCOMMIT operation.

The LAYOUTCOMMIT operation is responsible for committing a modified layout to the metadata server. The data should be written and committed to the appropriate storage devices before the LAYOUTCOMMIT occurs. The scope of the LAYOUTCOMMIT operation depends on the storage protocol in use. It is important to note that the level of synchronization is from the point of view of the client which sent

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 is not REQUIRED to maintain a global view that accounts for other clients' I/O that may have occurred within the same time frame.

For block/volume-based layouts, LAYOUTCOMMIT may require updating the block list that comprises the file and committing this layout to stable storage. For file-layouts synchronization of attributes between the metadata and storage devices primarily the size attribute is required.

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.

The client MUST have a layout in order to issue LAYOUTCOMMIT.

[12.5.4.1](#). LAYOUTCOMMIT and change/time_modify

The change and time_modify attributes may be updated by the server when the LAYOUTCOMMIT operation is processed. The reason for this is that some layout types do not support the update of these attributes when the storage devices process I/O operations. If client has a layout with the LAYOUTIOMODE4_RW iomode on the file, the client MAY provide a suggested value to the server for time_modify within the arguments to LAYOUTCOMMIT. Based on the layout type, the provided value 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. The client always has the option to set time_modify through an explicit SETATTR operation.

For some layout protocols, the storage device is able to notify the metadata server of the occurrence of an I/O and as a result the change and time_modify attributes may be updated at the metadata server. For a metadata server that is capable of monitoring updates to the change and time_modify attributes, LAYOUTCOMMIT processing is not required to update the change attribute; in this case the metadata server must ensure that no further update to the data has occurred since the last update of the attributes; file-based protocols may have enough information to make this determination or may update the change attribute upon each file modification. This

also applies for the time_modify attribute. If the server implementation is able to determine that the file has not been modified since the last time_modify update, the server need not update time_modify at LAYOUTCOMMIT. At LAYOUTCOMMIT completion, the updated attributes should be visible if that file was modified since

the latest previous LAYOUTCOMMIT or LAYOUTGET.

[12.5.4.2.](#) LAYOUTCOMMIT and size

The size of a file may be updated when the LAYOUTCOMMIT operation is used by the client. One of the fields in the argument to LAYOUTCOMMIT is `loca_last_write_offset`; this field indicates the highest byte offset written but not yet committed with the LAYOUTCOMMIT operation. The data type of `loca_last_write_offset` is `newoffset4` and is switched on a boolean value, `no_newoffset`, that indicates if a previous write occurred or not. If `no_newoffset` is `FALSE`, an offset is not given. If the client has a layout with `LAYOUTIOMODE4_RW` `iomode` on the file, with an `lo_offset` and `lo_length` that overlaps `loca_last_write_offset`, then the client MAY set `no_newoffset` to `TRUE` and provide an offset that will update the file size. Keep in mind that offset is not the same as length, though they are related. For example, a `loca_last_write_offset` value of zero means that one byte was written at offset zero, and so the length of the file is at least one byte.

The metadata server may do one of the following:

1. Update the file's size using the last write offset provided by the client as either the true file size or as a hint of the file size. If the metadata server has a method available, any new value for file size should be sanity checked. For example, the file must not be truncated if the client presents a last write offset less than the file's current size.
2. Ignore the client provided last write offset; the metadata server must have sufficient knowledge from other sources to determine the file's size. For example, the metadata server queries the storage devices with the control protocol.

The method chosen to update the file's size will depend on the storage device's and/or the control protocol's capabilities. 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 last write offset appropriately.

The results of LAYOUTCOMMIT contain a new size value in the form of a `newsize4` union data type. If the file's size is set as a result of LAYOUTCOMMIT, the metadata server must reply with the new size;

otherwise the new size is not provided. If the file size is updated, the metadata server SHOULD update the storage devices such that the new file size is reflected when LAYOUTCOMMIT processing is complete. For example, the client should be able to READ up to the new file size.

The client can extend the length of a file or truncate a file by sending a SETATTR operation to the metadata server with the size attribute specified. If the size specified is larger than the current size of the file, the file is "zero extended", i.e., zeroes are implicitly added between the file's previous EOF and the new EOF. (In many implementations the zero extended region of the file consists of unallocated holes in the file.) When the client writes past EOF via WRITE, the SETATTR operation does not need to be used.

[12.5.4.3](#). LAYOUTCOMMIT and layoutupdate

The LAYOUTCOMMIT argument contains a `loca_layoutupdate` field ([Section 18.42.1](#)) of data type `layoutupdate4` ([Section 3.3.18](#)). 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 or did not use. The content of `loca_layoutupdate` (field `lou_body`) need not be the same layout type-specific content returned by LAYOUTGET ([Section 18.43.2](#)) 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.

[12.5.5](#). 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 byte range it represents. Any operation or action, such as 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 20.3](#)) and returned with LAYOUTRETURN [Section 18.44](#). The CB_LAYOUTRECALL operation may recall a layout identified by a byte range, all the layouts associated with a file system (FSID), or all layouts associated with a client ID. [Section 12.5.5.2](#) discusses sequencing issues surrounding the getting, returning, and recalling of layouts.

An `iomode` is also specified when recalling a layout. Generally, the `iomode` in the recall request must match the layout being returned;

Internet-Draft

NFSv4.1

September 2008

for example, a recall with an iomode of LAYOUTIOMODE4_RW should cause the client to only return LAYOUTIOMODE4_RW layouts and not LAYOUTIOMODE4_READ layouts. However, a special LAYOUTIOMODE4_ANY enumeration is defined to enable recalling a layout of any iomode; in other words, the client must return both read-only and read/write layouts.

A REMOVE operation SHOULD 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. After the last reference on the file has been released and the file has been removed, the client should no longer be able to perform I/O using the layout. In the case of a files based layout, the data server SHOULD return NFS4ERR_STALE in response to any operation on the removed file.

Once a layout has been returned, the client MUST NOT send I/Os to the storage devices for the file, byte range, and iomode represented by the returned layout. If a client does send an I/O to a storage device for which it does not hold a layout, the storage device SHOULD reject the I/O.

Although pNFS does not alter the file data 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 negatively affect the latency in returning a layout in response to a CB_LAYOUTRECALL; this is similar to file delegations and the impact that file data caching has on DELEGRETURN. Client implementations SHOULD limit the amount of unwritten data they have outstanding at any one time in order to prevent excessively long responses to CB_LAYOUTRECALL. Once a layout is recalled, a server MUST wait one lease period before taking further action. As soon as a lease period has past, the server may choose to fence the client's access to the storage devices if the server perceives the client has taken too long to return a layout. However, just as in the case of data delegation and DELEGRETURN, the server may choose to wait given that the client is showing forward progress on its way to returning the layout. This forward progress can take the form of successful interaction with the storage devices or sub-portions of the layout being returned by the client. The server can also limit exposure to these problems by

limiting the byte ranges initially provided in the layouts and thus the amount of outstanding modified data.

[12.5.5.1.](#) Layout 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 iomode. 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:

- o 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.
- o It may be 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.
- o It may be useful for clients to "forget" details about what layouts and ranges the client actually has, leading to the server's layout ranges being beyond those what the client "thinks" it has. As long as the client does not assume it has layouts that are beyond what the server has granted, this is a safe practice. When a client forgets what ranges and layouts it has, and it receives a CB_LAYOUTRECALL operation, the client MUST follow up with a LAYOUTRETURN for what the server recalled, or alternatively return the NFS4ERR_NOMATCHING_LAYOUT error if it has no layout to return in the recalled range.

- o 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 that the client does not know about, it is 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 CB_LAYOUTRECALL.

Thus, in light of the above, it is useful for a server to be able to send 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 layouts 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 portions. 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.

When a layout is returned, the client MUST NOT have any outstanding I/O requests to the storage devices involved in the layout. Rephrasing, the client MUST NOT return the layout while it has outstanding I/O requests to the storage device.

Even with this requirement for the client, it is possible that I/O requests may be presented to a storage device no longer allowed to perform them. Since the server has no strict control as to when the client will return the layout, the server may later decide to unilaterally revoke the client's access to the storage devices as provided by the layout. In choosing to revoke access, the server must deal with the possibility of lingering I/O request; those outstanding I/O requests are still in flight to storage devices identified by the revoked layout. All layout specifications MUST define whether unilateral layout revocation by the metadata server is supported; if it is, the specification must also describe how lingering writes are processed. For example, storage devices identified by the revoked layout could be fenced off from the client that held the layout.

In order to ensure client/server convergence with regard to 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 byte range; even if layouts pertaining to partial ranges were previously returned. In addition, if the client holds no layouts that overlaps the range being recalled, the client should return the NFS4ERR_NOMATCHING_LAYOUT error code to CB_LAYOUTRECALL. This allows the server to update its view of the client's layout state.

[12.5.5.2](#). Sequencing of Layout Operations

As with other stateful operations, pNFS requires the correct sequencing of layout operations. PNFS uses the "seqid" in the layout stateid to provide the correct sequencing between regular operations and callbacks. It is the server's responsibility to avoid inconsistencies regarding the layouts provided and the client's responsibility to properly serialize its layout requests and layout returns.

[12.5.5.2.1](#). Layout Recall and Return Sequencing

One critical issue with regard to layout operations sequencing concerns callbacks. The protocol must defend against races between the reply to a LAYOUTGET or LAYOUTRETURN operation and a subsequent CB_LAYOUTRECALL. A client MUST NOT process a CB_LAYOUTRECALL that implies one or more outstanding LAYOUTGET or LAYOUTRETURN operations to which the client has not yet received a reply. The client detects such a CB_LAYOUTRECALL by examining the "seqid" field of the recall's layout stateid. If the "seqid" is not one higher than what the client currently has recorded, and the client has at least one LAYOUTGET and/or LAYOUTRETURN operation outstanding, the client knows the server sent the CB_LAYOUTRECALL after sending a response to an outstanding LAYOUTGET or LAYOUTRETURN. The client MUST wait before processing such a CB_LAYOUTRECALL until it processes all replies for outstanding LAYOUTGET and LAYOUTRETURN operations for the corresponding file with seqid less than the seqid given by CB_LAYOUTRECALL (lor_stateid, see [Section 20.3](#).)

In addition to the seqid-based mechanism, [Section 2.10.6.3](#) describes the sessions mechanism for allowing the client to detect callback

race conditions and delay processing such a CB_LAYOUTRECALL. The server MAY reference conflicting operations in the CB_SEQUENCE that precedes the CB_LAYOUTRECALL. Because the server has already sent replies for these operations before issuing the callback, the replies may race with the CB_LAYOUTRECALL. The client MUST wait for all the referenced calls to complete and update its view of the layout state before processing the CB_LAYOUTRECALL.

[12.5.5.2.1.1](#). Get/Return Sequencing

The protocol allows the client to send concurrent LAYOUTGET and LAYOUTRETURN operations to the server. The protocol does not provide any means for the server to process the requests in the same order in which they were created. However, through the use of the "seqid" field in the layout stateid, the client can determine the order in which parallel outstanding operations were processed by the server. Thus, when a layout retrieved by an outstanding LAYOUTGET operation intersects with a layout returned by an outstanding LAYOUTRETURN on the same file, the order in which the two conflicting operations are processed determines the final state of the overlapping layout. The order is determined by the "seqid" returned in each operation: the operation with the higher seqid was executed later.

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, and a mix of both.

It is permissible for the client to use the current stateid (see [Section 16.2.3.1.2](#)) for LAYOUTGET operations for example when compounding LAYOUTGETs or compounding OPEN and LAYOUTGETs. It is also permissible to use the current stateid when compounding LAYOUTRETURNS.

It is permissible for the client to use the current stateid when combining LAYOUTRETURN and LAYOUTGET operations for the same file in the same COMPOUND request since the server MUST process these in order. However, if a client does send such COMPOUND 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.

12.5.5.2.1.2. Client Considerations

Consider a pNFS client that has sent a LAYOUTGET and before it receives the reply to LAYOUTGET, it receives a CB_LAYOUTRECALL for the same file with an overlapping range. There are two possibilities, which the client can distinguish via the layout stateid in the recall.

1. The server processed the LAYOUTGET before issuing the recall, so the LAYOUTGET must be waited for because it may be carrying layout information that will need to be returned to deal with the CB_LAYOUTRECALL.
2. The server sent the callback before receiving the LAYOUTGET. The server will not respond to the LAYOUTGET until the CB_LAYOUTRECALL is processed.

If these possibilities cannot be distinguished, a deadlock could result, 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. Note that in the first case, the "seqid" in the layout stateid of the recall is two greater than what the client has recorded and in the second case, the "seqid" is one greater than what the client has recorded. This allows the client to disambiguate between the two cases. The client thus knows precisely which possibility applies.

In case 1 the client knows it needs to wait for the LAYOUTGET response before processing the recall (or the client can return NFS4ERR_DELAY).

In case 2 the client will not wait for the LAYOUTGET response before processing the recall, because waiting would cause deadlock. 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 recall process can be considered completed when the final LAYOUTRETURN operation for the recalled range is completed. The LAYOUTRETURN uses the layout stateid (with seqid) specified in CB_LAYOUTRECALL. If the client uses multiple LAYOUTRETURNS in

processing the recall, the first LAYOUTRETURN will use the layout stateid as specified in CB_LAYOUTRECALL. Subsequent LAYOUTRETURNS will use the highest seqid as is the usual case.

12.5.5.2.1.3. Server Considerations

Consider a race from the metadata server's point of view. The metadata server has sent a CB_LAYOUTRECALL and receives an overlapping LAYOUTGET for the same file before the LAYOUTRETURN(s) that respond to the CB_LAYOUTRECALL. There are three cases:

1. The client sent the LAYOUTGET before processing the CB_LAYOUTRECALL. The "seqid" in the layout stateid of LAYOUTGET is two less than the "seqid" in CB_LAYOUTRECALL. The server returns NFS4ERR_RECALLCONFLICT to the client, which indicates to the client that there is a pending recall.
2. The client sent the LAYOUTGET after processing the CB_LAYOUTRECALL, but the LAYOUTGET arrived before the LAYOUTRETURN and the response to CB_LAYOUTRECALL that completed that processing. The "seqid" in the layout stateid of LAYOUTGET is equal to or greater than that of the "seqid" in CB_LAYOUTRECALL. The server has not received a response to the CB_LAYOUTRECALL, so it returns NFS4ERR_RECALLCONFLICT.
3. The client sent the LAYOUTGET after processing the CB_LAYOUTRECALL, the server received the CB_LAYOUTRECALL response, but the LAYOUTGET arrived before the LAYOUTRETURN that completed that processing. The "seqid" in the layout stateid of LAYOUTGET is equal to that of the "seqid" in CB_LAYOUTRECALL. The server has received a response to the CB_LAYOUTRECALL, so it returns NFS4ERR_RETURNCONFLICT.

12.5.5.2.1.4. Wraparound and Validation of Seqid

The rules for layout stateid processing differ from other stateids in the protocol because the "seqid" value cannot be zero and the stateid's "seqid" value changes in a CB_LAYOUTRECALL operation. The non-zero requirement combined with the inherent parallelism of layout operations means that a set of LAYOUTGET and LAYOUTRETURN operations may contain the same value for "seqid". The server uses a slightly

modified version of the modulo arithmetic as described in [Section 2.10.6.1](#) when incrementing the layout stateid's "seqid". The modification to that modulo arithmetic description is to not use zero. The modulo arithmetic is also used for the comparisons of "seqid" values in the processing of CB_LAYOUTRECALL events as described above in [Section 12.5.5.2.1.3](#).

Just as the server validates the "seqid" in the event of CB_LAYOUTRECALL usage, as described in [Section 12.5.5.2.1.3](#), the server also validates the "seqid" value to ensure that it is within an appropriate range. This range represents the degree of parallelism the server supports for layout stateids. If the client is sending multiple layout operations to the server in parallel, by definition, the "seqid" value in the supplied stateid will not be the current "seqid" as held by the server. The range of parallelism spans from the highest or current "seqid" to a "seqid" value in the past. To assist in the discussion, the server's current "seqid" value for a layout stateid is defined as: SERVER_CURRENT_SEQID. The lowest "seqid" value that is acceptable to the server is represented by PAST_SEQID. And the value for the range of valid "seqid"s or range of parallelism is VALID_SEQID_RANGE. Therefore, the following holds: $\text{VALID_SEQID_RANGE} = \text{SERVER_CURRENT_SEQID} - \text{PAST_SEQID}$. In the following, all arithmetic is the modulo arithmetic as described above.

The server MUST support a minimum VALID_SEQID_RANGE. The minimum is defined as: $\text{VALID_SEQID_RANGE} = \text{summation of } 1..N \text{ of } (\text{ca_maxoperations}(i) - 1)$ where N is the number of session fore channels and ca_maxoperations(i) is the value of the ca_maxoperations returned from CREATE_SESSION of the i'th session. The reason for minus 1 is to allow for the required SEQUENCE operation. The server MAY support a VALID_SEQID_RANGE value larger than the minimum. The maximum VALID_SEQID_RANGE is $(2^{32} - 2)$ (accounts for 0 not being a valid "seqid" value).

If the server finds the "seqid" is zero, the NFS4ERR_BAD_STATEID error is returned to the client. The server further validates the "seqid" to ensure it is within the range of parallelism, VALID_SEQID_RANGE. If the "seqid" value is outside of that range, the error NFS4ERR_OLD_STATEID is returned to the client. Upon receipt of NFS4ERR_OLD_STATEID, the client updates the stateid in the layout request based on processing of other layout requests and re-sends the operation to the server.

[12.5.5.2.1.5](#). Bulk Recall and Return

PNFS supports recalling and returning all layouts that are for files belonging to a particular fsid (LAYOUTRECALL4_FSID,

Internet-Draft

NFSv4.1

September 2008

LAYOUTRETURN4_FSID) or client ID (LAYOUTRECALL4_ALL, LAYOUTRETURN4_ALL). There are no "bulk" stateids, so detection of races via the seqid is not possible. The server MUST NOT initiate bulk recall while another recall is in progress, or the corresponding LAYOUTRETURN is in progress or pending. In the event the server sends a bulk recall while the client has pending or in progress LAYOUTRETURN, CB_LAYOUTRECALL, or LAYOUTGET, the client returns NFS4ERR_DELAY. In the event the client sends a LAYOUTGET or LAYOUTRETURN while a bulk recall is in progress, the server returns NFS4ERR_RECALLCONFLICT. If the client sends a LAYOUTGET or LAYOUTRETURN after the server receives NFS4ERR_DELAY from a bulk recall, then to ensure forward progress, the server MAY return NFS4ERR_RECALLCONFLICT.

Once a CB_LAYOUTRECALL of LAYOUTRECALL4_ALL is sent, the server MUST NOT allow the client to use any layout stateid except for LAYOUTCOMMIT operations. Once the client receives a CB_LAYOUTRECALL of LAYOUTRECALL4_ALL, it MUST NOT use any layout stateid except for LAYOUTCOMMIT operations. Once a LAYOUTRETURN of LAYOUTRETURN4_ALL is sent, all layout stateids granted to the client ID are freed. The client MUST NOT use the layout stateids again. It MUST use LAYOUTGET to obtain new layout stateids.

Once a CB_LAYOUTRECALL of LAYOUTRECALL4_FSID is sent, the server MUST NOT allow the client to use any layout stateid that refers to a file with the specified fsid except for LAYOUTCOMMIT operations. Once the client receives a CB_LAYOUTRECALL of LAYOUTRECALL4_ALL, it MUST NOT use any layout stateid that refers to a file with the specified fsid except for LAYOUTCOMMIT operations. Once a LAYOUTRETURN of LAYOUTRETURN4_FSID is sent, all layout stateids granted to the referenced fsid are freed. The client MUST NOT use the layout stateids for files with the referenced fsid again. It MUST use LAYOUTGET to obtain new layout stateids files with the referenced fsid.

If the server has sent a bulk CB_LAYOUTRECALL, and receives a LAYOUTGET, or a LAYOUTRETURN with a stateid, the server MUST return NFS4ERR_RECALLCONFLICT. If the server has sent a bulk CB_LAYOUTRECALL, and receives a LAYOUTRETURN with an lr_returntype that is not equal to the lor_recalltype of the CB_LAYOUTRECALL, the server MUST return NFS4ERR_RECALLCONFLICT.

[12.5.6](#). Revoking Layouts

Parallel NFS permits servers to revoke layouts from clients that fail to response to recalls and/or fail to renew their lease in time. Whether the server revokes the layout or not depends on the layout type, and what actions are taken with respect to the client's I/O to

data servers is also layout type specific.

[12.5.7.](#) 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 may give out either the old data, the new data, or a mixture of new and old. Upon completion of a synchronous WRITE or COMMIT (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. Note that the layouts MUST be recalled prior to the server responding to the associated WRITE operations.

[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 sends a GETATTR to the NFSv4.1 server for the `fs_layout_type` ([Section 5.12.1](#)) attribute. If the attribute returns at least one layout type, and the layout types returned are among the set supported by the client, the client knows that pNFS is a possibility for the file system. 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; in this case, the server MUST return NFS4ERR_NOTSUPP in response to any pNFS operation.

The client then creates a session, requesting a persistent session, so that exclusive creates can be done with single round trip via the createmode4 of GUARDED4. If the session ends up not being persistent, the client will use EXCLUSIVE4_1 for exclusive creates.

If a file is to be created on a pNFS enabled file system, the client uses the OPEN operation. With the normal set of attributes that may be provided upon OPEN used for creation, there is an OPTIONAL

layout_hint attribute. The client's use of layout_hint allows the client to express its preference for a layout type and its associated layout details. The use of a createmode4 of UNCHECKED4, GUARDED4, or EXCLUSIVE4_1 will allow the client to provide the layout_hint attribute at create time. The client MUST NOT use EXCLUSIVE4 (see Table 10). The client is RECOMMENDED to combine a GETATTR operation after the OPEN within the same COMPOUND. The GETATTR may then retrieve the layout_type attribute for the newly created file. The client will then know what layout type the server has chosen for the file and therefore 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 and it chooses to use pNFS for data access, it then sends LAYOUTGET using the filehandle and stateid returned by OPEN, specifying the range it wants to do I/O on. The response is a layout, which may be a subset of the range for which the client asked. 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 send 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 sends a GETDEVICEINFO to find the device address(es) of the device ID. The client then sends the I/O request one of device ID's device addresses, using the storage protocol defined for the layout type. Note that if a client has multiple I/Os to send, these I/O requests may be done in parallel.

If the I/O was a WRITE, then at some point the client may want to use LAYOUTCOMMIT to commit the modification time and the new size of the file (if it believes it extended the file size) to the metadata server and the modified data to the file system.

[12.7.](#) Recovery

Recovery is complicated by the distributed nature of the pNFS protocol. In general, crash recovery for layouts is similar to crash recovery for delegations in the base NFSv4.1 protocol. However, the client's ability to perform I/O without contacting the metadata

server introduces subtleties that must be handled correctly if the possibility of file system corruption is to be avoided.

[12.7.1.](#) Recovery from Client Restart

Client recovery for layouts is similar to client recovery for other lock and delegation state. When an pNFS client restarts, 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 no other layout conflicts.

The second is through the client restarting 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. In this scenario, the data written by the client but not covered by a successful LAYOUTCOMMIT is in an undefined state; it may have been written or it may now be lost. This is acceptable behavior and it is the client's responsibility to use LAYOUTCOMMIT to achieve the desired level of stability.

12.7.2. Dealing with Lease Expiration on the Client

If a client believes its lease has expired, it MUST NOT send I/O to the storage device until it has validated its lease. The client can send 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 NOT use currently held layouts. The client has two choices to recover from the lease expiration. First, for all modified but uncommitted data, write it to the metadata server using the FILE_SYNC4 flag for the WRITES or WRITE and COMMIT. Second, the client reestablishes a client ID and session with the server and obtain new layouts and device ID to device address mappings for the modified data ranges and then write the data to the storage devices with the newly obtained

layouts.

If sr_status_flags from the metadata server has SEQ4_STATUS_RESTART_RECLAIM_NEEDED set (or SEQUENCE returns NFS4ERR_BAD_SESSION and CREATE_SESSION returns NFS4ERR_STALE_CLIENTID) then the metadata server has restarted, and the client SHOULD recover 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 11.7.7.1](#). After that, the client may get an indication that the layout state was not moved with the file system. The client recovers as in the other applicable situations discussed in Paragraph 1 or Paragraph 2 of this section.

If `sr_status_flags` reports no loss of state, then the lease for the layouts the client has are valid and renewed, and the client can once again send I/O requests to the storage devices.

While clients SHOULD NOT send I/Os to storage devices that may extend past the lease expiration time period, this is not always possible; for example, an extended network partition that starts after the I/O is sent and does not heal until the I/O request is received by the storage device. Thus the metadata server and/or storage devices are responsible for protecting themselves 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 is a description of the case where all of the following are true:

- o the metadata server has not restarted
- o a pNFS client's layouts have been discarded (usually because the client's lease expired) and are invalid
- o an I/O from the pNFS client arrives at the storage device

The metadata server and its storage devices MUST solve this by fencing the client. In other words, 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 ([Section 13.11](#)), 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 via the methods described in [Section 8.4.2](#) and discussed in a pNFS-specific context in Paragraph 2, of [Section 12.7.2](#). The client MUST stop using layouts and delete the 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 layouts via LAYOUTCOMMIT, then the client has additional work to do in order to have the client, metadata server and storage device(s) all synchronized on the state of the data.

- o 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 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 18.32](#)) operation, and then obtain layouts as desired.
- o If the client asynchronously wrote data to the storage device, but still has a copy of the 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.
- o 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 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 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 send 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 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 file system.

To send LAYOUTCOMMIT in reclaim mode, the client sets the `loca_reclaim` field of the operation's arguments ([Section 18.42.1](#)) 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 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 storage device to the file system. 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 12.5.4.3](#). 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. This condition does not hold true for all layout types; for example, files-based storage devices neednot suffer from this limitation.

- o 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. For LAYOUTGET, the metadata server must reliably determine that servicing such a request will not conflict with an impending LAYOUTCOMMIT reclaim request. For WRITE, it must reliably determine that it will not conflict with an impending OPEN; or a LOCK where the file has mandatory file locking enabled.

As mentioned previously, some operations, namely WRITE and LAYOUTGET may 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 byte-range 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 which has the information necessary to recover non-committed data; since, it holds the modified data and probably nothing 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.

Two sub-cases can be distinguished.

1. The storage device uses NFSv4.1 as the storage protocol, i.e. same physical hardware is used to implement both a metadata and data server. See [Section 13.1](#) for a description how multiple roles are handled.
2. 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 for pNFS

pNFS separates file system metadata and data and provides access to both. There are pNFS-specific operations (listed in [Section 12.3](#)) that provide access to the metadata; all existing NFSv4.1 conventional (non-pNFS) security mechanisms and features apply to accessing the metadata. The combination of components in a pNFS system (see Figure 1) 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 forgo 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 as strong as or no weaker than those available via RPCSEC_GSS for NFSv4.1. Except for the storage protocol used for the

LAYOUT4_NFSV4_1_FILES layout (see [Section 13](#)), i.e. except for NFSv4.1, it is beyond the scope of this document to specify the security mechanisms for storage access protocols.

pNFS implementations MUST NOT remove NFSv4.1'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 layout types 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.](#) Client ID and Session Considerations

Sessions are a REQUIRED feature of NFSv4.1, and this extends to both the metadata server and file-based (NFSv4.1-based) data servers.

The role a server plays in pNFS is determined by the result it returns from EXCHANGE_ID. The roles are:

- o metadata server (EXCHGID4_FLAG_USE_PNFS_MDS is set in the result `eir_flags`),
- o data server (EXCHGID4_FLAG_USE_PNFS_DS)
- o non-metadata server (EXCHGID4_FLAG_USE_NON_PNFS). This is an NFSv4.1 server that does not support operations (e.g. `LAYOUTGET`) or attributes that pertain to pNFS.

The client MAY request zero or more of EXCHGID4_FLAG_USE_NON_PNFS, EXCHGID4_FLAG_USE_PNFS_DS, or EXCHGID4_FLAG_USE_PNFS_MDS, even though some combinations (e.g. EXCHGID4_FLAG_USE_NON_PNFS | EXCHGID4_FLAG_USE_PNFS_MDS) are contradictory. The server however

MUST only return the following acceptable combinations:

Acceptable Results from EXCHANGE_ID	
EXCHGID4_FLAG_USE_PNFS_MDS	
EXCHGID4_FLAG_USE_PNFS_MDS	EXCHGID4_FLAG_USE_PNFS_DS
EXCHGID4_FLAG_USE_PNFS_DS	
EXCHGID4_FLAG_USE_NON_PNFS	
EXCHGID4_FLAG_USE_PNFS_DS	EXCHGID4_FLAG_USE_NON_PNFS

As the above table implies, a server can have one or two roles. A server can be both a metadata server and a data server or it can be both a data server and non-metadata server. In addition to returning two roles in EXCHANGE_ID's results, and thus serving both roles via a common client ID, a server can serve two roles by returning a unique client ID and server owner for each role in each of two EXCHANGE_ID results, with each result indicating each role.

In the case of a server with concurrent PNFS roles that are served by a common client ID, if the EXCHANGE_ID request from the client has zero or a combination of the bits set in eia_flags, the server result should set bits which represent the higher of the acceptable combination of the server roles, with a preference to match the roles

requested by the client. Thus if a client request has (EXCHGID4_FLAG_USE_NON_PNFS | EXCHGID4_FLAG_USE_PNFS_MDS | EXCHGID4_FLAG_USE_PNFS_DS) flags set, and the server is both a metadata server and a data server, serving both the roles by a common client ID, the server SHOULD return with (EXCHGID4_FLAG_USE_PNFS_MDS | EXCHGID4_FLAG_USE_PNFS_DS) set.

In the case of a server that has multiple concurrent PNFS roles, each role served by a unique client ID, if the client specifies zero or a combination of roles in the request, the server results SHOULD return only one of the roles from the combination specified by the client request. If the role specified by the server result does not match the intended use by the client, the client should send the EXCHANGE_ID specifying just the interested PNFS role.

If a 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 the 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 the data server has no relationship with the client ID returned by a metadata server unless the client IDs are equal and the server owners and server scopes of the data server and metadata server are equal.

In NFSv4.1, the session ID 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 session ID in the preceding SEQUENCE operation is tied to the client ID of the data server, 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 metadata server routing and/or identity information is encoded in data server filehandles, when the metadata server identity or location changes, the data server filehandles it gave out must become invalid (stale), and so the metadata server must first recall the layouts. Invalidating a data server filehandle does not render the NFS client's data cache invalid. The client's cache should map a data server filehandle to a metadata server filehandle, and a metadata server filehandle to cached data.

If a server is both a metadata server and a data server, the server might need to distinguish operations on files that are directed to the metadata server from those that are directed to the data server. It is RECOMMENDED that the values of the filehandles returned by the LAYOUTGET operation to be different than the value of the filehandle returned by the OPEN of the same file.

Another scenario is for the metadata server and the storage device to be distinct from one client's point of view, and the roles reversed from 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 encode the values of filehandles according to their specific roles.

[13.1.1.1](#). Sessions Considerations for Data Servers

[Section 2.10.10.2](#) states that a client has to keep its lease renewed in order to prevent a session from being deleted by the server. If the reply to EXCHANGE_ID has just the EXCHGID4_FLAG_USE_PNFS_DS role set, then as noted in [Section 13.6](#) the client will not be able to

determine the data server's lease_time attribute, because GETATTR will not be permitted. Instead, the rule is that any time a client receives a layout referring it to a data server that returns just the EXCHGID4_FLAG_USE_PNFS_DS role, the client MAY assume that the lease_time attribute from the metadata server that returned the layout applies to the data server. Thus the data server MUST be aware of the values of all lease_time attributes of all metadata servers it is providing I/O for, and MUST use the maximum of all such lease_time values as the lease interval for all client IDs and sessions established on it.

For example, if one metadata server has a `lease_time` attribute of 20 seconds, and a second metadata server has a `lease_time` attribute of 10 seconds, then if both servers return layouts that refer to an `EXCHGID4_FLAG_USE_PNFS_DS`-only data server, the data server **MUST** renew a client's lease if the interval between two `SEQUENCE` operations on different `COMPOUND` requests is less than 20 seconds.

[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 fixed size quantity of data written to a data server.

Pattern. A pattern is a method of distributing one or more equal sized units across a set of data servers. A pattern is iterated one or more times.

Stripe. An stripe is a set of data distributed across a set of data servers in a pattern before that pattern repeats.

Stripe Count. A stripe count is the number of units in a pattern.

Stripe Width. A stripe width is the size of stripe in bytes. The stripe width = the stripe count * the size of the stripe unit.

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 (and depending on the implementation, will possibly assign a unique data filehandle to each data file).

[13.3.](#) File Layout Data Types

The high level NFSv4.1 layout types are `nfsv4_1_file_layouthint4`, `nfsv4_1_file_layout_ds_addr4`, and `nfsv4_1_file_layout4`.

The SETATTR operation supports a layout hint attribute ([Section 5.12.4](#)). 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.

```
const NFL4_UFLG_MASK           = 0x0000003F;
const NFL4_UFLG_DENSE         = 0x00000001;
const NFL4_UFLG_COMMIT_THRU_MDS = 0x00000002;
const NFL4_UFLG_STRIPE_UNIT_SIZE_MASK
                               = 0xFFFFF0;

typedef uint32_t nfl_util4;

enum filelayout_hint_care4 {
    NFLH4_CARE_DENSE           = NFL4_UFLG_DENSE,

    NFLH4_CARE_COMMIT_THRU_MDS
                               = NFL4_UFLG_COMMIT_THRU_MDS,

    NFLH4_CARE_STRIPE_UNIT_SIZE
                               = 0x00000040,

    NFLH4_CARE_STRIPE_COUNT = 0x00000080
};

/* Encoded in the loh_body field of type layouthint4: */

struct nfsv4_1_file_layouthint4 {
    uint32_t      nflh_care;
    nfl_util4     nflh_util;
    count4        nflh_stripe_count;
};
```

The generic layout hint structure is described in [Section 3.3.19](#). The client uses the layout hint in the layout_hint ([Section 5.12.4](#)) attribute to indicate the preferred 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 three fields. The first field, nflh_care, is a set of flags indicating which values of the hint the client cares about. If the NFLH4_CARE_DENSE flag is set, then the client indicates in the second field, nflh_util, a

preference for how the data file is packed ([Section 13.4.4](#)), which is controlled by the value of `nflh_util` & `NFL4_UFLG_DENSE`. If the `NFLH4_CARE_COMMIT_THRU_MDS` flag is set, then the client indicates a preference for whether the client should send COMMIT operations to the metadata server or data server ([Section 13.7](#)), which is controlled by the value of `nflh_util` & `NFL4_UFLG_COMMIT_THRU_MDS`. If the `NFLH4_CARE_STRIPE_UNIT_SIZE` flag is set, the client indicates its preferred stripe unit size, which is indicated in `nflh_util` & `NFL4_UFLG_STRIPE_UNIT_SIZE_MASK` (thus the stripe unit size MUST be a multiple of 64 bytes). The minimum stripe unit size is 64 bytes. If the `NFLH4_CARE_STRIPE_COUNT` flag is set, the client indicates in the third field, `nflh_stripe_count`, the stripe count. The stripe count multiplied by the stripe unit size is the stripe width.

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 a storage device ID (field `nfl_deviceid`) 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`.

```
typedef netaddr4 multipath_list4<>;

/* Encoded in the da_addr_body field of type device_addr4: */
struct nfsv4_1_file_layout_ds_addr4 {
    uint32_t          nfl_da_stripe_indices<>;
    multipath_list4 nfl_da_multipath_ds_list<>;
};
```

The `nfsv4_1_file_layout_ds_addr4` data type represents the device address. It is composed of two fields:

1. `nfl_da_multipath_ds_list`: An array of lists of data servers, where each list can be one or more elements, and each element represents a (see [Section 13.5](#)) data server address which may serve equally as the target of IO operations. The length of this array might be different than the stripe count.
2. `nfl_da_stripe_indices`: An array of indexes used to index into `nfl_da_multipath_ds_list`. Each element of `nfl_da_stripe_indices` MUST be less than the number of elements in `nfl_da_multipath_ds_list`. Each element of `nfl_da_multipath_ds_list`

SHOULD be referred to by one or more elements of

Internet-Draft

NFSv4.1

September 2008

nfl_da_stripe_indices. The number of elements in nfl_da_stripe_indices is always equal to the stripe count.

```
/* Encoded in the loc_body field of type layout_content4: */
struct nfsv4_1_file_layout4 {
    deviceid4      nfl_deviceid;
    nfl_util4      nfl_util;
    uint32_t       nfl_first_stripe_index;
    offset4        nfl_pattern_offset;
    nfs_fh4        nfl_fh_list<>;
};
```

The nfsv4_1_file_layout4 data type represents the layout. It is composed of the following fields:

1. nfl_deviceid: The device ID which maps to a value of type nfsv4_1_file_layout_ds_addr4.
2. nfl_util: Like the nflh_util field of data type nfsv4_1_file_layouthint4, a compact representation of how the data on a file on each data server is packed, whether the client should send COMMIT operations to the metadata server or data server, and the stripe unit size. If a server returns two or more overlapping layouts, each stripe unit size in each overlapping layout MUST be the same.
3. nfl_first_stripe_index: The index into the first element of the nfl_da_stripe_indices array to use.
4. nfl_pattern_offset: This field is the logical offset into the file where the striping pattern starts. It is required for converting the client's logical I/O offset (e.g. the current offset in a POSIX file descriptor before the read() or write() system call is sent) into the stripe unit number (see [Section 13.4.1](#)).

If dense packing is used, then nfl_pattern_offset is also needed to convert the client's logical I/O offset to an offset on the

file on the data server corresponding to the stripe unit number (see [Section 13.4.4](#)).

Note that `nfl_pattern_offset` is not always the same as `lo_offset`. For example, via the `LAYOUTGET` operation, a client might request a layout starting at offset 1000 of a file that has its striping pattern start at offset 0.

5. `nfl_fh_list`: An array of data server filehandles for each list of data servers in each element of the `nfl_da_multipath_ds_list` array. The number of elements in `nfl_fh_list` depends on whether sparse or dense packing is being used.

- * If sparse packing is being used, the number of elements in `nfl_fh_list` MUST be one of three values:
 - + Zero. This means that filehandles used for each data server are the same as the filehandle returned by the `OPEN` operation from the metadata server.
 - + One. This means that every data server uses the same filehandle: what is specified in `nfl_fh_list[0]`.
 - + The same number of elements in `nfl_da_multipath_ds_list`. Thus, in this case, when issuing an I/O to any data server in `nfl_da_multipath_ds_list[X]`, the filehandle in `nfl_fh_list[X]` MUST be used.

See the discussion on sparse packing in [Section 13.4.4](#).

- * If dense packing is being used, number of elements in `nfl_fh_list` MUST be the same as the number of elements in `nfl_da_stripe_indices`. Thus when issuing I/O to any data server in `nfl_da_multipath_ds_list[nfl_da_stripe_indices[Y]]`, the filehandle in `nfl_fh_list[Y]` MUST be used. In addition, any time there exists `i`, and `j`, (`i != j`) such that the intersection of `nfl_da_multipath_ds_list[nfl_da_stripe_indices[i]]` and `nfl_da_multipath_ds_list[nfl_da_stripe_indices[j]]` is not empty, then `nfl_fh_list[i]` MUST NOT equal `nfl_fh_list[j]`. In other

words, when dense packing is being used, if a data server appears in two or more units of a striping pattern, each reference to the data server MUST use a different filehandle.

Indeed, if there are multiple striping patterns, as indicated by the presence of multiple objects of data type layout4 (either returned in one or multiple LAYOUTGET operations), and a data server is the target of a unit of one pattern and another unit of another pattern, then each reference to each data server MUST use a different filehandle.

See the discussion on dense packing in [Section 13.4.4](#).

The details on the interpretation of the layout are in [Section 13.4](#).

[13.4](#). Interpreting the File Layout

[13.4.1](#). Determining the Stripe Unit Number

To find the stripe unit number that corresponds to the client's logical file offset, the pattern offset must also be used. The i 'th stripe unit (SU_i) is:

```
relative_offset = file_offset - nfl_pattern_offset;  
 $SU_i$  = floor(relative_offset / stripe_unit_size);
```

[13.4.2](#). Interpreting the File Layout Using Sparse Packing

When sparse packing is used, the algorithm for determining the filehandle and set of data server network addresses to write stripe unit i (SU_i) to is:

```
stripe_count = number of elements in nfl_da_stripe_indices;  
  
 $j$  = ( $SU_i$  + nfl_first_stripe_index) % stripe_count;  
  
 $idx$  = nfl_da_stripe_indices[ $j$ ];  
  
 $fh\_count$  = number of elements in nfl_fh_list;  
 $ds\_count$  = number of elements in nfl_da_multipath_ds_list;
```

```

switch (fh_count) {
  case ds_count:
    fh = nfl_fh_list[idx];
    break;

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

  case 0:
    fh = filehandle returned by OPEN;
    break;

  default:
    throw a fatal exception;
    break;
}

address_list = nflda_multipath_ds_list[idx];

```

The client would then select a data server from `address_list`, and send a READ or WRITE operation using the filehandle specified in `fh`.

Consider the following example:

Suppose we have a device address consisting of seven data servers, arranged in three equivalence ([Section 13.5](#)) classes:

```
{ A, B, C, D }, { E }, { F, G }
```

Where A through G are network addresses.

Then

```
nflda_multipath_ds_list<> = { A, B, C, D }, { E }, { F, G }
```

i.e.

```
nflda_multipath_ds_list[0] = { A, B, C, D }
```



```
nfl_da_multipath_ds_list[1] = { E }
```

```
nfl_da_multipath_ds_list[2] = { F, G }
```

Suppose the striping index array is:

```
nfl_da_stripe_indices<> = { 2, 0, 1, 0 }
```

Now suppose the client gets a layout which has a device ID that maps to the above device address. The initial index,

```
nfl_first_stripe_index = 2,
```

and

```
nfl_fh_list = { 0x36, 0x87, 0x67 }.
```

If the client wants to write to SU0, the set of valid { network address, filehandle } combinations for SU_i are determined by:

```
nfl_first_stripe_index = 2
```

So

```
idx = nfl_da_stripe_indices[(0 + 2) % 4]
```

```
    = nfl_da_stripe_indices[2]
```

```
    = 1
```

So

```
nfl_da_multipath_ds_list[1] = { E }
```

and

```
nfl_fh_list[1] = { 0x87 }
```

The client can thus write SU0 to { 0x87, { E }, }.

The destinations of the first thirteen storage units are:

SUI	filehandle	data servers
0	87	E
1	36	A,B,C,D
2	67	F,G
3	36	A,B,C,D
4	87	E
5	36	A,B,C,D
6	67	F,G
7	36	A,B,C,D
8	87	E
9	36	A,B,C,D
10	67	F,G
11	36	A,B,C,D
12	87	E

[13.4.3.](#) Interpreting the File Layout Using Dense Packing

When dense packing is used, the algorithm for determining the filehandle and set of data server network addresses to write stripe unit *i* (SUI) to is:

```
stripe_count = number of elements in nflda_stripe_indices;
```

```
j = (SUI + nfl_first_stripe_index) % stripe_count;
```

```
idx = nflda_stripe_indices[j];
```

```

fh_count = number of elements in nfl_fh_list;
ds_count = number of elements in nflda_multipath_ds_list;

switch (fh_count) {
    case stripe_count:
        fh = nfl_fh_list[j];
        break;

    default:
        throw a fatal exception;
        break;
}

address_list = nflda_multipath_ds_list[idx];

```

The client would then select a data server from `address_list`, and send a READ or WRITE operation using the filehandle specified in `fh`.

Consider the following example (which is the same as the sparse packing example, except for the filehandle list):

Suppose we have a device address consisting of seven data servers, arranged in three equivalence ([Section 13.5](#)) classes:

```
{ A, B, C, D }, { E }, { F, G }
```

Where A through G are network addresses.

Then

```
nflda_multipath_ds_list<> = { A, B, C, D }, { E }, { F, G }
```

i.e.

```
nflda_multipath_ds_list[0] = { A, B, C, D }
```

```
nflda_multipath_ds_list[1] = { E }
```

```
nflda_multipath_ds_list[2] = { F, G }
```

Suppose the striping index array is:

```
nfl_da_stripe_indices<> = { 2, 0, 1, 0 }
```

Now suppose the client gets a layout which has a device ID that maps to the above device address. The initial index,

```
nfl_first_stripe_index = 2,
```

and

```
nfl_fh_list = { 0x67, 0x37, 0x87, 0x36 }.
```

The interesting examples for dense packing are SU1 and SU3, because each stripe unit refers to the same data server list, yet MUST use a different filehandle. If the client wants to write to SU1, the set of valid { network address, filehandle } combinations for SU_i are determined by:

```
nfl_first_stripe_index = 2
```

So

```
j = (1 + 2) % 4 = 3
```

```
idx = nfl_da_stripe_indices[j]
```

```
= nfl_da_stripe_indices[3]
```

```
= 0
```

So

```
nfl_da_multipath_ds_list[0] = { A, B, C, D }
```

and

```
nfl_fh_list[3] = { 0x36 }
```

The client can thus write SU1 to { 0x36, { A, B, C, D }, }.

For SU3, $j = (3 + 2) \% 4 = 1$, and `nfl_da_stripe_indices[1] = 0`. Then `nfl_da_multipath_ds_list[0] = { A, B, C, D }`, and `nfl_fh_list[1] = 0x37`. The client can thus write SU3 to { 0x37, { A, B, C, D } }.

The destinations of the first thirteen storage units are:

Internet-Draft

NFSv4.1

September 2008

SUi	filehandle	data servers
0	87	E
1	36	A,B,C,D
2	67	F,G
3	37	A,B,C,D
4	87	E
5	36	A,B,C,D
6	67	F,G
7	37	A,B,C,D
8	87	E
9	36	A,B,C,D
10	67	F,G
11	37	A,B,C,D
12	87	E

[13.4.4.](#) Sparse and Dense Stripe Unit Packing

The flag `NFL4_UFLG_DENSE` of the `nfl_util4` data type (field `nflh_util` of the data type `nfsv4_1_file_layouthint4` and field `nfl_util` of data type `nfsv4_1_file_layout_ds_addr4`) specifies how the data is packed within the data file on a data server. It allows for two different data packings: sparse and dense. The packing 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.

If `nfl_util & NFL4_UFLG_DENSE` is zero, this means that sparse packing is being used. Hence 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 there is a pattern with three stripe units, the stripe unit size is a 4096 bytes, and there are three data servers in the pattern, then the file in data server 1 will have stripe units 0, 3, 6, 9, ... filled, data server 2's file will have stripe units 1, 4, 7, 10, ... filled, and data server 3's file will have stripe units 2, 5, 8, 11, ... filled. The unfilled stripe units of each file will be holes, hence the files in each data server are sparse.

If sparse packing is being used and 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 3 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 sparse packing.

If `nfl_util & NFL4_UFLG_DENSE` is one, this means that dense packing is being used and the data server files have no holes. Dense packing might be selected because the data server does not (efficiently) support holey files, or because the data server cannot recognize read-ahead unless there are no holes. If dense packing 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 sparse packing example, the corresponding dense packing would have all stripe units of all data files filled. Logical stripe units 0, 3, 6, ... of the file would live on stripe units 0, 1, 2, ... of the file of data server 1, logical stripe units 1, 4, 7, ... of the file would live on stripe units 0, 1, 2, ... of the file of data server 2, and logical stripe units 2, 5, 8, ... of the file would live on stripe units 0, 1, 2, ... of the file of data server 3.

Because dense packing 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 data servers need not know the file's striping pattern.

The calculation to determine the byte offset within the data file for dense data server layouts is:

```
stripe_width = stripe_unit_size * N;
  where N = number of elements in nfl_da_stripe_indices.

relative_offset = file_offset - nfl_pattern_offset;

data_file_offset = floor(relative_offset / stripe_width)
  * stripe_unit_size
  + relative_offset % stripe_unit_size
```

If dense packing is being used, and a data server appears more than once in a striping pattern, then to distinguish one stripe unit from another, the data server MUST use a different filehandle. Let's

suppose there are two data servers. Logical stripe units 0, 3, 6 are served by data server 1, logical stripe units 1, 4, 7 are served by data server 2, and logical stripe units 2, 5, 8 are also served by data server 2. Unless data server 2 has two filehandles (each referring to a different data file), then, for example, a write to logical stripe unit 1 overwrites the write to logical stripe unit 2, because both logical stripe units are located in the same stripe unit (0) of data server 2.

[13.5.](#) Data Server Multipathing

The NFSv4.1 file layout supports multipathing to multiple data server addresses. Data server-level multipathing is used for bandwidth

scaling via trunking ([Section 2.10.5](#)) and for higher availability of use in the case of a data server failure. Multipathing allows the client to switch to another data server address which may that of 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, each element of the `nfl_da_multipath_ds_list` contains an array of one more data server network addresses. This array (data type `multipath_list4`) represents a list of data servers (each identified by a network address), with it being possible that some data servers will appear in the list multiple times.

The client is free to use any of the network addresses as a destination to send data server requests. If some network addresses are less optimal paths to the data than others, then the MDS SHOULD NOT include those network addresses in an element of `nfl_da_multipath_ds_list`. If less optimal network addresses exist to provide fail over, the RECOMMENDED method to offer the addresses is to provide them in a replacement device ID to device address mapping, or a replacement device ID. When a client finds that no data server in an element of `nfl_da_multipath_ds_list` responds, it SHOULD send a `GETDEVICEINFO` to attempt to replace the existing device ID to device address mappings. If the MDS detects that all data servers represented by an element of `nfl_da_multipath_ds_list` are unavailable, the MDS SHOULD send a `CB_NOTIFY_DEVICEID` (if the client has indicated it wants device ID notifications for changed device IDs) to change the device ID to device address mappings to the available data

servers. If the device ID itself must be replaced, the MDS SHOULD recall all layouts with the device ID, and thus force the client to get new layouts and device ID mappings via LAYOUTGET and GETDEVICEINFO.

Generally if two network addresses appear in an element of `nfl_da_multipath_ds_list` they will designate the same data server and the two data server addresses will support the implementation client ID or session trunking (the latter is RECOMMENDED) as defined in [Section 2.10.5](#), and the two data server addresses will share the same server owner, or major ID of the server owner. It is not always necessary for the two data server addresses to designate the same server with trunking being used. For example the data could be read-only, and the data consist of exact replicas.

[13.6](#). Operations Sent to NFSv4.1 Data Servers

Clients accessing data on an NFSv4.1 data server MUST send only the NULL procedure and COMPOUND procedures whose operations are taken only from two restricted subsets of the operations defined as valid

NFSv4.1 operations. Clients MUST use the filehandle specified by the layout when accessing data on NFSv4.1 data servers.

The first of these operation subsets consist of management operations where the current filehandle is not relevant. This subset consists of the BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CREATE_SESSION, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, SECINFO_NO_NAME, SET_SSV, and SEQUENCE operations. The client may use these operations in order to set up and maintain the appropriate client IDs, sessions, and security contexts involved in communication with the data server. Henceforth these will be referred to as data-server housekeeping operations.

The second subset consists of COMMIT, READ, WRITE, and PUTFH, These operations must be used with a current filehandle specified by the layout. In the case of PUTFH, the new current filehandle must be one taken from the layout. Henceforth, these will be referred to as data-server I/O operations. As described in [Section 12.5.1](#), a client MUST NOT send an I/O to a data server for which it does not hold a valid layout; the data server MUST reject such an I/O.

Unless the server has a concurrent non-data-server personality, i.e. EXCHANGE_ID results returned (EXCHGID4_FLAG_USE_PNFS_DS | EXCHGID4_FLAG_USE_PNFS_MDS) or (EXCHGID4_FLAG_USE_PNFS_DS | EXCHGID4_FLAG_USE_NON_PNFS), see [Section 13.1](#), any use of operations other than those specified in the two subsets above MUST return NFS4ERR_NOTSUPP to the client.

When the server has concurrent data server and non-data-server personalities, each COMPOUND sent by the client MUST be constructed so that it is appropriate to one of the two personalities, and must not contain operations directed to a mix of those personalities. The server MUST enforce this. To understand the constraints, operations within a COMPOUND are divided into the following three classes:

1. An operation which is ambiguous regarding its personality assignment. These include all of the data-server housekeeping operations. Additionally, if the server has assigned filehandles so that the ones defined by the layout are the same as those used by the metadata server, all operations in the second class are within this group unless a stateid used is incompatible with a data-server personality in that it is a special stateid or has a non-zero seqid field.
2. An operation which is referable to the data server personality. These are data-server I/O operations where the filehandle is one that can only be validly directed to the data-server personality.

3. An operation which is referable to the non-data-server personality. These include all COMPOUND operations that are neither data-server housekeeping nor data-server I/O operations plus data-server I/O operations where the current fh (or the one to be made the current fh in the case of PUTFH) is one that is only valid on the metadata server or where a stateid is used that is incompatible with the data server, i.e. is a special stateid or has a non-zero seqid value.

When a COMPOUND first executes an operation from class 3 above, it acts as a normal COMPOUND on any other server and the data server personality ceases to be relevant. There are no special restrictions on the operations in the COMPOUND to limit them to those for a data server. When a PUTFH is done, filehandles derived from the layout

are not valid. If their format is not normally acceptable, then NFS4ERR_BADHANDLE MUST result. Similarly, current filehandles for other operations do not accept filehandles derived from layouts and are not normally usable on the metadata server. Using these will result in NFS4ERR_STALE.

When a COMPOUND first executes an operation from class 2, which would be PUTFH where the filehandle is one from a layout, the COMPOUND henceforth is interpreted with respect to the data server personality. Operations outside the two classes discussed above MUST result in NFS4ERR_NOTSUPP. Filehandles are validated using the rules of the data server, resulting in NFS4ERR_BADHANDLE and/or NFS4ERR_STALE even when they would not normally do so when addressed to the non-data-server personality. Stateids must obey the rules of the data server in that any use of special stateids or stateids with non-zero seqid values must result in NFS4ERR_BAD_STATEID.

Until the server first executes an operation from class 2 or class 3, the client MUST NOT depend on the operation being executed by either the data-server or the non-data-server personality. The server MUST pick one personality consistently for a given COMPOUND, with the only possible transition being a single one when the first operation from class 2 or class 3 is executed.

Because of the complexity induced by assigning filehandles so they can be used on both a data server and a metadata server, it is RECOMMENDED that where the same server can have both personalities, the server assign separate unique filehandles to both personalities. This makes it unambiguous for which server a given request is intended.

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 12.5.4.2](#)). [Section 13.10](#), describes the mechanism by which the client is to handle data server files that do not reflect the metadata server's size.

The file layout provides two alternate means of providing for the commit of data written through data servers. The flag `NFL4_UFLG_COMMIT_THRU_MDS` in the field `nfl_util` of the file layout (data type `nfsv4_1_file_layout4`) is an indication from the metadata server to the client of the REQUIRED way of performing COMMIT, either by sending the COMMIT to the data server or the metadata server. These two methods of dealing with the issue correspond to broad styles of implementation for a pNFS server supporting the files layout type.

- o When the flag is FALSE, COMMIT operations MUST to be sent to the data server to which the corresponding WRITE operations were sent. This approach is most useful when striping of files is implemented as part of pNFS server, with the individual data servers each implementing their own file systems.
- o When the flag is TRUE, COMMIT operations MUST be sent to the metadata server, rather than to the individual data servers. This approach is most useful when the pNFS server is implemented on top of a clustered file system. In such an implementation, sending COMMIT's to multiple data servers may result in repeated writes of metadata blocks as each individual COMMIT is executed, to the detriment of write performance. Sending a single COMMIT to the metadata server can provide more efficiency when there exists a clustered file system capable of implementing such a co-ordinated COMMIT.

If `nfl_util` & `NFL4_UFLG_COMMIT_THRU_MDS` is TRUE, then in order to maintain the current NFSv4.1 commit and recovery model, the data servers MUST return a common `writeverf` verifier in all WRITE responses for a given file layout, and the metadata server's COMMIT implementation must return the same `writeverf`. 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.

Note that if the layout specified dense packing, then the offset used to a COMMIT to the MDS may differ than that of an offset used to a COMMIT to the data server.

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 re-send 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 `nfl_util & NFL4_UFLG_COMMIT_THRU_MDS` is FALSE, sending a COMMIT to the metadata server might have no effect. If `nfl_util & NFL4_UFLG_COMMIT_THRU_MDS` is FALSE, a COMMIT sent to the metadata server should be used only to commit data that was written to the metadata server. See [Section 12.7.6](#) for recovery options.

[13.8.](#) 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. 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.

[13.9.](#) Metadata and Data Server State Coordination

[13.9.1.](#) Global Stateid Requirements

When the client sends I/O to a data server, the stateid used MUST NOT be a layout stateid as returned by `LAYOUTGET` or sent by `CB_LAYOUTRECALL`. Permitted stateids are based on one of the following: an open stateid (the stateid field of data type `OPEN4resok` as returned by `OPEN`), a delegation stateid (the stateid field of data types `open_read_delegation4` and `open_write_delegation4` as returned by `OPEN` or `WANT_DELEGATION`, or as sent by `CB_PUSH_DELEG`), or a stateid returned by the `LOCK` or `LOCKU` operations. The stateid sent to the data server MUST be sent with the seqid set to zero, indicating the most current version of that stateid, rather than indicating a

Internet-Draft

NFSv4.1

September 2008

specific non-zero seqid value. In no case is the use of special stateid values allowed.

The stateid used for I/O MUST have the same effect and be subject to the same validation on a 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 13.9.2](#). Depending on when stateids are propagated, the existence of a valid stateid on the data server may act as proof of a valid layout.

Clients performing I/O operations need to select an appropriate stateid based on the locks (including opens and delegations) held by the client and the various types of state-owners issuing the I/O requests. The rules for doing so when referencing data servers are somewhat different from those discussed in [Section 8.2.5](#) which apply when accessing metadata servers.

The following rules, applied in order of decreasing priority, govern the selection of the appropriate stateid:

- o If the client holds a delegation for the file in question, the delegation stateid should be used.
- o Otherwise, there must be an open stateid for the current open-owner, and that open stateid for the open file in question is used, unless mandatory locking, prevents that. See below.
- o If the data server had previously responded with NFS4ERR_LOCKED to use of the open stateid, then the client should use the lock stateid whenever one exists for that open file with the current lock-owner.
- o Special stateids should never be used and if used the data server MUST reject the I/O with an NFS4ERR_BAD_STATEID error.

[13.9.2](#). 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, 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.

[13.9.2.1](#). 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. The effect MUST be the same as if the mandatory lock state were synchronously propagated to the data servers, even though the details of the control protocol may avoid actual transfer of the state under certain circumstances.

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. 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 byte-range 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.

If a client gets back an NFS4ERR_LOCKED error from a data server, this is an indication that mandatory byte-range locking is in force. The client recovers from this by getting a byte-range lock that covers the affected range and re-sends the I/O with the stateid of the byte-range lock.

[13.9.2.2](#). Open and Deny Mode Validation

Open and deny mode validation **MUST** be performed against the open and deny mode(s) held by the data servers. When access is reduced or a deny mode made more restrictive (because of CLOSE or DOWNGRADE) the data server **MUST** prevent any I/Os that would be denied if performed on the metadata server. When access is expanded, the data server **MUST** make sure that no requests are subsequently rejected because of open or deny issues that no longer apply, given the previous relaxation.

[13.9.2.3](#). 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 12.5.4](#) 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 18.16.4](#)) does not impose any requirement that I/O operations on an open file have the same credentials as the OPEN itself (unless EXCHGID4_FLAG_BIND_PRINC_STATEID is set when

EXCHANGE_ID creates the client ID) 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.

[13.10.](#) 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 byte 131072; the client then seeks to the beginning of the file and reads byte 100. The client should receive 0s back as a result of the READ. However, if the READ falls on a data server other than the one that received client's original

Shepler, et al.

Expires March 9, 2009

[Page 323]

Internet-Draft

NFSv4.1

September 2008

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.

[13.11.](#) Layout Revocation and Fencing

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 LAYOUT4_NFSV4_1_FILES layout type can prevent all I/Os to data servers from being executed after lease expiration, without relying on a precise client lease timer and without requiring data servers to maintain lease timers. However, while LAYOUT4_NFSV4_1_FILES pNFS server is free to deny the client all access to the data servers, because it supports revocation of layouts, it is also free to perform a denial on a per file basis only when revoking a layout.

In addition to lease expiration, the reasons a layout can be revoked include: client fails to respond to a CB_LAYOUTRECALL, the metadata server restarts, or administrative intervention. Regardless of the reason, once a client's layout has been revoked, the pNFS server MUST prevent the client from issuing I/O for the affected file from and to all data servers, in other words, it MUST fence the client from the affected file on the data servers.

Fencing 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 with NFS4ERR_PNFS_NO_LAYOUT. The server can simply check the stateid, and

additionally, make the data filehandle stale if the layout specified a data filehandle that is different from the metadata server's filehandle for the file (see the `nfl_fh_list` description in [Section 13.3](#)).

Before the metadata server takes any action to invalidate layout state given out by a previous instance, it must make sure that all layout state 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.

[13.12.](#) Security Considerations for the File Layout Type

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 13.9.2](#) for more details.

The methods for authentication, integrity, and privacy for file layout-based data servers are the same as those used by 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 the secinfo4 values across a metadata server and its data servers differ for a specific file, the mapping of the principal to the server's internal user identifier MUST be the same in order for the access control checks based on ACL, mode, open and deny mode, and mandatory locking to be consistent across on the pNFS server.

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 NFSv4.1 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 [[14](#)] allows for this type of access and follows

the policy described in "IETF Policy on Character Sets and Languages", [RFC2277](#) [15].

[RFC3454](#) [16], otherwise known 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 NFSv4.1 stringprep profiles. Much of the terminology used for the remainder of this section comes from stringprep.

There are three UTF-8 string types defined for NFSv4.1: `utf8str_cs`, `utf8str_cis`, and `utf8str_mixed`. Separate profiles are defined for each. Each profile defines the following, as required by stringprep:

- o The intended applicability of the profile
- o The character repertoire that is the input and output to stringprep (which is Unicode 3.2 for referenced version of stringprep)
- o The mapping tables from stringprep used (as described in [section 3](#) of stringprep)
- o Any additional mapping tables specific to the profile
- o The Unicode normalization used, if any (as described in [section 4](#) of stringprep)
- o The tables from stringprep listing of characters that are prohibited as output (as described in [section 5](#) of stringprep)
- o The bidirectional string testing used, if any (as described in [section 6](#) of stringprep)
- o Any additional characters that are prohibited as output specific to the profile

Stringprep discusses Unicode characters, whereas NFSv4.1 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](#) [17].

[14.1.](#) Stringprep profile for the utf8str_cs type

Every use of the utf8str_cs type definition in the NFSv4 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 NFSv4.1 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 NFSv4.1 server will need to either:

- o disallow the creation of a second name if its post processed form collides with that of an existing name, or
- o 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 NFSv4.1 file server supports the case_insensitive file system attribute, and if case_insensitive is TRUE, the NFSv4.1 server MUST use Table B.2 (in addition to Table B1) when processing utf8str_cs strings, and the NFSv4.1 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 NFSv4.1 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 NFSv4.1 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 NFSv4.1 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

Table B.2

[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](#).

Shepler, et al.

Expires March 9, 2009

[Page 329]

Internet-Draft

NFSv4.1

September 2008

[14.3.](#) Stringprep profile for the utf8str_mixed type

Every use of the utf8str_mixed type definition in the NFSv4.1 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 NFSv4.1 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

Shepler, et al.

Expires March 9, 2009

[Page 330]

Internet-Draft

NFSv4.1

September 2008

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 Capabilities

```
const FSCHARSET_CAP4_CONTAINS_NON_UTF8 = 0x1;
const FSCHARSET_CAP4_ALLWS_ONLY_UTF8  = 0x2;
```

```
typedef uint32_t      fs_charset_cap4;
```

Because some operating environments and file systems do not enforce

character set encodings, NFSv4.1 supports the `fs_charset_cap` attribute ([Section 5.8.2.11](#)) that indicates to the client a file system's UTF-8 capabilities. The attribute is an integer containing a pair of flags. The first flag is `FSCHARSET_CAP4_CONTAINS_NON_UTF8`, which, if set to one tells the client the file system contains non-UTF-8 characters, and the server will not convert non-UTF characters to UTF-8 if the client reads a symlink or directory, nor will operations with component names or pathnames in the arguments convert the strings to UTF-8. The second flag is `FSCHARSET_CAP4_ALLOWS_ONLY_UTF8` which if set to one, indicates that the server will accept (and generate) only UTF-8 characters on the file system. If `FSCHARSET_CAP4_ALLOWS_ONLY_UTF8` is set to one, `FSCHARSET_CAP4_CONTAINS_NON_UTF8` MUST be set to zero. `FSCHARSET_CAP4_ALLOWS_ONLY_UTF8` SHOULD always be set to one.

[14.5.](#) UTF-8 Related Errors

Where the client sends an invalid UTF-8 string, the server should return `NFS4ERR_INVALID` (see Table 5). 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 bytes on a file system that supports Unicode characters only), the server should return `NFS4ERR_BADCHAR`.

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 5). 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

(COMPOUND or CB_COMPOUND) request. A Compound request contains a number of NFS operations that have their results encoded in sequence in a Compound reply. The results of successful operations will consist of an NFS4_OK status followed by the encoded results of the operation. If an NFS operation fails, an error status will be entered in the reply and the Compound request will be terminated.

15.1. Error Definitions

Protocol Error Definitions

Error	Number	Description
NFS4_OK	0	Section 15.1.3.1
NFS4ERR_ACCESS	13	Section 15.1.6.1
NFS4ERR_ATTRNOTSUPP	10032	Section 15.1.15.1
NFS4ERR_ADMIN_REVOKED	10047	Section 15.1.5.1
NFS4ERR_BACK_CHAN_BUSY	10057	Section 15.1.12.1
NFS4ERR_BADCHAR	10040	Section 15.1.7.1
NFS4ERR_BADHANDLE	10001	Section 15.1.2.1
NFS4ERR_BADIOMODE	10049	Section 15.1.10.1
NFS4ERR_BADLAYOUT	10050	Section 15.1.10.2
NFS4ERR_BADNAME	10041	Section 15.1.7.2
NFS4ERR_BADOWNER	10039	Section 15.1.15.2
NFS4ERR_BADSESSION	10052	Section 15.1.11.1
NFS4ERR_BADSLOT	10053	Section 15.1.11.2
NFS4ERR_BADTYPE	10007	Section 15.1.4.1
NFS4ERR_BADXDR	10036	Section 15.1.1.1
NFS4ERR_BAD_COOKIE	10003	Section 15.1.1.2

NFS4ERR_BAD_HIGH_SLOT	10077	Section 15.1.11.3
NFS4ERR_BAD_RANGE	10042	Section 15.1.8.1
NFS4ERR_BAD_SEQID	10026	Section 15.1.16.1
NFS4ERR_BAD_SESSION_DIGEST	10051	Section 15.1.12.2
NFS4ERR_BAD_STATEID	10025	Section 15.1.5.2
NFS4ERR_CB_PATH_DOWN	10048	Section 15.1.11.4
NFS4ERR_CLID_INUSE	10017	Section 15.1.13.2
NFS4ERR_CLIENTID_BUSY	10074	Section 15.1.13.1
NFS4ERR_COMPLETE_ALREADY	10054	Section 15.1.9.1
NFS4ERR_CONN_NOT_BOUND_TO_SESSION	10055	Section 15.1.11.6
NFS4ERR_DEADLOCK	10045	Section 15.1.8.2

NFS4ERR_DEADSESSION	10078	Section 15.1.11.5
NFS4ERR_DELAY	10008	Section 15.1.1.3
NFS4ERR_DELEG_ALREADY_WANTED	10056	Section 15.1.14.1
NFS4ERR_DELEG_REVOKED	10087	Section 15.1.5.3
NFS4ERR_DENIED	10010	Section 15.1.8.3
NFS4ERR_DIRDELEG_UNAVAIL	10084	Section 15.1.14.2
NFS4ERR_DQUOT	69	Section 15.1.4.2
NFS4ERR_ENCR_ALG_UNSUPP	10079	Section 15.1.13.3
NFS4ERR_EXIST	17	Section 15.1.4.3
NFS4ERR_EXPIRED	10011	Section 15.1.5.4
NFS4ERR_FBIG	27	Section 15.1.4.4
NFS4ERR_FHEXPIRED	10014	Section 15.1.2.2
NFS4ERR_FILE_OPEN	10046	Section 15.1.4.5
NFS4ERR_GRACE	10013	Section 15.1.9.2
NFS4ERR_HASH_ALG_UNSUPP	10072	Section 15.1.13.4
NFS4ERR_INVALID	22	Section 15.1.1.4
NFS4ERR_IO	5	Section 15.1.4.6
NFS4ERR_ISDIR	21	Section 15.1.2.3
NFS4ERR_LAYOUTTRYLATER	10058	Section 15.1.10.3
NFS4ERR_LAYOUTUNAVAILABLE	10059	Section 15.1.10.4
NFS4ERR_LEASE_MOVED	10031	Section 15.1.16.2
NFS4ERR_LOCKED	10012	Section 15.1.8.4
NFS4ERR_LOCKS_HELD	10037	Section 15.1.8.5
NFS4ERR_LOCK_NOTSUPP	10043	Section 15.1.8.6
NFS4ERR_LOCK_RANGE	10028	Section 15.1.8.7
NFS4ERR_MINOR_VERS_MISMATCH	10021	Section 15.1.3.2
NFS4ERR_MLINK	31	Section 15.1.4.7
NFS4ERR_MOVED	10019	Section 15.1.2.4
NFS4ERR_NAME_TOO_LONG	63	Section 15.1.7.3
NFS4ERR_NOENT	2	Section 15.1.4.8
NFS4ERR_NOFILEHANDLE	10020	Section 15.1.2.5
NFS4ERR_NOMATCHING_LAYOUT	10060	Section 15.1.10.5
NFS4ERR_NOSPC	28	Section 15.1.4.9
NFS4ERR_NOTDIR	20	Section 15.1.2.6
NFS4ERR_NOTEMPTY	66	Section 15.1.4.10
NFS4ERR_NOTSUPP	10004	Section 15.1.1.5
NFS4ERR_NOT_ONLY_OP	10081	Section 15.1.3.3

NFS4ERR_NOT_SAME	10027	Section 15.1.15.3
NFS4ERR_NO_GRACE	10033	Section 15.1.9.3
NFS4ERR_NXIO	6	Section 15.1.16.3
NFS4ERR_OLD_STATEID	10024	Section 15.1.5.5

NFS4ERR_OPENMODE	10038	Section 15.1.8.8
NFS4ERR_OP_ILLEGAL	10044	Section 15.1.3.4
NFS4ERR_OP_NOT_IN_SESSION	10071	Section 15.1.3.5
NFS4ERR_PERM	1	Section 15.1.6.2
NFS4ERR_PNFS_IO_HOLE	10075	Section 15.1.10.6
NFS4ERR_PNFS_NO_LAYOUT	10080	Section 15.1.10.7
NFS4ERR_RECALLCONFLICT	10061	Section 15.1.14.3
NFS4ERR_RECLAIM_BAD	10034	Section 15.1.9.4
NFS4ERR_RECLAIM_CONFLICT	10035	Section 15.1.9.5
NFS4ERR_REJECT_DELEG	10085	Section 15.1.14.4
NFS4ERR_REP_TOO_BIG	10066	Section 15.1.3.6
NFS4ERR_REP_TOO_BIG_TO_CACHE	10067	Section 15.1.3.7
NFS4ERR_REQ_TOO_BIG	10065	Section 15.1.3.8
NFS4ERR_RESTOREFH	10030	Section 15.1.16.4
NFS4ERR_RETRY_UNCACHED_REP	10068	Section 15.1.3.9
NFS4ERR_RETURNCONFLICT	10086	Section 15.1.10.8
NFS4ERR_ROFS	30	Section 15.1.4.11
NFS4ERR_SAME	10009	Section 15.1.15.4
NFS4ERR_SHARE_DENIED	10015	Section 15.1.8.9
NFS4ERR_SEQUENCE_POS	10064	Section 15.1.3.10
NFS4ERR_SEQ_FALSE_RETRY	10076	Section 15.1.11.7
NFS4ERR_SEQ_MISORDERED	10063	Section 15.1.11.8
NFS4ERR_SERVERFAULT	10006	Section 15.1.1.6
NFS4ERR_STALE	70	Section 15.1.2.7
NFS4ERR_STALE_CLIENTID	10022	Section 15.1.13.5
NFS4ERR_STALE_STATEID	10023	Section 15.1.16.5
NFS4ERR_SYMLINK	10029	Section 15.1.2.8
NFS4ERR_TOOSMALL	10005	Section 15.1.1.7
NFS4ERR_TOO_MANY_OPS	10070	Section 15.1.3.11
NFS4ERR_UNKNOWN_LAYOUTTYPE	10062	Section 15.1.10.9
NFS4ERR_UNSAFE_COMPOUND	10069	Section 15.1.3.12
NFS4ERR_WRONGSEC	10016	Section 15.1.6.3
NFS4ERR_WRONG_CRED	10082	Section 15.1.6.4
NFS4ERR_WRONG_TYPE	10083	Section 15.1.2.9
NFS4ERR_XDEV	18	Section 15.1.4.12

Table 5

[15.1.1.1](#). General Errors

This section deals with errors that are applicable to a broad set of different purposes.

[15.1.1.1.](#) NFS4ERR_BADXDR (Error Code 10036)

The arguments for this operation do not match those specified in the XDR definition. This includes situations in which the request ends before all the arguments have been seen. Note that this error applies when fixed enumerations (these include booleans) have a value within the input stream which is not valid for the enum. A replier may pre-parse all operations for a Compound procedure before doing any operation execution and return RPC-level XDR errors in that case.

[15.1.1.2.](#) NFS4ERR_BAD_COOKIE (Error Code 10003)

Used for operations that provide a set of information indexed by some quantity provided by the client or cookie sent by the server for an earlier invocation. Where the value cannot be used for its intended purpose, this error results.

[15.1.1.3.](#) NFS4ERR_DELAY (Error Code 10008)

For any of a number of reasons, the replier could not process this operation in what was deemed a reasonable time. The client should wait and then try the request with a new slot and sequence value.

Some example of situations that might lead to this situation:

- o A server that supports hierarchical storage receives a request to process a file that had been migrated.
- o An operation requires a delegation recall to proceed and waiting for this delegation recall makes processing this request in a timely fashion impossible.

In such cases, the error NFS4ERR_DELAY allows these preparatory operations to proceed without holding up client resources such as a session slot. After delaying for period of time, the client can then re-send the operation in question (but not with the same slot ID and sequence ID; one or both MUST be different on the re-send).

Note that without the ability to return NFS4ERR_DELAY and the client's willingness to re-send when receiving it, deadlock might well result. E.g., if a recall is done, and if the delegation return or operations preparatory to delegation return are held up by other operations that need the delegation to be returned, session slots might not be available. The result could be deadlock.

Internet-Draft

NFSv4.1

September 2008

[15.1.1.4.](#) NFS4ERR_INVALID (Error Code 22)

The arguments for this operation are not valid for some reason, even though they do match those specified in the XDR definition for the request.

[15.1.1.5.](#) NFS4ERR_NOTSUPP (Error Code 10004)

Operation not supported, either because the operation is an OPTIONAL one and is not supported by this server or because the operation MUST NOT be implemented in the current minor version.

[15.1.1.6.](#) NFS4ERR_SERVERFAULT (Error Code 10006)

An error occurred on the server which does not map to any of the specific legal NFSv4.1 protocol error values. The client should translate this into an appropriate error. UNIX clients may choose to translate this to EIO.

[15.1.1.7.](#) NFS4ERR_TOOSMALL (Error Code 10005)

Used where an operation returns a variable amount of data, with a limit specified by the client. Where the data returned cannot be fit within the limit specified by the client, this error results.

[15.1.2.](#) Filehandle Errors

These errors deal with the situation in which the current or saved filehandle, or the filehandle passed to PUTFH intended to become the current filehandle, is invalid in some way. This includes situations in which the filehandle is a valid filehandle in general but is not of the appropriate object type for the current operation.

Where the error description indicates a problem with the current or saved filehandle, it is to be understood that filehandles are only checked for the condition if they are implicit arguments of the operation in question.

[15.1.2.1.](#) NFS4ERR_BADHANDLE (Error Code 10001)

Illegal NFS filehandle for the current server. The current file handle failed internal consistency checks. Once accepted as valid (by PUTFH), no subsequent status change can cause the filehandle to generate this error.

[15.1.2.2.](#) NFS4ERR_FHEXPIRED (Error Code 10014)

A current or saved filehandle which is an argument to the current operation is volatile and has expired at the server.

[15.1.2.3.](#) NFS4ERR_ISDIR (Error Code 21)

The current or saved filehandle designates a directory when the current operation does not allow a directory to be accepted as the target of this operation.

[15.1.2.4.](#) NFS4ERR_MOVED (Error Code 10019)

The file system which contains the current filehandle object is not present at the server. It may have been relocated, migrated to another server or may have never been present. The client may obtain the new file system location by obtaining the "fs_locations" or "fs_locations_info" attribute for the current filehandle. For further discussion, refer to [Section 11.2](#)

[15.1.2.5.](#) NFS4ERR_NOFILEHANDLE (Error Code 10020)

The logical current or saved filehandle value is required by the current operation and is not set. This may be a result of a malformed COMPOUND operation (i.e. no PUTFH or PUTROOTFH before an operation that requires the current filehandle be set).

[15.1.2.6.](#) NFS4ERR_NOTDIR (Error Code 20)

The current (or saved) filehandle designates an object which is not a directory for an operation in which a directory is required.

[15.1.2.7.](#) NFS4ERR_STALE (Error Code 70)

The current or saved filehandle value designating an argument to the current operation is invalid. The file referred to by that filehandle no longer exists or access to it has been revoked.

[15.1.2.8.](#) NFS4ERR_SYMLINK (Error Code 10029)

The current filehandle designates a symbolic link when the current operation does not allow a symbolic link as the target.

[15.1.2.9.](#) NFS4ERR_WRONG_TYPE (Error Code 10083)

The current (or saved) filehandle designates an object which is of an invalid type for the current operation and there is no more specific error (such as NFS4ERR_ISDIR or NFS4ERR_SYMLINK) that applies. Note

that in NFSv4.0, such situations generally resulted in the less specific error NFS4ERR_INVALID.

[15.1.3.](#) Compound Structure Errors

This section deals with errors that relate to overall structure of a Compound request (by which we mean to include both COMPOUND and CB_COMPOUND), rather than to particular operations.

There are a number of basic constraints on the operations that may appear in a Compound request. Sessions adds to these basic constraints by requiring a Sequence operation (either SEQUENCE or CB_SEQUENCE) at the start of the Compound.

[15.1.3.1.](#) NFS_OK (Error code 0)

Indicates the operation completed successfully, in that all of the constituent operations completed without error.

[15.1.3.2.](#) NFS4ERR_MINOR_VERS_MISMATCH (Error code 10021)

The minor version specified is not one that the current listener supports. This value is returned in the overall status for the Compound but is not associated with a specific operation since the results must specify a result count of zero.

[15.1.3.3.](#) NFS4ERR_NOT_ONLY_OP (Error Code 10081)

Certain operations, which are allowed to be executed outside of a session, must be the only operation within a COMPOUND. This error results when that constraint is not met.

[15.1.3.4.](#) NFS4ERR_OP_ILLEGAL (Error Code 10044)

The operation code is not a valid one for the current Compound procedure. The opcode in the result stream matched with this error is the ILLEGAL value, although the value that appears in the request stream may be different. Where an illegal value appears and the replier pre-parses all operations for a Compound procedure before doing any operation execution, an RPC-level XDR error may be returned in this case.

[15.1.3.5.](#) NFS4ERR_OP_NOT_IN_SESSION (Error Code 10071)

Most forward operations and all callback operations are only valid within the context of a session, so that the Compound request in question must begin with a Sequence operation. If an attempt is made to execute these operations outside the context of session, this

error results.

[15.1.3.6.](#) NFS4ERR_REP_TOO_BIG (Error Code 10066)

The reply to a Compound would exceed the channel's negotiated maximum response size.

[15.1.3.7.](#) NFS4ERR_REP_TOO_BIG_TO_CACHE (Error Code 10067)

The reply to a Compound would exceed the channel's negotiated maximum size for replies cached in the reply cache when the Sequence for the current request specifies that this request is to be cached.

[15.1.3.8.](#) NFS4ERR_REQ_TOO_BIG (Error Code 10065)

The Compound request exceeds the channel's negotiated maximum size for requests.

[15.1.3.9.](#) NFS4ERR_RETRY_UNCACHED_REP (Error Code 10068)

The requester has attempted a retry of a Compound which it previously requested not be placed in the reply cache.

[15.1.3.10.](#) NFS4ERR_SEQUENCE_POS (Error Code 10064)

A Sequence operation appeared in a position other than the first operation of a Compound request.

[15.1.3.11.](#) NFS4ERR_TOO_MANY_OPS (Error Code 10070)

The Compound request has too many operations, exceeding the count negotiated when the session was created.

[15.1.3.12.](#) NFS4ERR_UNSAFE_COMPOUND (Error Code 10068)

The client has sent a COMPOUND request with an unsafe mix of operations, specifically with a non-idempotent operation changing the current filehandle which is not followed by a GETFH.

[15.1.4.](#) File System Errors

These errors describe situations which occurred in the underlying file system implementation rather than in the protocol or any NFSv4.x feature.

[15.1.4.1.](#) NFS4ERR_BADTYPE (Error Code 10007)

An attempt was made to create an object with an inappropriate type specified to CREATE. This may be because the type is undefined, because it is a type not supported by the server, or because it is a type for which create is not intended such as a regular file or named attribute, for which OPEN is used to do the file creation.

[15.1.4.2.](#) NFS4ERR_DQUOT (Error Code 19)

Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

[15.1.4.3.](#) NFS4ERR_EXIST (Error Code 17)

A file of the specified target name (when creating, renaming or linking) already exists.

[15.1.4.4.](#) NFS4ERR_FBIG (Error Code 27)

File too large. The operation would have caused a file to grow beyond the server's limit.

[15.1.4.5.](#) NFS4ERR_FILE_OPEN (Error Code 10046)

The operation is not allowed because a file involved in the operation is currently open. Servers may, but are not required to disallow linking-to, removing, or renaming open files.

[15.1.4.6.](#) NFS4ERR_IO (Error Code 5)

Indicates that an I/O error occurred for which the file system was unable to provide recovery.

[15.1.4.7.](#) NFS4ERR_MLINK (Error Code 31)

The request would have caused the server's limit for the number of hard links a file may have to be exceeded.

[15.1.4.8.](#) NFS4ERR_NOENT (Error Code 2)

Indicates no such file or directory. The file or directory name specified does not exist.

[15.1.4.9.](#) NFS4ERR_NOSPC (Error Code 28)

Indicates no space left on device. The operation would have caused the server's file system to exceed its limit.

[15.1.4.10.](#) NFS4ERR_NOTEMPTY (Error Code 66)

An attempt was made to remove a directory that was not empty.

[15.1.4.11.](#) NFS4ERR_ROFS (Error Code 30)

Indicates a read-only file system. A modifying operation was attempted on a read-only file system.

[15.1.4.12.](#) NFS4ERR_XDEV (Error Code 18)

Indicates an attempt to do an operation, such as linking, that inappropriately crosses a boundary. This may be due to such boundaries as:

- o That between file systems (where the fsids are different).
- o That between different named attribute directories or between a named attribute directory and an ordinary directory.
- o That between regions of a file system that the file system implementation treats as separate (for example for space accounting purposes), and where cross-connection between the regions are not allowed.

[15.1.5.](#) State Management Errors

These errors indicate problems with the stateid (or one of the stateids) passed to a given operation. This includes situations in which the stateid is invalid as well as situations in which the stateid is valid but designates revoked locking state. Depending on the operation, the stateid when valid may designate opens, byte-range locks, file or directory delegations, layouts, or device maps.

[15.1.5.1.](#) NFS4ERR_ADMIN_REVOKED (Error Code 10047)

A stateid designates locking state of any type that has been revoked due to administrative interaction, possibly while the lease is valid.

[15.1.5.2.](#) NFS4ERR_BAD_STATEID (Error Code 10026)

A stateid does not properly designate any valid state. See [Section 8.2.4](#) and [Section 8.2.3](#) for a discussion of how stateids are validated.

[15.1.5.3.](#) NFS4ERR_DELEG_REVOKED (Error Code 10087)

A stateid designates recallable locking state of any type (delegation or layout) that has been revoked due to the failure of the client to return the lock when it was recalled.

[15.1.5.4.](#) NFS4ERR_EXPIRED (Error Code 10011)

A stateid designates locking state of any type that has been revoked due to expiration of the client's lease, either immediately upon lease expiration, or following a later request for a conflicting lock.

[15.1.5.5.](#) NFS4ERR_OLD_STATEID (Error Code 10024)

A stateid with a non-zero seqid value does match the current seqid for the state designated by the user.

[15.1.6.](#) Security Errors

These are the various permission-related errors in NFSv4.1.

[15.1.6.1.](#) NFS4ERR_ACCESS (Error Code 13)

Indicates permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with NFS4ERR_PERM ([Section 15.1.6.2](#)), which restricts itself to owner or privileged user permission failures, and NFS4ERR_WRONG_CRED ([Section 15.1.6.4](#)) which deals with appropriate permission to delete or modify transient objects, based on the credentials of the user that created them.

[15.1.6.2.](#) NFS4ERR_PERM (Error Code 1)

Indicates requester is not the owner. The operation was not allowed because the caller is neither a privileged user (root) nor the owner of the target of the operation.

[15.1.6.3.](#) NFS4ERR_WRONGSEC (Error Code 10016)

Indicates that the security mechanism being used by the client for the operation does not match the server's security policy. The client should change the security mechanism being used and re-send the operation (but not with the same slot ID and sequence ID; one or both MUST be different on the re-send). SECINFO and SECINFO_NO_NAME can be used to determine the appropriate mechanism.

Internet-Draft

NFSv4.1

September 2008

[15.1.6.4.](#) NFS4ERR_WRONG_CRED (Error Code 10082)

An operation manipulating state was attempted by a principal that was not allowed to modify that piece of state.

[15.1.7.](#) Name Errors

Names in NFSv4 are UTF-8 strings. When the strings are not valid UTF-8 or are of length zero, the error NFS4ERR_INVALID results. Besides this, there are a number of other errors to indicate specific problems with names.

[15.1.7.1.](#) NFS4ERR_BADCHAR (Error Code 10040)

A UTF-8 string contains a character which is not supported by the server in the context in which it being used.

[15.1.7.2.](#) NFS4ERR_BADNAME (Error Code 10041)

A name string in a request consisted of valid UTF-8 characters supported by the server but the name is not supported by the server as a valid name for current operation. An example might be creating a file or directory named ".." on a server whose file system uses that name for links to parent directories.

[15.1.7.3.](#) NFS4ERR_NAMETOOLONG (Error Code 63)

Returned when the filename in an operation exceeds the server's implementation limit.

[15.1.8.](#) Locking Errors

This section deal with errors related to locking, both as to share reservations and byte-range locking. It does not deal with errors specific to the process of reclaiming locks. Those are dealt with in the next section.

[15.1.8.1.](#) NFS4ERR_BAD_RANGE (Error Code 10042)

The range for a LOCK, LOCKT, or LOCKU operation is not appropriate to the allowable range of offsets for the server. E.g., this error results when a server which only supports 32-bit ranges receives a

range that cannot be handled by that server. (See [Section 18.10.3](#)).

[15.1.8.2](#). NFS4ERR_DEADLOCK (Error Code 10045)

The server has been able to determine a file locking deadlock condition for a blocking lock request.

[15.1.8.3](#). NFS4ERR_DENIED (Error Code 10010)

An attempt to lock a file is denied. Since this may be a temporary condition, the client is encouraged to re-send the lock request (but not with the same slot ID and sequence ID; one or both MUST be different on the re-send) until the lock is accepted. See [Section 9.6](#) for a discussion of the re-send.

[15.1.8.4](#). NFS4ERR_LOCKED (Error Code 10012)

A read or write operation was attempted on a file where there was a conflict between the I/O and an existing lock:

- o There is a share reservation inconsistent with the I/O being done.
- o The range to be read or written intersects an existing mandatory byte range lock.

[15.1.8.5](#). NFS4ERR_LOCKS_HELD (Error Code 10037)

An operation was prevented by the unexpected presence of locks.

[15.1.8.6](#). NFS4ERR_LOCK_NOTSUPP (Error Code 10043)

A locking request was attempted which would require the upgrade or downgrade of a lock range already held by the owner when the server does not support atomic upgrade or downgrade of locks.

[15.1.8.7](#). NFS4ERR_LOCK_RANGE (Error Code 10028)

A lock request is operating on a range that overlaps in part a currently held lock for the current lock owner and does not precisely match a single such lock where the server does not support this type of request, and thus does not implement POSIX locking semantics. See [Section 18.10.4](#), [Section 18.11.4](#), and [Section 18.12.4](#) for a

discussion of how this applies to LOCK, LOCKT, and LOCKU respectively.

[15.1.8.8.](#) NFS4ERR_OPENMODE (Error Code 10038)

The client attempted a READ, WRITE, LOCK or other operation not sanctioned by the stateid passed (e.g. writing to a file opened only for read).

[15.1.8.9.](#) NFS4ERR_SHARE_DENIED (Error Code 10015)

An attempt to OPEN a file with a share reservation has failed because of a share conflict.

Shepler, et al.

Expires March 9, 2009

[Page 344]

Internet-Draft

NFSv4.1

September 2008

[15.1.9.](#) Reclaim Errors

These errors relate to the process of reclaiming locks after a server restart.

[15.1.9.1.](#) NFS4ERR_COMPLETE_ALREADY (Error Code 10054)

The client previously sent a successful RECLAIM_COMPLETE operation. An additional RECLAIM_COMPLETE operation is not necessary and results in this error.

[15.1.9.2.](#) NFS4ERR_GRACE (Error Code 10013)

The server is in its recovery or grace period which should at least match the lease period of the server. A locking request other than a reclaim could not be granted during that period.

[15.1.9.3.](#) NFS4ERR_NO_GRACE (Error Code 10033)

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.

[15.1.9.4.](#) NFS4ERR_RECLAIM_BAD (Error Code 10034)

A reclaim attempted by the client does not match the server's state consistency checks and has been rejected therefore as invalid.

[15.1.9.5.](#) NFS4ERR_RECLAIM_CONFLICT (Error Code 10035)

The reclaim attempted by the client has encountered a conflict and cannot be satisfied. Potentially indicates a misbehaving client, although not necessarily the one receiving the error. The misbehavior might be on the part of the client that established the lock with which this client conflicted.

[15.1.10.](#) pNFS Errors

This section deals with pNFS-related errors including those that are associated with using NFSv4.1 to communicate with a data server.

[15.1.10.1.](#) NFS4ERR_BADIOMODE (Error Code 10049)

An invalid or inappropriate layout iomode was specified.

[15.1.10.2.](#) NFS4ERR_BADLAYOUT (Error Code 10050)

The layout specified is invalid in some way. For LAYOUTCOMMIT, this indicates that the specified layout is not held by the client or is not of mode LAYOUTIOMODE4_RW. For LAYOUTGET, it indicates that a layout matching the client's specification as to minimum length cannot be granted.

[15.1.10.3.](#) NFS4ERR_LAYOUTTRYLATER (Error Code 10058)

Layouts are temporarily unavailable for the file. The client should re-send later (but not with the same slot ID and sequence ID; one or both MUST be different on the re-send).

[15.1.10.4.](#) NFS4ERR_LAYOUTUNAVAILABLE (Error Code 10059)

Returned when layouts are not available for the current file system

or the particular specified file.

[15.1.10.5](#). NFS4ERR_NOMATCHING_LAYOUT (Error Code 10060)

Returned when layouts are recalled and the client has no layouts matching the specification of the layouts being recalled.

[15.1.10.6](#). NFS4ERR_PNFS_IO_HOLE (Error Code 10075)

The pNFS client has attempted to read from or write to an illegal hole of a file of a data server that is using sparse packing. See [Section 13.4.4](#).

[15.1.10.7](#). NFS4ERR_PNFS_NO_LAYOUT (Error Code 10080)

The pNFS client has attempted to read from or write to a file (using a request to a data server) without holding a valid layout. This includes the case where the client had a layout, but the iomode does not allow a WRITE.

[15.1.10.8](#). NFS4ERR_RETURNCONFLICT (Error Code 10086)

A layout is unavailable due to an attempt to perform the LAYOUTGET before a pending LAYOUTRETURN on the file has been received. See [Section 12.5.5.2.1.3](#).

[15.1.10.9](#). NFS4ERR_UNKNOWN_LAYOUTTYPE (Error Code 10062)

The client has specified a layout type which is not supported by the server.

[15.1.11](#). Session Use Errors

This section deals with errors encountered in using sessions, that is, in issuing requests over them using the Sequence (i.e. either SEQUENCE or CB_SEQUENCE) operations.

[15.1.11.1](#). NFS4ERR_BADSESSION (Error Code 10052)

The specified session ID is unknown to the server to which the

operation is addressed.

[15.1.11.2.](#) NFS4ERR_BADSLOT (Error Code 10053)

The requester sent a Sequence operation that attempted to use a slot the replier does not have in its slot table. It is possible the slot may have been retired.

[15.1.11.3.](#) NFS4ERR_BAD_HIGH_SLOT (Error Code 10077)

The highest_slot argument in a Sequence operation exceeds the replier's enforced highest_slotid.

[15.1.11.4.](#) NFS4ERR_CB_PATH_DOWN (Error Code 10048)

There is a problem contacting the client via the callback path. The function of this error has been mostly superseded by the use of status flags in the reply to the SEQUENCE operation (see [Section 18.46](#)).

[15.1.11.5.](#) NFS4ERR_DEADSESSION (Error Code 10078)

The specified session is a persistent session which is dead and does not accept new requests or perform new operations on existing requests (in the case in which a request was partially executed before server restart).

[15.1.11.6.](#) NFS4ERR_CONN_NOT_BOUND_TO_SESSION (Error Code 10055)

A Sequence operation was sent on a connection that has not been associated with the specified session, where the client specified that connection association was to be enforced with SP4_MACH_CRED or SP4_SSV state protection.

[15.1.11.7.](#) NFS4ERR_SEQ_FALSE_RETRY (Error Code 10076)

The requester sent a Sequence operation with a slot ID and sequence ID that are in the reply cache, but the replier has detected that the retried request is not the same as the original request.

[15.1.11.8.](#) NFS4ERR_SEQ_MISORDERED (Error Code 10063)

The requester sent a Sequence operation with an invalid sequence ID.

[15.1.12.](#) Session Management Errors

This section deals with errors associated with requests used in session management.

[15.1.12.1.](#) NFS4ERR_BACK_CHAN_BUSY (Error Code 10057)

An attempt was made to destroy a session when the session cannot be destroyed because the server has callback requests outstanding.

[15.1.12.2.](#) NFS4ERR_BAD_SESSION_DIGEST (Error Code 10051)

The digest used in a SET_SSV request is not valid.

[15.1.13.](#) Client Management Errors

This sections deals with errors associated with requests used to create and manage client IDs.

[15.1.13.1.](#) NFS4ERR_CLIENTID_BUSY (Error Code 10074)

The DESTROY_CLIENTID operation has found there are sessions and/or unexpired state associated with the client ID to be destroyed.

[15.1.13.2.](#) NFS4ERR_CLID_INUSE (Error Code 10017)

While processing an EXCHANGE_ID operation, the server was presented with a co_ownerid field matches an existing client with valid leased state but the principal issuing the EXCHANGE_ID is different than that establishing the existing client. This indicates a (most likely due to chance) collision between clients. The client should recover by changing the co_ownerid and re-sending EXCHANGE_ID (but not with the same slot ID and sequence ID; one or both MUST be different on the re-send).

[15.1.13.3.](#) NFS4ERR_ENCR_ALG_UNSUPP (Error Code 10079)

An EXCHANGE_ID was sent which specified state protection via SSV, and where the set of encryption algorithms presented by the client did not include any supported by the server.

[15.1.13.4.](#) NFS4ERR_HASH_ALG_UNSUPP (Error Code 10072)

An EXCHANGE_ID was sent which specified state protection via SSV, and where the set of hashing algorithms presented by the client did not include any supported by the server.

[15.1.13.5.](#) NFS4ERR_STALE_CLIENTID (Error Code 10022)

A client ID not recognized by the server was passed to an operation. Note that unlike the case of NFSv4.0, client IDs are not passed explicitly to the server in ordinary locking operations and cannot result in this error. Instead, when there is a server restart, it is first manifested through an error on the associated session and the staleness of the client ID is detected when trying to associate a client ID with a new session.

[15.1.14.](#) Delegation Errors

This section deals with errors associated with requesting and returning delegations.

[15.1.14.1.](#) NFS4ERR_DELEG_ALREADY_WANTED (Error Code 10056)

The client has requested a delegation when it had already registered that it wants that same delegation.

[15.1.14.2.](#) NFS4ERR_DIRDELEG_UNAVAIL (Error Code 10084)

This error is returned when the server is unable or unwilling to provide a requested directory delegation.

[15.1.14.3.](#) NFS4ERR_RECALLCONFLICT (Error Code 10061)

A recallable object (i.e. a layout or delegation) is unavailable due to a conflicting recall operation for that object that is currently in progress.

[15.1.14.4.](#) NFS4ERR_REJECT_DELEG (Error Code 10085)

The callback operation invoked to deal with a new delegation has rejected it.

Internet-Draft

NFSv4.1

September 2008

[15.1.15.](#) Attribute Handling Errors

This section deals with errors specific to attribute handling within NFSv4.

[15.1.15.1.](#) NFS4ERR_ATTRNOTSUPP (Error Code 10032)

An attribute specified is not supported by the server. This error MUST NOT be returned by the GETATTR operation.

[15.1.15.2.](#) NFS4ERR_BADOWNER (Error Code 10039)

Returned when an owner or owner_group attribute value or the who field of an ace within an ACL attribute value cannot be translated to a local representation.

[15.1.15.3.](#) NFS4ERR_NOT_SAME (Error Code 10027)

This error is returned by the VERIFY operation to signify that the attributes compared were not the same as those provided in the client's request.

[15.1.15.4.](#) NFS4ERR_SAME (Error Code 10009)

This error is returned by the NVERIFY operation to signify that the attributes compared were the same as those provided in the client's request.

[15.1.16.](#) Obsoleted Errors

These errors MUST NOT be generated by any NFSv4.1 operation. This can be for a number of reasons.

- o The function provided by the error has been superseded by one of the status bits returned by the SEQUENCE operation.
- o The new session structure and associated change in locking have made the error unnecessary.
- o There has been a restructuring of some errors for NFSv4.1 which resulted in the elimination of certain of the errors.

[15.1.16.1.](#) NFS4ERR_BAD_SEQID (Error Code 10026)

The sequence number (seqid) in a locking request is neither the next expected number or the last number processed. These seqids are ignored in NFSv4.1.

[15.1.16.2](#). NFS4ERR_LEASE_MOVED (Error Code 10031)

A lease being renewed is associated with a file system that has been migrated to a new server. The error has been superseded by the SEQ4_STATUS_LEASE_MOVED status bit (see [Section 18.46](#)).

[15.1.16.3](#). NFS4ERR_NXIO (Error Code 5)

I/O error. No such device or address. This error is for errors involving block and character device access, but NFSv4.1 is not a device access protocol.

[15.1.16.4](#). NFS4ERR_RESTOREFH (Error Code 10030)

The RESTOREFH operation does not have a saved filehandle (identified by SAVEFH) to operate upon. In NFSv4.1, this error has been superseded by NFS4ERR_NOFILEHANDLE.

[15.1.16.5](#). NFS4ERR_STALE_STATEID (Error Code 10023)

A stateid generated by an earlier server instance was used. This error is moot in NFSv4.1 because all operations that take a stateid MUST be preceded by the SEQUENCE operation, and the earlier server instance is detected by the session infrastructure that supports SEQUENCE.

[15.2](#). Operations and their valid errors

This section contains a table which gives the valid error returns for each protocol operation. The error code NFS4_OK (indicating no error) is not listed but should be understood to be returnable by all operations with two important exceptions:

- o The operations which MUST NOT be implemented: OPEN_CONFIRM, RELEASE_LOCKOWNER, RENEW, SETCLIENTID, and SETCLIENTID_CONFIRM.

- o The invalid operation: ILLEGAL.

Valid error returns for each protocol operation

Operation	Errors
ACCESS	NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
BACKCHANNEL_CTL	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_INVAL, NFS4ERR_NOENT, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS
BIND_CONN_TO_SESSION	NFS4ERR_BADSESSION, NFS4ERR_BADXDR, NFS4ERR_BAD_SESSION_DIGEST, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_INVAL, NFS4ERR_NOT_ONLY_OP, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS
CLOSE	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION,

	NFS4ERR_DELAY, NFS4ERR_EXPIRED,
	NFS4ERR_FHEXPIRED, NFS4ERR_LOCKS_HELD,
	NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OLD_STATEID,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_WRONG_CRED

COMMIT	NFS4ERR_ACCESS, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_IO,
	NFS4ERR_ISDIR, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_SYMLINK,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_TYPE
CREATE	NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,
	NFS4ERR_BADCHAR, NFS4ERR_BADNAME,
	NFS4ERR_BADOWNER, NFS4ERR_BADTYPE,
	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_DQUOT,
	NFS4ERR_EXIST, NFS4ERR_FHEXPIRED,
	NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MLINK,
	NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,
	NFS4ERR_NOTDIR, NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_PERM, NFS4ERR_REP_TOO_BIG,

CREATE_SESSION	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNSAFE_COMPOUND
	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_INVAL,
	NFS4ERR_NOENT, NFS4ERR_NOT_ONLY_OP,
	NFS4ERR_NOSPC, NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_SEQ_MISORDERED,
	NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE_CLIENTID, NFS4ERR_TOOSMALL,
DELEGPURGE	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED
	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED

DELEGRETURN	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR,
	NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_DELEG_REVOKED,
	NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED,
	NFS4ERR_INVAL, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTSUPP,
	NFS4ERR_OLD_STATEID,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_WRONG_CRED
DESTROY_CLIENTID	NFS4ERR_BADXDR, NFS4ERR_CLIENTID_BUSY,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,

DESTROY_SESSION	NFS4ERR_NOT_ONLY_OP, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE_CLIENTID, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED NFS4ERR_BACK_CHAN_BUSY, NFS4ERR_BADSESSION, NFS4ERR_BADXDR, NFS4ERR_CB_PATH_DOWN, NFS4ERR_CONN_NOT_BOUND_TO_SESSION, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_NOT_ONLY_OP, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE_CLIENTID, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED
EXCHANGE_ID	NFS4ERR_BADCHAR, NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_ENCR_ALG_UNSUPP, NFS4ERR_HASH_ALG_UNSUPP, NFS4ERR_INVALID, NFS4ERR_NOENT, NFS4ERR_NOT_ONLY_OP, NFS4ERR_NOT_SAME, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS
FREE_STATEID	NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_LOCKS_HELD, NFS4ERR_OLD_STATEID, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED

GET_DIR_DELEGATION	NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DIRDELEG_UNAVAIL, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR, NFS4ERR_NOTSUPP, NFS4ERR_OP_NOT_IN_SESSION,
--------------------	--

GETATTR	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
	NFS4ERR_ACCESS, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,
	NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
GETDEVICEINFO	NFS4ERR_WRONG_TYPE
	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_INVALID,
	NFS4ERR_NOENT, NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_TOOSMALL, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNKNOWN_LAYOUTTYPE
GETDEVICELIST	NFS4ERR_BADXDR, NFS4ERR_BAD_COOKIE,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVALID,
	NFS4ERR_IO, NFS4ERR_NOFILEHANDLE,
	NFS4ERR_NOTSUPP, NFS4ERR_NOT_SAME,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNKNOWN_LAYOUTTYPE
GETFH	NFS4ERR_FHEXPIRED, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_STALE
ILLEGAL	NFS4ERR_BADXDR NFS4ERR_OP_ILLEGAL

LAYOUTGET

NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADIOMODE,
NFS4ERR_BADLAYOUT, NFS4ERR_BADXDR,
NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
NFS4ERR_DELEG_REVOKED, NFS4ERR_EXPIRED,
NFS4ERR_FBIG, NFS4ERR_FHEXPIRED,
NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO,
NFS4ERR_ISDIR NFS4ERR_MOVED,
NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTSUPP,
NFS4ERR_NO_GRACE,
NFS4ERR_OP_NOT_IN_SESSION,
NFS4ERR_RECLAIM_BAD,
NFS4ERR_RECLAIM_CONFLICT,
NFS4ERR_REP_TOO_BIG,
NFS4ERR_REP_TOO_BIG_TO_CACHE,
NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
NFS4ERR_STALE, NFS4ERR_SYMLINK,
NFS4ERR_TOO_MANY_OPS,
NFS4ERR_UNKNOWN_LAYOUTTYPE,
NFS4ERR_WRONG_CRED
NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,
NFS4ERR_BADIOMODE, NFS4ERR_BADLAYOUT,
NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID,
NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
NFS4ERR_DELEG_REVOKED, NFS4ERR_DQUOT,
NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,
NFS4ERR_INVAL, NFS4ERR_IO,
NFS4ERR_LAYOUTTRYLATER,
NFS4ERR_LAYOUTUNAVAILABLE, NFS4ERR_LOCKED,
NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,
NFS4ERR_NOSPC, NFS4ERR_NOTSUPP,
NFS4ERR_OLD_STATEID, NFS4ERR_OPENMODE,
NFS4ERR_OP_NOT_IN_SESSION,
NFS4ERR_RECALLCONFLICT,
NFS4ERR_REP_TOO_BIG,
NFS4ERR_REP_TOO_BIG_TO_CACHE,
NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
NFS4ERR_STALE, NFS4ERR_TOOSMALL,
NFS4ERR_TOO_MANY_OPS,
NFS4ERR_UNKNOWN_LAYOUTTYPE,
NFS4ERR_WRONG_TYPE

LAYOUTRETURN	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DELEG_REVOKED, NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_ISDIR, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTSUPP, NFS4ERR_NO_GRACE, NFS4ERR_OLD_STATEID, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS, NFS4ERR_UNKNOWN_LAYOUTTYPE, NFS4ERR_WRONG_CRED, NFS4ERR_WRONG_TYPE
LINK	NFS4ERR_ACCESS, NFS4ERR_BADCHAR, NFS4ERR_BADNAME, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DQUOT, NFS4ERR_EXIST, NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN, NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_ISDIR, NFS4ERR_IO, NFS4ERR_MLINK, NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC, NFS4ERR_NOTDIR, NFS4ERR_NOTSUPP, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_SYMLINK, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC, NFS4ERR_WRONG_TYPE, NFS4ERR_XDEV

Internet-Draft

NFSv4.1

September 2008

LOCK	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR, NFS4ERR_BAD_RANGE, NFS4ERR_BAD_STATEID, NFS4ERR_DEADLOCK, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DENIED, NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_ISDIR, NFS4ERR_LOCK_NOTSUPP, NFS4ERR_LOCK_RANGE, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_NO_GRACE, NFS4ERR_OLD_STATEID, NFS4ERR_OPENMODE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_RECLAIM_BAD, NFS4ERR_RECLAIM_CONFLICT, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_SYMLINK, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED, NFS4ERR_WRONG_TYPE
LOCKT	NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_RANGE, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DENIED, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_ISDIR, NFS4ERR_LOCK_RANGE, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS, NFS4ERR_STALE, NFS4ERR_SYMLINK, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED, NFS4ERR_WRONG_TYPE
LOCKU	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR, NFS4ERR_BAD_RANGE, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED, NFS4ERR_INVALID,

	NFS4ERR_LOCK_RANGE, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_WRONG_CRED

LOOKUP	NFS4ERR_ACCESS, NFS4ERR_BADCHAR,
	NFS4ERR_BADNAME, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVALID,
	NFS4ERR_IO, NFS4ERR_MOVED,
	NFS4ERR_NAME_TOO_LONG, NFS4ERR_NOENT,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_SYMLINK,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC
LOOKUPP	NFS4ERR_ACCESS, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,
	NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOENT,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_SYMLINK,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC
NVERIFY	NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,
	NFS4ERR_BADCHAR, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,
	NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,

NFS4ERR_REQ_TOO_BIG, NFS4ERR_SAME,
NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
NFS4ERR_TOO_MANY_OPS,
NFS4ERR_UNKNOWN_LAYOUTTYPE,
NFS4ERR_WRONG_TYPE

OPEN

NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,
NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR,
NFS4ERR_BADNAME, NFS4ERR_BADOWNER,
NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID,
NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
NFS4ERR_DELEG_ALREADY_WANTED,
NFS4ERR_DELEG_REVOKED, NFS4ERR_DQUOT,
NFS4ERR_EXIST, NFS4ERR_EXPIRED,
NFS4ERR_FBIG, NFS4ERR_FHEXPIRED,
NFS4ERR_GRACE, NFS4ERR_INVALID,
NFS4ERR_ISDIR, NFS4ERR_IO, NFS4ERR_MOVED,
NFS4ERR_NAMETOOLONG, NFS4ERR_NOENT,
NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,
NFS4ERR_NOTDIR, NFS4ERR_NO_GRACE,
NFS4ERR_OLD_STATEID,
NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_PERM,
NFS4ERR_RECLAIM_BAD,
NFS4ERR_RECLAIM_CONFLICT,
NFS4ERR_REP_TOO_BIG,
NFS4ERR_REP_TOO_BIG_TO_CACHE,
NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS,
NFS4ERR_SERVERFAULT, NFS4ERR_SHARE_DENIED,
NFS4ERR_STALE, NFS4ERR_SYMLINK,
NFS4ERR_TOO_MANY_OPS,
NFS4ERR_UNSAFE_COMPOUND, NFS4ERR_WRONGSEC,

OPEN_CONFIRM	NFS4ERR_WRONG_TYPE
	NFS4ERR_NOTSUPP
OPEN_DOWNGRADE	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR,
	NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_EXPIRED,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVALID,
	NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OLD_STATEID,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_CRED

OPENATTR	NFS4ERR_ACCESS, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_DQUOT, NFS4ERR_FHEXPIRED,
	NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOENT,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,
	NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNSAFE_COMPOUND,
	NFS4ERR_WRONG_TYPE
PUTFH	NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_MOVED, NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,

	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_WRONGSEC
PUTPUBFH	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC
PUTROOTFH	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC

READ	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,
	NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_DELEG_REVOKED, NFS4ERR_EXPIRED,
	NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,
	NFS4ERR_INVALID, NFS4ERR_ISDIR, NFS4ERR_IO,
	NFS4ERR_LOCKED, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_OLD_STATEID,
	NFS4ERR_OPENMODE,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_PNFS_IO_HOLE,

	NFS4ERR_PNFS_NO_LAYOUT,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_SYMLINK,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_TYPE
READDIR	NFS4ERR_ACCESS, NFS4ERR_BADXDR,
	NFS4ERR_BAD_COOKIE, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,
	NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR,
	NFS4ERR_NOT_SAME,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOOSMALL,
	NFS4ERR_TOO_MANY_OPS
READLINK	NFS4ERR_ACCESS, NFS4ERR_DEADSESSION,
	NFS4ERR_DELAY, NFS4ERR_FHEXPIRED,
	NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_WRONG_TYPE

RECLAIM_COMPLETE	NFS4ERR_BADXDR, NFS4ERR_COMPLETE_ALREADY,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVALID,
	NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,

	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT,
	NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_WRONG_CRED, NFS4ERR_WRONG_TYPE
RELEASE_LOCKOWNER	NFS4ERR_NOTSUPP
REMOVE	NFS4ERR_ACCESS, NFS4ERR_BADCHAR,
	NFS4ERR_BADNAME, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN,
	NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_IO,
	NFS4ERR_MOVED, NFS4ERR_NAME_TOO_LONG,
	NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE,
	NFS4ERR_NOTDIR, NFS4ERR_NOTEMPTY,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_TOO_MANY_OPS
RENAME	NFS4ERR_ACCESS, NFS4ERR_BADCHAR,
	NFS4ERR_BADNAME, NFS4ERR_BADXDR,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_DQUOT, NFS4ERR_EXIST,
	NFS4ERR_FHEXPIRED, NFS4ERR_FILE_OPEN,
	NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_IO,
	NFS4ERR_MOVED, NFS4ERR_NAME_TOO_LONG,
	NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE,
	NFS4ERR_NOSPC, NFS4ERR_NOTDIR,
	NFS4ERR_NOTEMPTY,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC,
	NFS4ERR_XDEV
RENEW	NFS4ERR_NOTSUPP

RESTOREFH	NFS4ERR_DEADSESSION, NFS4ERR_FHEXPIRED, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC
SAVEFH	NFS4ERR_DEADSESSION, NFS4ERR_FHEXPIRED, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
SECINFO	NFS4ERR_ACCESS, NFS4ERR_BADCHAR, NFS4ERR_BADNAME, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVALID, NFS4ERR_MOVED, NFS4ERR_NAME_TOO_LONG, NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
SECINFO_NO_NAME	NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVALID, NFS4ERR_MOVED, NFS4ERR_NOENT, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTDIR, NFS4ERR_NOTSUPP, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS

Internet-Draft

NFSv4.1

September 2008

SEQUENCE	NFS4ERR_BADSESSION, NFS4ERR_BADSLOT, NFS4ERR_BADXDR, NFS4ERR_BAD_HIGH_SLOT, NFS4ERR_CONN_NOT_BOUND_TO_SESSION, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_RETRY_UNCACHED_REP, NFS4ERR_SEQUENCE_POS, NFS4ERR_SEQ_FALSE_RETRY, NFS4ERR_SEQ_MISORDERED, NFS4ERR_TOO_MANY_OPS
SET_SSV	NFS4ERR_BADXDR, NFS4ERR_BAD_SESSION_DIGEST, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_INVAL, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_TOO_MANY_OPS
SETATTR	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR, NFS4ERR_BADOWNER, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DELEG_REVOKED, NFS4ERR_DQUOT, NFS4ERR_EXPIRED, NFS4ERR_FBIG, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_LOCKED, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC, NFS4ERR_OLD_STATEID, NFS4ERR_OPENMODE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_PERM, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS, NFS4ERR_UNKNOWN_LAYOUTTYPE, NFS4ERR_WRONG_TYPE
SETCLIENTID	NFS4ERR_NOTSUPP
SETCLIENTID_CONFIRM	NFS4ERR_NOTSUPP

TEST_STATEID	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS
--------------	---

VERIFY	NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP, NFS4ERR_BADCHAR, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOT_SAME, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS, NFS4ERR_UNKNOWN_LAYOUTTYPE, NFS4ERR_WRONG_TYPE
WANT_DELEGATION	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DELEG_ALREADY_WANTED, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOTSUPP, NFS4ERR_NO_GRACE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_RECALLCONFLICT, NFS4ERR_RECLAIM_BAD, NFS4ERR_RECLAIM_CONFLICT, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_TYPE
WRITE	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DELEG_REVOKED, NFS4ERR_DQUOT,

	NFS4ERR_EXPIRED, NFS4ERR_FBIG,
	NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,
	NFS4ERR_INVALID, NFS4ERR_IO, NFS4ERR_ISDIR,
	NFS4ERR_LOCKED, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC,
	NFS4ERR_OLD_STATEID, NFS4ERR_OPENMODE,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_PNFS_IO_HOLE,
	NFS4ERR_PNFS_NO_LAYOUT,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG, NFS4ERR_ROFS,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_SYMLINK, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_WRONG_TYPE

+-----+-----+

Table 6

[15.3.](#) Callback operations and their valid errors

This section contains a table which gives the valid error returns for each callback operation. The error code NFS4_OK (indicating no error) is not listed but should be understood to be returnable by all callback operations with the exception of CB_ILLEGAL.

Valid error returns for each protocol callback operation

Callback Operation	Errors
CB_GETATTR	NFS4ERR_BADHANDLE, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_INVALID, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS,
CB_ILLEGAL	NFS4ERR_BADXDR, NFS4ERR_OP_ILLEGAL
CB_LAYOUTRECALL	NFS4ERR_BADHANDLE, NFS4ERR_BADIOMODE,

CB_NOTIFY	NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID,
	NFS4ERR_DELAY, NFS4ERR_INVAL,
	NFS4ERR_NOMATCHING_LAYOUT,
	NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNKNOWN_LAYOUTTYPE,
	NFS4ERR_WRONG_TYPE
	NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,
	NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,
	NFS4ERR_INVAL, NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS

CB_NOTIFY_LOCK	NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,
	NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,
	NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS
	NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,
CB_PUSH_DELEG	NFS4ERR_DELAY, NFS4ERR_INVAL,
	NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REJECT_DELEG,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS,

CB_RECALL	NFS4ERR_WRONG_TYPE
	NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,
	NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,
CB_RECALL_ANY	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
CB_RECALLABLE_OBJ_AVAIL	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS
CB_RECALLABLE_OBJ_AVAIL	NFS4ERR_BADXDR, NFS4ERR_DELAY,
	NFS4ERR_INVAL, NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
CB_RECALLABLE_OBJ_AVAIL	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
CB_RECALLABLE_OBJ_AVAIL	NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS

CB_RECALL_SLOT	NFS4ERR_BADXDR, NFS4ERR_BAD_HIGH_SLOT,
	NFS4ERR_DELAY,
	NFS4ERR_OP_NOT_IN_SESSION,
CB_RECALL_SLOT	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
CB_SEQUENCE	NFS4ERR_TOO_MANY_OPS
	NFS4ERR_BADSESSION, NFS4ERR_BADSLOT,
	NFS4ERR_BADXDR, NFS4ERR_BAD_HIGH_SLOT,
CB_SEQUENCE	NFS4ERR_CONN_NOT_BOUND_TO_SESSION,
	NFS4ERR_DELAY, NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,

CB_WANTS_CANCELLED	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_RETRY_UNCACHED_REP,
	NFS4ERR_SEQUENCE_POS,
	NFS4ERR_SEQ_FALSE_RETRY,
	NFS4ERR_SEQ_MISORDERED,
	NFS4ERR_TOO_MANY_OPS
	NFS4ERR_BADXDR, NFS4ERR_DELAY,
	NFS4ERR_NOTSUPP,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_REP_TOO_BIG,
	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS

Table 7

[15.4.](#) Errors and the operations that use them

Error	Operations
NFS4ERR_ACCESS	ACCESS, COMMIT, CREATE, GETATTR, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, READ, REaddir, READLINK, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WRITE

NFS4ERR_ADMIN_REVOKED	CLOSE, DELEGRETURN,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, LOCK, LOCKU,
	OPEN, OPEN_DOWNGRADE, READ,
	SETATTR, WRITE

NFS4ERR_ATTRNOTSUPP	CREATE, LAYOUTCOMMIT, NVERIFY, OPEN, SETATTR, VERIFY
NFS4ERR_BACK_CHAN_BUSY	DESTROY_SESSION
NFS4ERR_BADCHAR	CREATE, EXCHANGE_ID, LINK, LOOKUP, NVERIFY, OPEN, REMOVE, RENAME, SECINFO, SETATTR, VERIFY
NFS4ERR_BADHANDLE	CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_LOCK, CB_PUSH_DELEG, CB_RECALL, PUTFH
NFS4ERR_BADIOMODE	CB_LAYOUTRECALL, LAYOUTCOMMIT, LAYOUTGET
NFS4ERR_BADLAYOUT	LAYOUTCOMMIT, LAYOUTGET
NFS4ERR_BADNAME	CREATE, LINK, LOOKUP, OPEN, REMOVE, RENAME, SECINFO
NFS4ERR_BADOWNER	CREATE, OPEN, SETATTR
NFS4ERR_BADSESSION	BIND_CONN_TO_SESSION, CB_SEQUENCE, DESTROY_SESSION, SEQUENCE
NFS4ERR_BADSLOT	CB_SEQUENCE, SEQUENCE
NFS4ERR_BADTYPE	CREATE

NFS4ERR_BADXDR	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_ILLEGAL, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_LOCK, CB_PUSH_DELEG, CB_RECALL, CB_RECALLABLE_OBJ_AVAIL, CB_RECALL_ANY, CB_RECALL_SLOT, CB_SEQUENCE, CB_WANTS_CANCELLED, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGPURGE, DELEGRETURN, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, FREE_STATEID, GETATTR, GETDEVICEINFO, GETDEVICELIST, GET_DIR_DELEGATION, ILLEGAL, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, RECLAIM_COMPLETE, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME, SEQUENCE, SETATTR, SET_SSV, TEST_STATEID, VERIFY, WANT_DELEGATION, WRITE
NFS4ERR_BAD_COOKIE	GETDEVICELIST, READDIR
NFS4ERR_BAD_HIGH_SLOT	CB_RECALL_SLOT, CB_SEQUENCE, SEQUENCE
NFS4ERR_BAD_RANGE	LOCK, LOCKT, LOCKU
NFS4ERR_BAD_SESSION_DIGEST	BIND_CONN_TO_SESSION, SET_SSV
NFS4ERR_BAD_STATEID	CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_LOCK, CB_RECALL, CLOSE, DELEGRETURN, FREE_STATEID, LAYOUTGET, LAYOUTRETURN, LOCK, LOCKU, OPEN, OPEN_DOWNGRADE, READ, SETATTR, WRITE
NFS4ERR_CB_PATH_DOWN	DESTROY_SESSION
NFS4ERR_CLID_INUSE	EXCHANGE_ID
NFS4ERR_CLIENTID_BUSY	DESTROY_CLIENTID
NFS4ERR_COMPLETE_ALREADY	RECLAIM_COMPLETE
NFS4ERR_CONN_NOT_BOUND_TO_SESSION	CB_SEQUENCE, DESTROY_SESSION, SEQUENCE
NFS4ERR_DEADLOCK	LOCK

Internet-Draft

NFSv4.1

September 2008

NFS4ERR_DEADSESSION

ACCESS, BACKCHANNEL_CTL,
BIND_CONN_TO_SESSION, CLOSE,
COMMIT, CREATE,
CREATE_SESSION, DELEGPURGE,
DELEGRETURN,
DESTROY_CLIENTID,
DESTROY_SESSION, EXCHANGE_ID,
FREE_STATEID, GETATTR,
GETDEVICEINFO, GETDEVICELIST,
GET_DIR_DELEGATION,
LAYOUTCOMMIT, LAYOUTGET,
LAYOUTRETURN, LINK, LOCK,
LOCKT, LOCKU, LOOKUP,
LOOKUPP, NVERIFY, OPEN,
OPENATTR, OPEN_DOWNGRADE,
PUTFH, PUTPUBFH, PUTROOTFH,
READ, READDIR, READLINK,
RECLAIM_COMPLETE, REMOVE,
RENAME, RESTOREFH, SAVEFH,
SECINFO, SECINFO_NO_NAME,
SEQUENCE, SETATTR, SET_SSV,
TEST_STATEID, VERIFY,
WANT_DELEGATION, WRITE

Internet-Draft

NFSv4.1

September 2008

NFS4ERR_DELAY	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_LOCK, CB_PUSH_DELEG, CB_RECALL, CB_RECALLABLE_OBJ_AVAIL, CB_RECALL_ANY, CB_RECALL_SLOT, CB_SEQUENCE, CB_WANTS_CANCELLED, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGPURGE, DELEGRETURN, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, FREE_STATEID, GETATTR, GETDEVICEINFO, GETDEVICELIST, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RECLAIM_COMPLETE, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME, SEQUENCE, SETATTR, SET_SSV, TEST_STATEID, VERIFY, WANT_DELEGATION, WRITE
NFS4ERR_DELEG_ALREADY_WANTED	OPEN, WANT_DELEGATION
NFS4ERR_DELEG_REVOKED	DELEGRETURN, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, OPEN, READ, SETATTR, WRITE
NFS4ERR_DENIED	LOCK, LOCKT

NFS4ERR_DIRDELEG_UNAVAIL	GET_DIR_DELEGATION
NFS4ERR_DQUOT	CREATE, LAYOUTGET, LINK,
	OPEN, OPENATTR, RENAME,
	SETATTR, WRITE
NFS4ERR_ENCR_ALG_UNSUPP	EXCHANGE_ID
NFS4ERR_EXIST	CREATE, LINK, OPEN, RENAME
NFS4ERR_EXPIRED	CLOSE, DELEGRETURN,
	LAYOUTCOMMIT, LAYOUTRETURN,
	LOCK, LOCKU, OPEN,
	OPEN_DOWNGRADE, READ,
	SETATTR, WRITE
NFS4ERR_FBIG	LAYOUTCOMMIT, OPEN, SETATTR,
	WRITE

NFS4ERR_FHEXPIRED	ACCESS, CLOSE, COMMIT,
	CREATE, DELEGRETURN, GETATTR,
	GETDEVICELIST, GETFH,
	GET_DIR_DELEGATION,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, LINK, LOCK,
	LOCKT, LOCKU, LOOKUP,
	LOOKUPP, NVERIFY, OPEN,
	OPENATTR, OPEN_DOWNGRADE,
	READ, READDIR, READLINK,
	RECLAIM_COMPLETE, REMOVE,
	RENAME, RESTOREFH, SAVEFH,
	SECINFO, SECINFO_NO_NAME,
	SETATTR, VERIFY,
	WANT_DELEGATION, WRITE
NFS4ERR_FILE_OPEN	LINK, REMOVE, RENAME
NFS4ERR_GRACE	GETATTR, GET_DIR_DELEGATION,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, LINK, LOCK,
	LOCKT, NVERIFY, OPEN, READ,
	REMOVE, RENAME, SETATTR,
	VERIFY, WANT_DELEGATION,
	WRITE
NFS4ERR_HASH_ALG_UNSUPP	EXCHANGE_ID
NFS4ERR_INVAL	ACCESS, BACKCHANNEL_CTL,
	BIND_CONN_TO_SESSION,
	CB_GETATTR, CB_LAYOUTRECALL,
	CB_NOTIFY, CB_PUSH_DELEG,

	CB_RECALLABLE_OBJ_AVAIL,
	CB_RECALL_ANY, CREATE,
	CREATE_SESSION, DELEGRETURN,
	EXCHANGE_ID, GETATTR,
	GETDEVICEINFO, GETDEVICELIST,
	GET_DIR_DELEGATION,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, LINK, LOCK,
	LOCKT, LOCKU, LOOKUP,
	NVERIFY, OPEN,
	OPEN_DOWNGRADE, READ,
	READDIR, READLINK,
	RECLAIM_COMPLETE, REMOVE,
	RENAME, SECINFO,
	SECINFO_NO_NAME, SETATTR,
	SET_SSV, VERIFY,
	WANT_DELEGATION, WRITE

NFS4ERR_IO	ACCESS, COMMIT, CREATE,
	GETATTR, GETDEVICELIST,
	GET_DIR_DELEGATION,
	LAYOUTCOMMIT, LAYOUTGET,
	LINK, LOOKUP, LOOKUPP,
	NVERIFY, OPEN, OPENATTR,
	READ, READDIR, READLINK,
	REMOVE, RENAME, SETATTR,
	VERIFY, WANT_DELEGATION,
	WRITE
NFS4ERR_ISDIR	COMMIT, LAYOUTCOMMIT,
	LAYOUTRETURN, LINK, LOCK,
	LOCKT, OPEN, READ, WRITE
NFS4ERR_LAYOUTTRYLATER	LAYOUTGET
NFS4ERR_LAYOUTUNAVAILABLE	LAYOUTGET
NFS4ERR_LOCKED	LAYOUTGET, READ, SETATTR,
	WRITE
NFS4ERR_LOCKS_HELD	CLOSE, FREE_STATEID
NFS4ERR_LOCK_NOTSUPP	LOCK
NFS4ERR_LOCK_RANGE	LOCK, LOCKT, LOCKU
NFS4ERR_MLINK	CREATE, LINK

NFS4ERR_MOVED	ACCESS, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, GETFH, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, RECLAIM_COMPLETE, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WANT_DELEGATION, WRITE
NFS4ERR_NAMETOOLONG	CREATE, LINK, LOOKUP, OPEN, REMOVE, RENAME, SECINFO
NFS4ERR_NOENT	BACKCHANNEL_CTL, CREATE_SESSION, EXCHANGE_ID, GETDEVICEINFO, LOOKUP, LOOKUPP, OPEN, OPENATTR, REMOVE, RENAME, SECINFO, SECINFO_NO_NAME

NFS4ERR_NOFILEHANDLE	ACCESS, CLOSE, COMMIT, CREATE, DELEGRETURN, GETATTR, GETDEVICELIST, GETFH, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, READ, READDIR, READLINK, RECLAIM_COMPLETE, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY,
----------------------	---

NFS4ERR_NOMATCHING_LAYOUT	WANT_DELEGATION, WRITE
NFS4ERR_NOSPC	CB_LAYOUTRECALL
	CREATE, CREATE_SESSION,
	LAYOUTGET, LINK, OPEN,
	OPENATTR, RENAME, SETATTR,
	WRITE
NFS4ERR_NOTDIR	CREATE, GET_DIR_DELEGATION,
	LINK, LOOKUP, LOOKUPP, OPEN,
	READDIR, REMOVE, RENAME,
	SECINFO, SECINFO_NO_NAME
NFS4ERR_NOTEMPTY	REMOVE, RENAME
NFS4ERR_NOTSUPP	CB_LAYOUTRECALL, CB_NOTIFY,
	CB_NOTIFY_LOCK,
	CB_PUSH_DELEG,
	CB_RECALLABLE_OBJ_AVAIL,
	CB_WANTS_CANCELLED,
	DELEGPURGE, DELEGRETURN,
	GETDEVICEINFO, GETDEVICELIST,
	GET_DIR_DELEGATION,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, LINK, OPENATTR,
	OPEN_CONFIRM,
	RELEASE_LOCKOWNER, RENEW,
	SECINFO_NO_NAME, SETCLIENTID,
	SETCLIENTID_CONFIRM,
	WANT_DELEGATION
NFS4ERR_NOT_ONLY_OP	BIND_CONN_TO_SESSION,
	CREATE_SESSION,
	DESTROY_CLIENTID,
	DESTROY_SESSION, EXCHANGE_ID
NFS4ERR_NOT_SAME	EXCHANGE_ID, GETDEVICELIST,
	READDIR, VERIFY
NFS4ERR_NO_GRACE	LAYOUTCOMMIT, LAYOUTRETURN,
	LOCK, OPEN, WANT_DELEGATION

NFS4ERR_OLD_STATEID	CLOSE, DELEGRETURN,
	FREE_STATEID, LAYOUTGET,
	LAYOUTRETURN, LOCK, LOCKU,
	OPEN, OPEN_DOWNGRADE, READ,
	SETATTR, WRITE
NFS4ERR_OPENMODE	LAYOUTGET, LOCK, READ,
	SETATTR, WRITE

NFS4ERR_OP_ILLEGAL	CB_ILLEGAL, ILLEGAL
NFS4ERR_OP_NOT_IN_SESSION	ACCESS, BACKCHANNEL_CTL,
	CB_GETATTR, CB_LAYOUTRECALL,
	CB_NOTIFY, CB_NOTIFY_LOCK,
	CB_PUSH_DELEG, CB_RECALL,
	CB_RECALLABLE_OBJ_AVAIL,
	CB_RECALL_ANY,
	CB_RECALL_SLOT,
	CB_WANTS_CANCELLED, CLOSE,
	COMMIT, CREATE, DELEGPURGE,
	DELEGRETURN, FREE_STATEID,
	GETATTR, GETDEVICEINFO,
	GETDEVICELIST, GETFH,
	GET_DIR_DELEGATION,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, LINK, LOCK,
	LOCKT, LOCKU, LOOKUP,
	LOOKUPP, NVERIFY, OPEN,
	OPENATTR, OPEN_DOWNGRADE,
	PUTFH, PUTPUBFH, PUTROOTFH,
	READ, READDIR, READLINK,
	RECLAIM_COMPLETE, REMOVE,
	RENAME, RESTOREFH, SAVEFH,
	SECINFO, SECINFO_NO_NAME,
	SETATTR, SET_SSV,
	TEST_STATEID, VERIFY,
	WANT_DELEGATION, WRITE
NFS4ERR_PERM	CREATE, OPEN, SETATTR
NFS4ERR_PNFS_IO_HOLE	READ, WRITE
NFS4ERR_PNFS_NO_LAYOUT	READ, WRITE
NFS4ERR_RECALLCONFLICT	LAYOUTGET, WANT_DELEGATION
NFS4ERR_RECLAIM_BAD	LAYOUTCOMMIT, LOCK, OPEN,
	WANT_DELEGATION
NFS4ERR_RECLAIM_CONFLICT	LAYOUTCOMMIT, LOCK, OPEN,
	WANT_DELEGATION
NFS4ERR_REJECT_DELEG	CB_PUSH_DELEG

NFS4ERR_REP_TOO_BIG

ACCESS, BACKCHANNEL_CTL,
BIND_CONN_TO_SESSION,
CB_GETATTR, CB_LAYOUTRECALL,
CB_NOTIFY, CB_NOTIFY_LOCK,
CB_PUSH_DELEG, CB_RECALL,
CB_RECALLABLE_OBJ_AVAIL,
CB_RECALL_ANY,
CB_RECALL_SLOT, CB_SEQUENCE,
CB_WANTS_CANCELLED, CLOSE,
COMMIT, CREATE,
CREATE_SESSION, DELEGPURGE,
DELEGRETURN,
DESTROY_CLIENTID,
DESTROY_SESSION, EXCHANGE_ID,
FREE_STATEID, GETATTR,
GETDEVICEINFO, GETDEVICELIST,
GET_DIR_DELEGATION,
LAYOUTCOMMIT, LAYOUTGET,
LAYOUTRETURN, LINK, LOCK,
LOCKT, LOCKU, LOOKUP,
LOOKUPP, NVERIFY, OPEN,
OPENATTR, OPEN_DOWNGRADE,
PUTFH, PUTPUBFH, PUTROOTFH,
READ, READDIR, READLINK,
RECLAIM_COMPLETE, REMOVE,
RENAME, RESTOREFH, SAVEFH,
SECINFO, SECINFO_NO_NAME,
SEQUENCE, SETATTR, SET_SSV,
TEST_STATEID, VERIFY,
WANT_DELEGATION, WRITE

Internet-Draft

NFSv4.1

September 2008

NFS4ERR_REP_TOO_BIG_TO_CACHE	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_LOCK, CB_PUSH_DELEG, CB_RECALL, CB_RECALLABLE_OBJ_AVAIL, CB_RECALL_ANY, CB_RECALL_SLOT, CB_SEQUENCE, CB_WANTS_CANCELLED, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGPURGE, DELEGRETURN, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, FREE_STATEID, GETATTR, GETDEVICEINFO, GETDEVICELIST, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RECLAIM_COMPLETE, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SEQUENCE, SETATTR, SET_SSV, TEST_STATEID, VERIFY, WANT_DELEGATION, WRITE
------------------------------	---

Internet-Draft

NFSv4.1

September 2008

NFS4ERR_REQ_TOO_BIG	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_LOCK, CB_PUSH_DELEG, CB_RECALL, CB_RECALLABLE_OBJ_AVAIL, CB_RECALL_ANY, CB_RECALL_SLOT, CB_SEQUENCE, CB_WANTS_CANCELLED, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGPURGE, DELEGRETURN, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, FREE_STATEID, GETATTR, GETDEVICEINFO, GETDEVICELIST, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RECLAIM_COMPLETE, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SEQUENCE, SETATTR, SET_SSV, TEST_STATEID, VERIFY, WANT_DELEGATION, WRITE
NFS4ERR_RETRY_UNCACHED_REP	CB_SEQUENCE, SEQUENCE
NFS4ERR_ROFS	CREATE, LINK, LOCK, LOCKT, OPEN, OPENATTR, OPEN_DOWNGRADE, REMOVE, RENAME, SETATTR, WRITE
NFS4ERR_SAME	NVERIFY

NFS4ERR_SEQUENCE_POS	CB_SEQUENCE, SEQUENCE
NFS4ERR_SEQ_FALSE_RETRY	CB_SEQUENCE, SEQUENCE
NFS4ERR_SEQ_MISORDERED	CB_SEQUENCE, CREATE_SESSION, SEQUENCE

NFS4ERR_SERVERFAULT	ACCESS, BIND_CONN_TO_SESSION, CB_GETATTR, CB_NOTIFY, CB_NOTIFY_LOCK, CB_PUSH_DELEG, CB_RECALL, CB_RECALLABLE_OBJ_AVAIL, CB_WANTS_CANCELLED, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGPURGE, DELEGRETURN, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, FREE_STATEID, GETATTR, GETDEVICEINFO, GETDEVICELIST, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, PUTPUBFH, PUTROOTFH, READ, READDIR, READLINK, RECLAIM_COMPLETE, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, TEST_STATEID, VERIFY, WANT_DELEGATION, WRITE
NFS4ERR_SHARE_DENIED	OPEN
NFS4ERR_STALE	ACCESS, CLOSE, COMMIT,

	CREATE, DELEGRETURN, GETATTR, GETFH, GET_DIR_DELEGATION, LAYOUTCOMMIT, LAYOUTGET, LAYOUTRETURN, LINK, LOCK, LOCKT, LOCKU, LOOKUP, LOOKUPP, NVERIFY, OPEN, OPENATTR, OPEN_DOWNGRADE, PUTFH, READ, READDIR, READLINK, RECLAIM_COMPLETE, REMOVE, RENAME, RESTOREFH, SAVEFH, SECINFO, SECINFO_NO_NAME, SETATTR, VERIFY, WANT_DELEGATION, WRITE
NFS4ERR_STALE_CLIENTID	CREATE_SESSION, DESTROY_CLIENTID, DESTROY_SESSION

NFS4ERR_SYMLINK	COMMIT, LAYOUTCOMMIT, LINK, LOCK, LOCKT, LOOKUP, LOOKUPP, OPEN, READ, WRITE
NFS4ERR_TOOSMALL	CREATE_SESSION, GETDEVICEINFO, LAYOUTGET, READDIR
NFS4ERR_TOO_MANY_OPS	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_LOCK, CB_PUSH_DELEG, CB_RECALL, CB_RECALLABLE_OBJ_AVAIL, CB_RECALL_ANY, CB_RECALL_SLOT, CB_SEQUENCE, CB_WANTS_CANCELLED, CLOSE, COMMIT, CREATE, CREATE_SESSION, DELEGPURGE, DELEGRETURN, DESTROY_CLIENTID, DESTROY_SESSION, EXCHANGE_ID, FREE_STATEID, GETATTR, GETDEVICEINFO, GETDEVICELIST,

	GET_DIR_DELEGATION,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, LINK, LOCK,
	LOCKT, LOCKU, LOOKUP,
	LOOKUPP, NVERIFY, OPEN,
	OPENATTR, OPEN_DOWNGRADE,
	PUTFH, PUTPUBFH, PUTROOTFH,
	READ, READDIR, READLINK,
	RECLAIM_COMPLETE, REMOVE,
	RENAME, RESTOREFH, SAVEFH,
	SECINFO, SECINFO_NO_NAME,
	SEQUENCE, SETATTR, SET_SSV,
	TEST_STATEID, VERIFY,
	WANT_DELEGATION, WRITE
NFS4ERR_UNKNOWN_LAYOUTTYPE	CB_LAYOUTRECALL,
	GETDEVICEINFO, GETDEVICELIST,
	LAYOUTCOMMIT, LAYOUTGET,
	LAYOUTRETURN, NVERIFY,
	SETATTR, VERIFY
NFS4ERR_UNSAFE_COMPOUND	CREATE, OPEN, OPENATTR
NFS4ERR_WRONGSEC	LINK, LOOKUP, LOOKUPP, OPEN,
	PUTFH, PUTPUBFH, PUTROOTFH,
	RENAME, RESTOREFH

NFS4ERR_WRONG_CRED	CLOSE, CREATE_SESSION,
	DELEGPURGE, DELEGRETURN,
	DESTROY_CLIENTID,
	DESTROY_SESSION,
	FREE_STATEID, LAYOUTCOMMIT,
	LAYOUTRETURN, LOCK, LOCKT,
	LOCKU, OPEN_DOWNGRADE,
	RECLAIM_COMPLETE
NFS4ERR_WRONG_TYPE	CB_LAYOUTRECALL,
	CB_PUSH_DELEG, COMMIT,
	GETATTR, LAYOUTGET,
	LAYOUTRETURN, LINK, LOCK,
	LOCKT, NVERIFY, OPEN,
	OPENATTR, READ, READLINK,
	RECLAIM_COMPLETE, SETATTR,

		VERIFY, WANT_DELEGATION,	
		WRITE	
NFS4ERR_XDEV		LINK, RENAME	
+-----+-----+			

Table 8

[16.](#) NFSv4.1 Procedures

Both procedures, NULL and COMPOUND, MUST be implemented.

[16.1.](#) Procedure 0: NULL – No Operation

[16.1.1.](#) ARGUMENTS

void;

[16.1.2.](#) RESULTS

void;

[16.1.3.](#) DESCRIPTION

This is the standard NULL procedure with the standard void argument and 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.

[16.1.4.](#) ERRORS

None.

[16.2.](#) Procedure 1: COMPOUND – Compound Operations

[16.2.1.](#) ARGUMENTS

```

enum nfs_opnum4 {
    OP_ACCESS          = 3,
    OP_CLOSE           = 4,
    OP_COMMIT          = 5,
    OP_CREATE          = 6,
    OP_DELEGPURGE      = 7,
    OP_DELEGRETURN     = 8,
    OP_GETATTR         = 9,
    OP_GETFH           = 10,
    OP_LINK            = 11,
    OP_LOCK            = 12,
    OP_LOCKT           = 13,
    OP_LOCKU           = 14,
    OP_LOOKUP          = 15,
    OP_LOOKUPP         = 16,
    OP_NVERIFY         = 17,
    OP_OPEN            = 18,
    OP_OPENATTR        = 19,
    OP_OPEN_CONFIRM    = 20, /* Mandatory not-to-implement */
    OP_OPEN_DOWNGRADE = 21,
    OP_PUTFH           = 22,
    OP_PUTPUBFH        = 23,
    OP_PUTROOTFH       = 24,
    OP_READ            = 25,
    OP_READDIR         = 26,
    OP_READLINK        = 27,
    OP_REMOVE          = 28,
    OP_RENAME          = 29,
    OP_RENEW           = 30, /* Mandatory not-to-implement */
    OP_RESTOREFH       = 31,
    OP_SAVEFH          = 32,
    OP_SECINFO         = 33,
    OP_SETATTR         = 34,
    OP_SETCLIENTID     = 35, /* Mandatory not-to-implement */
    OP_SETCLIENTID_CONFIRM = 36, /* Mandatory not-to-implement */
    OP_VERIFY          = 37,
    OP_WRITE           = 38,
    OP_RELEASE_LOCKOWNER = 39, /* Mandatory not-to-implement */

    /* new operations for NFSv4.1 */

```

```

    OP_BACKCHANNEL_CTL = 40,

```

```

OP_BIND_CONN_TO_SESSION = 41,
OP_EXCHANGE_ID           = 42,
OP_CREATE_SESSION        = 43,
OP_DESTROY_SESSION       = 44,
OP_FREE_STATEID          = 45,
OP_GET_DIR_DELEGATION    = 46,
OP_GETDEVICEINFO         = 47,
OP_GETDEVICELIST         = 48,
OP_LAYOUTCOMMIT          = 49,
OP_LAYOUTGET             = 50,
OP_LAYOUTRETURN          = 51,
OP_SECINFO_NO_NAME       = 52,
OP_SEQUENCE               = 53,
OP_SET_SSV                = 54,
OP_TEST_STATEID          = 55,
OP_WANT_DELEGATION       = 56,
OP_DESTROY_CLIENTID      = 57,
OP_RECLAIM_COMPLETE      = 58,
OP_ILLEGAL                = 10044
};

```

```

union nfs_argop4 switch (nfs_opnum4 argop) {
    case OP_ACCESS:      ACCESS4args opaccess;
    case OP_CLOSE:       CLOSE4args opclose;
    case OP_COMMIT:      COMMIT4args opcommit;
    case OP_CREATE:      CREATE4args opcreate;
    case OP_DELEGPURGE:  DELEGPURGE4args opdelegpurge;
    case OP_DELEGRETURN: DELEGRETURN4args opdelegreturn;
    case OP_GETATTR:     GETATTR4args opgetattr;
    case OP_GETFH:       void;
    case OP_LINK:        LINK4args oplink;
    case OP_LOCK:        LOCK4args oplock;
    case OP_LOCKT:       LOCKT4args oplockt;
    case OP_LOCKU:       LOCKU4args oplocku;
    case OP_LOOKUP:      LOOKUP4args oplookup;
    case OP_LOOKUPP:     void;
    case OP_NVERIFY:     NVERIFY4args opnverify;
    case OP_OPEN:        OPEN4args opopen;
    case OP_OPENATTR:    OPENATTR4args opopenattr;

    /* Not for NFSv4.1 */
    case OP_OPEN_CONFIRM: OPEN_CONFIRM4args opopen_confirm;

    case OP_OPEN_DOWNGRADE:
        OPEN_DOWNGRADE4args opopen_downgrade;

```

```
case OP_PUTFH:          PUTFH4args opputfh;
case OP_PUTPUBFH:       void;
case OP_PUTROOTFH:      void;
case OP_READ:           READ4args opread;
case OP_READDIR:        READDIR4args opreaddir;
case OP_READLINK:       void;
case OP_REMOVE:         REMOVE4args opremove;
case OP_RENAME:         RENAME4args oprename;

/* Not for NFSv4.1 */
case OP_RENEW:          RENEW4args oprenew;

case OP_RESTOREFH:      void;
case OP_SAVEFH:         void;
case OP_SECINFO:        SECINFO4args opsecinfo;
case OP_SETATTR:        SETATTR4args opsetattr;

/* Not for NFSv4.1 */
case OP_SETCLIENTID:    SETCLIENTID4args opsetclientid;

/* Not for NFSv4.1 */
case OP_SETCLIENTID_CONFIRM: SETCLIENTID_CONFIRM4args
                                opsetclientid_confirm;
case OP_VERIFY:         VERIFY4args opverify;
case OP_WRITE:          WRITE4args opwrite;

/* Not for NFSv4.1 */
case OP_RELEASE_LOCKOWNER:
                                RELEASE_LOCKOWNER4args
                                oprelease_lockowner;

/* Operations new to NFSv4.1 */
case OP_BACKCHANNEL_CTL:
                                BACKCHANNEL_CTL4args opbackchannel_ctl;

case OP_BIND_CONN_TO_SESSION:
                                BIND_CONN_TO_SESSION4args
                                opbind_conn_to_session;

case OP_EXCHANGE_ID:    EXCHANGE_ID4args opexchange_id;

case OP_CREATE_SESSION:
                                CREATE_SESSION4args opcreate_session;

case OP_DESTROY_SESSION:
                                DESTROY_SESSION4args opdestroy_session;
```

```
case OP_FREE_STATEID:  FREE_STATEID4args opfree_stateid;
```

```
case OP_GET_DIR_DELEGATION:
    GET_DIR_DELEGATION4args
    opget_dir_delegation;

case OP_GETDEVICEINFO: GETDEVICEINFO4args opgetdeviceinfo;
case OP_GETDEVICELIST: GETDEVICELIST4args opgetdevicelist;
case OP_LAYOUTCOMMIT:  LAYOUTCOMMIT4args oplayoutcommit;
case OP_LAYOUTGET:     LAYOUTGET4args oplayoutget;
case OP_LAYOUTRETURN:  LAYOUTRETURN4args oplayoutreturn;

case OP_SECINFO_NO_NAME:
    SECINFO_NO_NAME4args opsecinfo_no_name;

case OP_SEQUENCE:      SEQUENCE4args opsequence;
case OP_SET_SSV:       SET_SSV4args opset_ssv;
case OP_TEST_STATEID:  TEST_STATEID4args optest_stateid;

case OP_WANT_DELEGATION:
    WANT_DELEGATION4args opwant_delegation;

case OP_DESTROY_CLIENTID:
    DESTROY_CLIENTID4args
    opdestroy_clientid;

case OP_RECLAIM_COMPLETE:
    RECLAIM_COMPLETE4args
    opreclaim_complete;

/* Operations not new to NFSv4.1 */
case OP_ILLEGAL:      void;
};

struct COMPOUND4args {
    utf8str_cs      tag;
    uint32_t        minorversion;
    nfs_argop4      argarray<>;
};
```


[16.2.2.](#) RESULTS

```
union nfs_resop4 switch (nfs_opnum4 resop) {
  case OP_ACCESS:      ACCESS4res opaccess;
  case OP_CLOSE:       CLOSE4res opclose;
  case OP_COMMIT:      COMMIT4res opcommit;
  case OP_CREATE:      CREATE4res opcreate;
  case OP_DELEGPURGE:  DELEGPURGE4res opdelegpurge;
  case OP_DELEGRETURN: DELEGRETURN4res opdelegreturn;
```

```
  case OP_GETATTR:      GETATTR4res opgetattr;
  case OP_GETFH:        GETFH4res opgetfh;
  case OP_LINK:         LINK4res oplink;
  case OP_LOCK:         LOCK4res oplock;
  case OP_LOCKT:        LOCKT4res oplockt;
  case OP_LOCKU:        LOCKU4res oplocku;
  case OP_LOOKUP:       LOOKUP4res oplookup;
  case OP_LOOKUPP:      LOOKUPP4res oplookupp;
  case OP_NVERIFY:      NVERIFY4res opnverify;
  case OP_OPEN:         OPEN4res opopen;
  case OP_OPENATTR:     OPENATTR4res opopenattr;
  /* Not for NFSv4.1 */
  case OP_OPEN_CONFIRM: OPEN_CONFIRM4res opopen_confirm;

  case OP_OPEN_DOWNGRADE:
                        OPEN_DOWNGRADE4res
                        opopen_downgrade;

  case OP_PUTFH:        PUTFH4res opputfh;
  case OP_PUTPUBFH:     PUTPUBFH4res opputpubfh;
  case OP_PUTROOTFH:    PUTROOTFH4res opputrootfh;
  case OP_READ:         READ4res opread;
  case OP_READDIR:      READDIR4res opreaddir;
  case OP_READLINK:     READLINK4res opreadlink;
  case OP_REMOVE:       REMOVE4res opremove;
  case OP_RENAME:       RENAME4res oprename;
  /* Not for NFSv4.1 */
  case OP_RENEW:        RENEW4res oprenew;
  case OP_RESTOREFH:    RESTOREFH4res oprestorefh;
  case OP_SAVEFH:       SAVEFH4res opsavefh;
  case OP_SECINFO:      SECINFO4res opsecinfo;
  case OP_SETATTR:      SETATTR4res opsetattr;
```

```

/* Not for NFSv4.1 */
case OP_SETCLIENTID: SETCLIENTID4res opsetclientid;

/* Not for NFSv4.1 */
case OP_SETCLIENTID_CONFIRM:
    SETCLIENTID_CONFIRM4res
    opsetclientid_confirm;
case OP_VERIFY:      VERIFY4res opverify;
case OP_WRITE:       WRITE4res opwrite;

/* Not for NFSv4.1 */
case OP_RELEASE_LOCKOWNER:
    RELEASE_LOCKOWNER4res
    oprelease_lockowner;

/* Operations new to NFSv4.1 */

```

```

case OP_BACKCHANNEL_CTL:
    BACKCHANNEL_CTL4res
    opbackchannel_ctl;

case OP_BIND_CONN_TO_SESSION:
    BIND_CONN_TO_SESSION4res
    opbind_conn_to_session;

case OP_EXCHANGE_ID:  EXCHANGE_ID4res opexchange_id;

case OP_CREATE_SESSION:
    CREATE_SESSION4res
    opcreate_session;

case OP_DESTROY_SESSION:
    DESTROY_SESSION4res
    opdestroy_session;

case OP_FREE_STATEID:  FREE_STATEID4res
    opfree_stateid;

case OP_GET_DIR_DELEGATION:
    GET_DIR_DELEGATION4res
    opget_dir_delegation;

```

```

case OP_GETDEVICEINFO: GETDEVICEINFO4res
                        opgetdeviceinfo;

case OP_GETDEVICELIST: GETDEVICELIST4res
                        opgetdevicelist;

case OP_LAYOUTCOMMIT:  LAYOUTCOMMIT4res oplayoutcommit;
case OP_LAYOUTGET:     LAYOUTGET4res oplayoutget;
case OP_LAYOUTRETURN:  LAYOUTRETURN4res oplayoutreturn;

case OP_SECINFO_NO_NAME:
                        SECINFO_NO_NAME4res
                        opsecinfo_no_name;

case OP_SEQUENCE:      SEQUENCE4res opsequence;
case OP_SET_SSV:       SET_SSV4res opset_ssv;
case OP_TEST_STATEID:  TEST_STATEID4res optest_stateid;

case OP_WANT_DELEGATION:
                        WANT_DELEGATION4res
                        opwant_delegation;

case OP_DESTROY_CLIENTID:

```

```

                        DESTROY_CLIENTID4res
                        opdestroy_clientid;

case OP_RECLAIM_COMPLETE:
                        RECLAIM_COMPLETE4res
                        opreclaim_complete;

/* Operations not new to NFSv4.1 */
case OP_ILLEGAL:      ILLEGAL4res opillegal;
};

struct COMPOUND4res {
    nfsstat4          status;
    utf8str_cs        tag;
    nfs_resop4        resarray<>;
};

```

16.2.3. DESCRIPTION

The COMPOUND procedure is used to combine one or more of the NFS operations into a single RPC request. The 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 2.10.6.4](#) 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 and is 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.

[16.2.3.1](#). Current Filehandle 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 filehandle while the second two relate to the current stateid.

[16.2.3.1.1](#). Current Filehandle

The current and saved filehandle are used throughout the protocol. Most operations implicitly use the current filehandle as a argument

and many set the current filehandle as part of the results. The combination of client specified sequences of operations and current and saved filehandle arguments and results allows for greater protocol flexibility. The best or easiest example of current filehandle usage is a sequence like the following:

PUTFH fh1	{fh1}
LOOKUP "compA"	{fh2}
GETATTR	{fh2}
LOOKUP "compB"	{fh3}
GETATTR	{fh3}
LOOKUP "compC"	{fh4}
GETATTR	{fh4}
GETFH	

Figure 2

In this example, the PUTFH ([Section 18.19](#)) operation explicitly sets the current filehandle value while the result of each LOOKUP operation sets the current filehandle value to the resultant file system object. Also, the client is able to insert GETATTR operations using the current filehandle as an argument.

The PUTROOTFH ([Section 18.21](#)) and PUTPUBFH ([Section 18.21](#)) operations also set the current filehandle. The above example would replace "PUTFH fh1" with PUTROOTFH or PUTPUBFH with no filehandle argument in order to achieve the same effect (on the assumption that "compA" is directly below the root of the namespace).

Along with the current filehandle, there is a saved filehandle. While the current filehandle is set as the result of operations like LOOKUP, the saved filehandle must be set directly with the use of the SAVEFH operation. The SAVEFH operations copies the current filehandle value to the saved value. The saved filehandle value is used in combination with the current filehandle value for the LINK and RENAME operations. The RESTOREFH operation will copy the saved filehandle value to the current filehandle value; as a result, the saved filehandle value may be used a sort of "scratch" area for the client's series of operations.

Internet-Draft

NFSv4.1

September 2008

Figure 4

This next example is similar to the second in how it passes the stateid sid2 generated by the LOCK operation to the next 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. The example also shows how SAVEFH and RESTOREFH can save and later re-use a filehandle and stateid, passing them as the current filehandle and stateid to a READ operation.

PUTFH fh1	- -> {fh1, (0, 0)}
LOCK 0, 1024, (sid1)	{fh1, (sid1)} -> {fh1, (sid2)}
READ (1, 0), 0, 1024	{fh1, (sid2)} -> {fh1, (sid2)}
LOCKU 0, 1024, (1, 0)	{fh1, (sid2)} -> {fh1, (sid3)}
SAVEFH	{fh1, (sid3)} -> {fh1, (sid3)}
PUTFH fh2	{fh1, (sid3)} -> {fh2, (0, 0)}
WRITE (1, 0), 0, 1024	{fh2, (0, 0)} -> {fh2, (0, 0)}
RESTOREFH	{fh2, (0, 0)} -> {fh1, (sid3)}
READ (1, 0), 1024, 1024	{fh1, (sid3)} -> {fh1, (sid3)}

Figure 5

The final example shows a disallowed use of the current stateid. The client is attempting to implicitly pass anonymous special stateid, (0,0) to the READ operation. The server MUST return NFS4ERR_BAD_STATEID in the reply to the READ operation.

PUTFH fh1	- -> {fh1, (0, 0)}
READ (1, 0), 0, 1024	{fh1, (0, 0)} -> NFS4ERR_BAD_STATEID

Figure 6

[16.2.4.](#) ERRORS

COMPOUND will of course return every error that each operation on the fore channel can return (see Table 6). However if COMPOUND returns zero operations, obviously the error returned by COMPOUND has nothing

to do with an error returned by an operation. The list of errors COMPOUND will return if it processes zero operations include:

COMPOUND error returns

Error	Notes
NFS4ERR_BADCHAR	The tag argument has a character the replier does not support.
NFS4ERR_BADXDR	
NFS4ERR_DELAY	The tag argument is not in UTF-8 encoding.
NFS4ERR_INVALID	
NFS4ERR_MINOR_VERS_MISMATCH	
NFS4ERR_SERVERFAULT	
NFS4ERR_TOO_MANY_OPS	
NFS4ERR_REP_TOO_BIG	
NFS4ERR_REP_TOO_BIG_TO_CACHE	
NFS4ERR_REQ_TOO_BIG	

Table 9

17. Operations: REQUIRED, RECOMMENDED, or OPTIONAL

The following tables summarize the operations of the NFSv4.1 protocol and the corresponding designation of REQUIRED, RECOMMENDED, OPTIONAL to implement or MUST NOT implement. The designation of MUST NOT implement is reserved for those operations that were defined in NFSv4.0 and MUST NOT be implemented in NFSv4.1.

For the most part, the REQUIRED, RECOMMENDED, or OPTIONAL designation for operations sent by the client is for the server implementation. The client is generally required to implement the operations needed

for the operating environment for which it serves. For example, a read-only NFSv4.1 client would have no need to implement the WRITE operation and is not required to do so.

The REQUIRED or OPTIONAL designation for callback operations sent by the server is for both the client and server. Generally, the client has the option of creating the backchannel and sending the operations on the fore channel that will be a catalyst for the server sending callback operations. A partial exception is CB_RECALL_SLOT; the only way the client can avoid supporting this operation is by not creating a backchannel.

Since this is a summary of the operations and their designation, there are subtleties that are not presented here. Therefore, if there is a question of the requirements of implementation, the

operation descriptions themselves must be consulted along with other relevant explanatory text within this specification.

The abbreviations used in the second and third columns of the table are defined as follows.

REQ REQUIRED to implement

REC RECOMMEND to implement

OPT OPTIONAL to implement

MNI MUST NOT implement

For the NFSv4.1 features that are OPTIONAL, the operations that support those features are OPTIONAL and the server would return NFS4ERR_NOTSUPP in response to the client's use of those operations. If an OPTIONAL feature is supported, it is possible that a set of operations related to the feature become REQUIRED to implement. The third column of the table designates the feature(s) and if the operation is REQUIRED or OPTIONAL in the presence of support for the feature.

The OPTIONAL features identified and their abbreviations are as follows:

pNFS Parallel NFS

FDELG File Delegations

DDELG Directory Delegations

Operations

Operation	REQ, REC, OPT, or MNI	Feature (REQ, REC, or OPT)	Definition
ACCESS	REQ		Section 18.1
BACKCHANNEL_CTL	REQ		Section 18.33
BIND_CONN_TO_SESSION	REQ		Section 18.34
CLOSE	REQ		Section 18.2
COMMIT	REQ		Section 18.3
CREATE	REQ		Section 18.4
CREATE_SESSION	REQ		Section 18.36
DELEGPURGE	OPT	FDELG (REQ)	Section 18.5

Shepler, et al.

Expires March 9, 2009

[Page 396]

Internet-Draft

NFSv4.1

September 2008

DELEGRETURN	OPT	FDELG, DDELG, pNFS (REQ)	Section 18.6
DESTROY_CLIENTID	REQ		Section 18.50
DESTROY_SESSION	REQ		Section 18.37
EXCHANGE_ID	REQ		Section 18.35
FREE_STATEID	REQ		Section 18.38
GETATTR	REQ		Section 18.7
GETDEVICEINFO	OPT	pNFS (REQ)	Section 18.40
GETDEVICELIST	OPT	pNFS (OPT)	Section 18.41
GETFH	REQ		Section 18.8
GET_DIR_DELEGATION	OPT	DDELG (REQ)	Section 18.39
LAYOUTCOMMIT	OPT	pNFS (REQ)	Section 18.42
LAYOUTGET	OPT	pNFS (REQ)	Section 18.43
LAYOUTRETURN	OPT	pNFS (REQ)	Section 18.44
LINK	OPT		Section 18.9
LOCK	REQ		Section 18.10
LOCKT	REQ		Section 18.11
LOCKU	REQ		Section 18.12

LOOKUP	REQ		Section 18.13
LOOKUPP	REQ		Section 18.14
NVERIFY	REQ		Section 18.15
OPEN	REQ		Section 18.16
OPENATTR	OPT		Section 18.17
OPEN_CONFIRM	MNI		N/A
OPEN_DOWNGRADE	REQ		Section 18.18
PUTFH	REQ		Section 18.19
PUTPUBFH	REQ		Section 18.20
PUTROOTFH	REQ		Section 18.21
READ	REQ		Section 18.22
READDIR	REQ		Section 18.23
READLINK	OPT		Section 18.24
RECLAIM_COMPLETE	REQ		Section 18.51
RELEASE_LOCKOWNER	MNI		N/A
REMOVE	REQ		Section 18.25
RENAME	REQ		Section 18.26
RENEW	MNI		N/A
RESTOREFH	REQ		Section 18.27
SAVEFH	REQ		Section 18.28
SECINFO	REQ		Section 18.29
SECINFO_NO_NAME	REC	pNFS files layout (REQ)	Section 18.45 , Section 13.12
SEQUENCE	REQ		Section 18.46
SETATTR	REQ		Section 18.30
SETCLIENTID	MNI		N/A
SETCLIENTID_CONFIRM	MNI		N/A
SET_SSV	REQ		Section 18.47
TEST_STATEID	REQ		Section 18.48

VERIFY	REQ		Section 18.31
WANT_DELEGATION	OPT	FDELG (OPT)	Section 18.49
WRITE	REQ		Section 18.32

Callback Operations:

Callback Operations

Operation	REQ, REC, OPT, or	Feature (REQ, REC,	Definition
-----------	----------------------	-----------------------	------------

	MNI	or OPT)	
CB_GETATTR	OPT	FDELG (REQ)	Section 20.1
CB_LAYOUTRECALL	OPT	pNFS (REQ)	Section 20.3
CB_NOTIFY	OPT	DDELG (REQ)	Section 20.4
CB_NOTIFY_DEVICEID	OPT	pNFS (OPT)	Section 20.4
CB_NOTIFY_LOCK	OPT		Section 20.11
CB_PUSH_DELEG	OPT	FDELG (OPT)	Section 20.5
CB_RECALL	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.2
CB_RECALL_ANY	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.6
CB_RECALL_SLOT	REQ		Section 20.8
CB_RECALLABLE_OBJ_AVAIL	OPT	DDELG, pNFS (REQ)	Section 20.7
CB_SEQUENCE	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.9
CB_WANTS_CANCELLED	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.10

[18.](#) NFSv4.1 Operations

[18.1.](#) Operation 3: ACCESS – Check Access Rights

[18.1.1.](#) ARGUMENTS

```
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;
};

```

[18.1.2.](#) RESULTS

```

struct ACCESS4resok {
    uint32_t      supported;
    uint32_t      access;
};

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

```

[18.1.3.](#) 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 reply's supported and access fields MUST NOT contain more values than originally set in the request's access field. For example, if the client sends an ACCESS operation with just the ACCESS4_READ value set and the server supports this value, the server

MUST NOT set more than ACCESS4_READ in the supported field even if it could have reliably checked other values.

The reply's access field MUST NOT contain more values than the supported field.

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 a regular file (no meaning for a directory).

On success, the current filehandle retains its value.

ACCESS4_EXECUTE is a challenging semantic to implement because NFS provides remote file access, not remote execution. This leads to the following:

- o Whether a regular file is executable or not ought to be the responsibility of the NFS client and not the server. And yet the ACCESS operation is specified to seemingly require a server to own that responsibility.
- o When a client executes a regular file, it has to read the file from the server. Strictly speaking, the server should not allow the client to read a file being executed unless the user has read permissions on the file. Requiring users and administrators to set read permissions on executable files in order to access them over NFS is not going to be acceptable to some people. Historically, NFS servers have allowed a user to READ a file if the user has execute access to the file.

Internet-Draft

NFSv4.1

September 2008

As a practical example, the UNIX specification [40] states that an implementation claiming conformance to UNIX may indicate in the `access()` programming interface's result that a privileged user has execute rights, even if no execute permission bits are set on the regular file's attributes. It is possible to claim conformance to the UNIX specification and instead not indicate execute rights in that situation, which is true for some operating environments. Suppose the operating environments of the client and server are implementing the `access()` semantics for privileged users differently, and the ACCESS operation implementations of the client and server follow their respective `access()` semantics. This can cause undesired behavior:

- o Suppose the client's `access()` interface returns `X_OK` if the user is privileged and no execute permission bits are set on the regular file's attribute, and the server's `access()` interface does not return `X_OK` in that situation. Then the client will be unable to execute files stored on the NFS server that could be executed if stored on a non-NFS file system.
- o Suppose the client's `access()` interface does not return `X_OK` if the user is privileged, and no execute permission bits are set on the regular file's attribute, and the server's `access()` interface does return `X_OK` in that situation. Then:
 - * The client will be able to execute files stored on the NFS server that could be executed if stored on a non-NFS file system, unless the client's execution subsystem also checks for execute permission bits.
 - * Even if the execution subsystem is checking for execute permission bits, there are more potential issues. E.g. suppose the client is invoking `access()` to build a "path search table" of all executable files in the user's "search path", where the path is a list of directories each containing executable files. Suppose there are two files each in separate directories of the search path, such that files have the same component name. In the first directory the file has no execute permission bits set, and in the second directory the file has execute bits set. The path search table will indicate that the first directory has the executable file, but the execute subsystem will fail to execute it. The command shell might fail to try the second

file in the second directory. And even if it did, this is a potential performance issue. Clearly the desired outcome for the client is for the path search table to not contain the first file.

To deal the problems described above, the smart client, stupid server

principle is used. The client owns overall responsibility for determining execute access and relies on the server to parse the execution permissions within the file's mode, acl, and dacl attributes. The rules for the client and server follow:

- o If the client is sending ACCESS in order to determine if the user can read the file, the client SHOULD set ACCESS4_READ in the request's access field.
- o If the client's operating environment only grants execution to the user if the user has execute access according to the execute permissions in the mode, acl, and dacl attributes, then if the client wants to determine execute access, the client SHOULD send an ACCESS request with ACCESS4_EXECUTE bit set in the request's access field.
- o If the client's operating environment grants execution to the user even if the user does not have execute access according to the execute permissions in the mode, acl, and dacl attributes, then if the client wants to determine execute access, it SHOULD send an ACCESS request with both the ACCESS4_EXECUTE and ACCESS4_READ bits set in the request's access field. This way, if any read or execute permission grants the user read or execute access (or if the server interprets the user as privileged), as indicated by the presence of ACCESS4_EXECUTE and/or ACCESS4_READ in the reply's access field, the client will be able to grant the user execute access to the file.
- o If the server supports execute permission bits, or some other method for denoting executability (e.g. the suffix of the name of the file might indicate execute), it MUST check only execute permissions, not read permissions, when determining whether the reply will have ACCESS4_EXECUTE set in the access field or not. The server MUST NOT also examine read permission bits when determining whether the reply will have ACCESS4_EXECUTE set in the

access field or not. Even if the server's operating environment would grant execute access to the user (e.g., the user is privileged), the server MUST NOT reply with ACCESS4_EXECUTE set in reply's access field, unless there is at least one execute permission bit set in the mode, acl, or dacl attributes. In the case of acl and dacl, the "one execute permission bit" MUST be an ACE4_EXECUTE bit set in an ALLOW ACE.

- o If the server does not support execute permission bits or some other method for denoting executability, it MUST NOT set ACCESS4_EXECUTE in the reply's supported and access fields. If the client set ACCESS4_EXECUTE in the ACCESS request's access field, and ACCESS4_EXECUTE is not set in the reply's supported

field, then the client will have to send an ACCESS request with the ACCESS4_READ bit set in the request's access field.

- o If the server supports read permission bits, it MUST only check for read permissions in the mode, acl, and dacl attributes when it receives an ACCESS request with ACCESS4_READ set the access field. The server MUST NOT also examine execute permission bits when determining whether the reply will have ACCESS4_READ set in the access field or not.

Note that if the ACCESS reply has ACCESS4_READ or ACCESS_EXECUTE set, then the user also has permissions to OPEN ([Section 18.16](#)) or READ ([Section 18.22](#)) the file. I.e., if client sends an ACCESS request with the ACCESS4_READ and ACCESS_EXECUTE set in the access field (or two separate requests, one with ACCESS4_READ set, and the other with ACCESS4_EXECUTE set), and the reply has just ACCESS4_EXECUTE set in the access field (or just one reply has ACCESS4_EXECUTE set), then the user has authorization to OPEN or READ the file.

[18.1.4](#). 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 NFSv2 protocol, the only reliable way to determine whether an operation was allowed was to try it and see if it succeeded or failed. Using the ACCESS operation in the NFSv4.1 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 interface `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.

[18.2.](#) Operation 4: CLOSE - Close File

[18.2.1.](#) ARGUMENTS

```
struct CLOSE4args {
```

```

        /* CURRENT_FH: object */
        seqid4          seqid;
        stateid4         open_stateid;
};

```

[18.2.2.](#) RESULTS

```

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

```

[18.2.3.](#) 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 byte-range locks are held, the client SHOULD release all locks before issuing a CLOSE. The server MAY free all outstanding locks on CLOSE but some servers may not support the CLOSE of a file that still

has byte-range locks held. The server MUST return failure if any locks would exist after the CLOSE.

The argument seqid MAY have any value and the server MUST ignore seqid.

On success, the current filehandle retains its value.

The server MAY require that the principal, security flavor, and applicable, the GSS mechanism, combination that sent the OPEN request also be the one to CLOSE the file. This might not be possible if credentials for the principal are no longer available. The server MAY allow the machine credential or SSV credential (see [Section 18.35](#)) to send CLOSE.

[18.2.4.](#) 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. To help find any uses of this stateid by clients, the server SHOULD return the invalid special stated (the "other" value is zero and the "seqid" field is NFS4_UINT32_MAX, see [Section 8.2.3](#)).

A CLOSE operation may make delegations grantable where they were not previously. Servers may choose to respond immediately if there are pending delegation want requests or may respond to the situation at a later time.

[18.3.](#) Operation 5: COMMIT - Commit Cached Data

[18.3.1.](#) ARGUMENTS

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

[18.3.2.](#) RESULTS

```
struct COMMIT4resok {
    verifier4      writeverf;
};

union COMMIT4res switch (nfsstat4 status) {
```

```
case NFS4_OK:
    COMMIT4resok    resok4;
default:
    void;
};
```

[18.3.3.](#) DESCRIPTION

The COMMIT operation forces or flushes uncommitted, modified 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 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 restarted; however, other events at the server may result in uncommitted data loss as well.

On success, the current filehandle retains its value.

[18.3.4.](#) 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 modified 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 modified data based on the offset and count, and flushes any modified 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 data to the server. How this is to be done is up to the implementor. If there is only one buffer of

Internet-Draft

NFSv4.1

September 2008

interest, then it should be sent in a WRITE request with the FILE_SYNC4 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. However, if the server repeatedly returns from COMMIT a verifier that differs from that returned by WRITE, the only way to ensure progress is to retransmit all of the buffers with WRITE requests with the FILE_SYNC4 stable parameter.

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.

[18.4.](#) Operation 6: CREATE - Create a Non-Regular File Object

[18.4.1.](#) ARGUMENTS

```
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;
};
```


[18.4.2.](#) RESULTS

```
struct CREATE4resok {
    change_info4    cinfo;
    bitmap4         attrset;    /* attributes set */
};

union CREATE4res switch (nfsstat4 status) {
    case NFS4_OK:
        /* new CURRENTFH: created object */
        CREATE4resok resok4;
    default:
        void;
};
```

[18.4.3.](#) DESCRIPTION

The CREATE operation creates a file object other than an ordinary file in a directory with a given name. The OPEN operation MUST be used to create a regular file or a named attribute.

The current filehandle must be a directory: an object of type NF4DIR. If the current filehandle is an attribute directory (type NF4ATTRDIR), the error NFS4ERR_WRONG_TYPE is returned. If the current file handle designates any other type of object, the error NFS4ERR_NOTDIR results.

The objname specifies the name for the new object. The objtype determines the type of object to be created: directory, symlink, etc. If the object type specified is that of an ordinary file, a named attribute, or a named attribute directory, the error NFS4ERR_BADTYPE results.

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` data type, 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_INVALID` 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, 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.

If the current filehandle designates a directory for which another client holds a directory delegation, then, unless the delegation is such that the situation can be resolved by sending a notification, the delegation MUST be recalled, and the CREATE operation MUST NOT proceed until the delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while delegation remains outstanding.

When the current filehandle designates a directory for which one or more directory delegations exist, then, when those delegations request such notifications, NOTIFY4_ADD_ENTRY will be generated as a result of this operation.

If the capability FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 is set ([Section 14.4](#)), and a symbolic link is being created, then the content of the symbolic link MUST be in UTF-8 encoding.

[18.4.4.](#) 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.

[18.5.](#) Operation 7: DELEGPURGE – Purge Delegations Awaiting Recovery

[18.5.1.](#) ARGUMENTS

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

[18.5.2.](#) RESULTS

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

[18.5.3.](#) 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.

The client is NOT specified by the `clientid` field of the request. The client SHOULD set the `client` field to zero and the server MUST ignore the `clientid` field. Instead the server MUST derive the client ID from the value of the session ID in the arguments of the SEQUENCE operation that precedes DELEGPURGE in the COMPOUND request.

This operation should be used by clients that record delegation information on stable storage on the client. In this case, DELEGPURGE should be sent 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 which 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.

[18.6.](#) Operation 8: DELEGRETURN - Return Delegation

[18.6.1.](#) ARGUMENTS

```
struct DELEGRETURN4args {  
    /* CURRENT_FH: delegated object */  
    stateid4      deleg_stateid;  
};
```

[18.6.2.](#) RESULTS

```

struct DELEGRETURN4res {
    nfsstat4      status;
};

```

[18.6.3.](#) 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.

The server MAY require that the principal, security flavor, and if applicable, the GSS mechanism, combination that acquired the delegation also be the one to send DELEGRETURN on the file. This might not be possible if credentials for the principal are no longer available. The server MAY allow the machine credential or SSV credential (see [Section 18.35](#)) to send DELEGRETURN.

[18.7.](#) Operation 9: GETATTR – Get Attributes

[18.7.1.](#) ARGUMENTS

```

struct GETATTR4args {
    /* CURRENT_FH: object */
    bitmap4      attr_request;
};

```

[18.7.2.](#) RESULTS

```

struct GETATTR4resok {
    fattr4      obj_attributes;
};

union GETATTR4res switch (nfsstat4 status) {
    case NFS4_OK:

```

```
        GETATTR4resok  resok4;
default:
        void;
};
```

[18.7.3.](#) 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 REQUIRED attributes as specified in File Attributes ([Section 11.3.1](#)), for all file systems, with the exception of absent file systems.

On success, the current filehandle retains its value.

[18.7.4.](#) IMPLEMENTATION

Suppose there is a write delegation held by another client for file in question and size and/or change are among the set of attributes being interrogated. The server has two choices. First, the server can obtain the actual current value of these attributes from the client holding the delegation by using the CB_GETATTR callback. Second, the server, particularly when the delegated client is unresponsive, can recall the delegation in question. The GETATTR MUST NOT proceed until one of the following occurs:

- o The requested attribute values are returned in the response to CB_GETATTR.
- o The write delegation is returned.
- o The write delegation is revoked.

Unless one of the above happens very quickly, one or more NFS4ERR_DELAY errors will be returned if while a delegation is outstanding.

[18.8.](#) Operation 10: GETFH - Get Current Filehandle

[18.8.1.](#) ARGUMENTS

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

[18.8.2.](#) RESULTS

```
struct GETFH4resok {  
    nfs_fh4      object;  
};  
  
union GETFH4res switch (nfsstat4 status) {  
    case NFS4_OK:  
        GETFH4resok      resok4;  
    default:  
        void;  
};
```

[18.8.3.](#) DESCRIPTION

This operation returns the current filehandle value.

On success, the current filehandle retains its value.

As described in [Section 2.10.6.4](#), GETFH is REQUIRED or RECOMMENDED to immediately follow certain operations, and servers are free to reject such operations the client fails to insert GETFH in the request as REQUIRED or RECOMMENDED. [Section 18.16.4.1](#) provides additional justification for why GETFH MUST follow OPEN.

[18.8.4](#). 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

[18.9](#). Operation 11: LINK - Create Link to a File

[18.9.1](#). ARGUMENTS

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

[18.9.2](#). RESULTS

```
struct LINK4resok {  
    change_info4    cinfo;  
};  
  
union LINK4res switch (nfsstat4 status) {  
    case NFS4_OK:  
        LINK4resok resok4;  
    default:  
        void;  
};
```


[18.9.3.](#) 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 data type, 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_INVALID will be returned.

[18.9.4.](#) IMPLEMENTATION

The server MAY impose restrictions on the LINK operation such that LINK may not be done when the file is open or when that open is done by particular protocols, or with particular options or access modes. When LINK is rejected because of such restrictions, the error NFS4ERR_FILE_OPEN is returned.

If a server does implement such restrictions and those restrictions include cases of NFSv4 opens preventing successful execution of a link, the server needs to recall any delegations which could hide the existence of opens relevant to that decision. The reason is that when a client holds a delegation, the server might not have an accurate account of the opens for that client, since the client may execute OPENS and CLOSEs locally. The LINK operation must be delayed only until a definitive result can be obtained. E.g., suppose there are multiple delegations and one of them establishes an open whose presence would prevent the link. Given the server's semantics, NFS4ERR_FILE_OPEN may be returned to the caller as soon as that delegation is returned without waiting for other delegations to be returned. Similarly, if such opens are not associated with

delegations, NFS4ERR_FILE_OPEN can be returned immediately with no delegation recall being done.

If the current filehandle designates a directory for which another client holds a directory delegation, then, unless the delegation is such that the situation can be resolved by sending a notification, the delegation MUST be recalled, and the operation cannot be performed successfully. until the delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY

errors will be returned to requests made while delegation remains outstanding.

When the current filehandle designates a directory for which one or more directory delegations exist, then, when those delegations request such notifications, instead of a recall, NOTIFY4_ADD_ENTRY will be generated as a result of the LINK operation.

If the current file system supports the numlinks attribute, and other clients have delegations to the file being linked, then those delegations MUST be recalled and the LINK operation MUST NOT proceed until all delegations are returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while delegation remains outstanding.

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

The statement "file and the target directory must reside within the same file system on the server" means that the fsid fields in the attributes for the objects are the same. If they reside on different file systems, the error NFS4ERR_XDEV is returned. This error may be returned by some server when there is an internal partitioning of a file system which the LINK operation would violate.

On some servers, "." and ".." are illegal values for newname and the error NFS4ERR_BADNAME will be returned if they are specified.

When the current filehandle designates a named attribute directory and the object to be linked (the saved filehandle) is not a named

attribute for the same object, the error NFS4ERR_XDEV MUST be returned. When the saved filehandle designates a named attribute and the current filehandle is not the appropriate named attribute directory, the error NFS4ERR_XDEV MUST also be returned.

When the current filehandle designates a named attribute directory and the object to be linked (the saved filehandle) is a named attribute within that directory, the server may return the error NFS4ERR_NOTSUPP.

In the case that newname is already linked to the file represented by the saved filehandle, the server will return NFS4ERR_EXIST.

Note that symbolic links are created with the CREATE operation.

[18.10](#). Operation 12: LOCK - Create Lock

[18.10.1](#). 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;
};

```

[18.10.2](#). RESULTS

```

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;
};
```

[18.10.3.](#) DESCRIPTION

The LOCK operation requests a byte-range lock for the byte range specified by the offset and length parameters, and lock type specified in the locktype parameter. If this is a reclaim request, the reclaim parameter will be TRUE.

Bytes in a file may be locked even if those bytes are not currently allocated to the file. To lock the file from a specific offset through the end-of-file (no matter how long the file actually is) use a length field equal to NFS4_UINT64_MAX. The server MUST return NFS4ERR_INVAL under the following combinations of length and offset:

- o Length is equal to zero.
- o Length is not equal to NFS4_UINT64_MAX, and the sum of length and offset exceeds NFS4_UINT64_MAX.

32-bit servers are servers that support locking for byte offsets that fit within 32 bits (i.e. less than or equal to NFS4_UINT32_MAX). If the client specifies a range that overlaps one or more bytes beyond offset NFS4_UINT32_MAX, but does not end at offset NFS4_UINT64_MAX, then such a 32-bit server MUST return the error NFS4ERR_BAD_RANGE.

If the server returns NFS4ERR_DENIED, 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 byte-range locking state associated with the current open file and lock-owner. In the case in which it has, the argument is just a stateid representing the set of locks associated with that open file and lock-owner, together with a lock_seqid value which MAY be any value and MUST be ignored by the server. In the case where no byte-range locking 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 with 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 following fields of the locker parameter MAY be set to any value by the client and MUST be ignored by the server:

- o The clientid field of the lock_owner field of the open_owner field (locker.open_owner.lock_owner.clientid). The reason the server MUST ignore the clientid field is that the server MUST derive the client ID from the session ID from the SEQUENCE operation of the COMPOUND request.
- o The open_seqid and lock_seqid fields of the open_owner field (locker.open_owner.open_seqid and locker.open_owner.lock_seqid).
- o The lock_seqid field of the lock_owner field (locker.lock_owner.lock_seqid).

Note that 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. Thus if the server returns NFS4ERR_DENIED, it MUST set the clientid field of the owner field of the denied field.

If the current filehandle is not an ordinary file, an error will be returned to the client. In the case that the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. if the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases, NFS4ERR_WRONG_TYPE is returned.

On success, the current filehandle retains its value.

[18.10.4](#). 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

LOCK operations are subject to permission checks and to checks against the access type of the associated file. However, the specific right and modes required for various type of locks, reflect the semantics of the server-exported file system, and are not specified by the protocol. For example, Windows 2000 allows a write lock of a file open for READ, while a POSIX-compliant system does not.

When the client makes a lock request that corresponds to a range that the lock-owner has locked already (with the same or different lock type), or to a sub-region of such a range, or to a region which includes multiple locks already granted to that lock-owner, in whole or in part, and the server does not support such locking operations (i.e. does not support POSIX locking semantics), the server will return the error NFS4ERR_LOCK_RANGE. In that case, the client may return an error, or it may emulate the required operations, using only LOCK for ranges that do not include any bytes already locked by that lock-owner and LOCKU of locks held by that lock-owner (specifying an exactly-matching range and type). Similarly, when the client makes a lock request that amounts to upgrading (changing from a read lock to a write lock) or downgrading (changing from write lock to a read lock) an existing byte-range lock, and the server does not support such a lock, the server will return NFS4ERR_LOCK_NOTSUPP. Such operations may not perfectly reflect the required semantics in the face of conflicting lock requests from other clients.

When a client holds a write delegation, the client holding that delegation is assured that there are no opens by other clients. Thus, there can be no conflicting LOCK requests from such clients. Therefore, the client may be handling locking requests locally, without doing LOCK operations on the server. If it does that, it must be prepared to update the lock status on the server, by doing appropriate LOCK and LOCKU requests before returning the delegation.

When one or more clients hold read delegations, any LOCK request where the server is implementing mandatory locking semantics, MUST result in the recall of all such delegations. The LOCK request may not be granted until all such delegations are return or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while the delegation remains outstanding.

[18.11.](#) Operation 13: LOCKT - Test For Lock

[18.11.1.](#) ARGUMENTS

```
struct LOCKT4args {  
    /* CURRENT_FH: file */  
    nfs_lock_type4  locktype;  
    offset4         offset;  
    length4         length;  
    lock_owner4     owner;  
};
```

[18.11.2.](#) RESULTS

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

[18.11.3.](#) 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 clientid field of the owner MAY be set to any value by the client and MUST be ignored by the server. The reason the server MUST ignore the clientid field is that the server MUST derive the client ID from the session ID from the SEQUENCE operation of the COMPOUND request.

If the current filehandle is not an ordinary file, an error will be

returned to the client. In the case that the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. if the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases, NFS4ERR_WRONG_TYPE is returned.

On success, the current filehandle retains its value.

[18.11.4.](#) 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.

LOCKT uses a lock_owner4 rather a stateid4, as is used in LOCK to identify the owner. This is because the client does not have to open the file to test for the existence of a lock, so a stateid might not be available.

As noted in [Section 18.10.4](#), some servers may return NFS4ERR_LOCK_RANGE to certain (otherwise non-conflicting) lock requests that overlap ranges already granted to the current lock-owner.

The LOCKT operation's test for conflicting locks SHOULD exclude locks for the current lock-owner, and thus should return NFS4_OK in such cases. Note that this means that a server might return NFS4_OK to a LOCKT request even though a LOCK request for the same range and lock owner would fail with NFS4ERR_LOCK_RANGE.

When a client holds a write delegation, it may choose (see [Section 18.10.4](#)) to handle LOCK requests locally. In such a case, LOCKT requests will similarly be handled locally.

[18.12.](#) Operation 14: LOCKU - Unlock File

[18.12.1.](#) ARGUMENTS

```
struct LOCKU4args {  
    /* CURRENT_FH: file */  
    nfs_lock_type4  locktype;
```

```

        seqid4          seqid;
        stateid4         lock_stateid;
        offset4          offset;
        length4          length;
};

```

[18.12.2.](#) RESULTS

```

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

```

[18.12.3.](#) DESCRIPTION

The LOCKU operation unlocks the byte-range lock specified by the parameters. The client may set the locktype field to any value that is legal for the `nfs_lock_type4` enumerated type, and the server MUST accept any legal value for locktype. Any legal value for locktype has no effect on the success or failure of the LOCKU operation.

The ranges are specified as for LOCK. The NFS4ERR_INVAL and NFS4ERR_BAD_RANGE errors are returned under the same circumstances as for LOCK.

The seqid parameter MAY be any value and the server MUST ignore it.

If the current filehandle is not an ordinary file, an error will be returned to the client. In the case that the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. if the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases, NFS4ERR_WRONG_TYPE is returned.

On success, the current filehandle retains its value.

The server MAY require that the principal, security flavor, and

applicable, the GSS mechanism, combination that sent a LOCK request also be the one to send LOCKU on the file. This might not be possible if credentials for the principal are no longer available. The server MAY allow the machine credential or SSV credential (see [Section 18.35](#)) to send LOCKU.

[18.12.4](#). IMPLEMENTATION

If the area to be unlocked does not correspond exactly to a lock actually held by the lock-owner the server may return the error NFS4ERR_LOCK_RANGE. This includes the case in which the area is not locked, where the area is a sub-range of the area locked, where it overlaps the area locked without matching exactly or the area specified includes multiple locks held by the lock-owner. In all of these cases, allowed by POSIX locking semantics, a client receiving

Shepler, et al.

Expires March 9, 2009

[Page 424]

Internet-Draft

NFSv4.1

September 2008

this error, should if it desires support for such operations, simulate the operation using LOCKU on ranges corresponding to locks it actually holds, possibly followed by LOCK requests for the sub-ranges not being unlocked.

When a client holds a write delegation, it may choose (See [Section 18.10.4](#)) to handle LOCK requests locally. In such a case, LOCKU requests will similarly be handled locally.

[18.13](#). Operation 15: LOOKUP - Lookup Filename

[18.13.1](#). ARGUMENTS

```
struct LOOKUP4args {
    /* CURRENT_FH: directory */
    component4      objname;
};
```

[18.13.2](#). RESULTS

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

[18.13.3.](#) 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.

[18.13.4.](#) 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):

Shepler, et al.

Expires March 9, 2009

[Page 425]

Internet-Draft

NFSv4.1

September 2008

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

Unlike NFSv3, NFSv4.1 allows 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 "..". NFSv4.1 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.

[18.14.](#) Operation 16: LOOKUPP – Lookup Parent Directory

[18.14.1.](#) ARGUMENTS

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

[18.14.2.](#) RESULTS

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

[18.14.3.](#) 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 is the case with 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 SECINFO_NO_NAME operation ([Section 18.45](#)), so that the client can gracefully determine the correct security flavor.

If the current filehandle is a named attribute directory that is associated with a file system object via OPENATTR (i.e. not a sub-directory of a named attribute directory) LOOKUPP SHOULD return the filehandle of the associated file system object.

[18.14.4](#). IMPLEMENTATION

An issue to note 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 an object to always be a directory. 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).

[18.15](#). Operation 17: NVERIFY - Verify Difference in Attributes

[18.15.1](#). ARGUMENTS

```
struct NVERIFY4args {
```

```

        /* CURRENT_FH: object */
        fattr4          obj_attributes;
};

```

[18.15.2.](#) RESULTS

```

struct NVERIFY4res {
    nfsstat4          status;
};

```

[18.15.3.](#) DESCRIPTION

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

On success, the current filehandle retains its value.

[18.15.4.](#) 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:

```

SEQUENCE
PUTFH fh
NVERIFY attrbits attrs
READ 0 32767

```

Contrast this with NFSv3, which would first send a GETATTR in one request/reply round trip, and then if attributes indicated that the client's cache was stale, then send a READ in another request/reply round trip.

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 set-only attribute (e.g. `time_modify_set`) is specified, the error `NFS4ERR_INVALID` is returned to the client.

[18.16.](#) Operation 18: OPEN – Open a Regular File

[18.16.1.](#) ARGUMENTS

```
/*
 * Various definitions for OPEN
 */
enum createmode4 {
    UNCHECKED4          = 0,
    GUARDED4            = 1,
    /* Deprecated in NFSv4.1. */
    EXCLUSIVE4          = 2,
    /*
     * New to NFSv4.1. If session is persistent,
     * GUARDED4 MUST be used. Otherwise, use
     * EXCLUSIVE4_1 instead of EXCLUSIVE4.
     */
    EXCLUSIVE4_1        = 3
};

struct creatverfattr {
    verifier4           cva_verf;
    fattr4              cva_attrs;
};

union createhow4 switch (createmode4 mode) {
    case UNCHECKED4:
    case GUARDED4:
        fattr4          createattrs;
    case EXCLUSIVE4:
        verifier4        createverf;
    case EXCLUSIVE4_1:
        creatverfattr     ch_createboth;
};

enum opentype4 {
    OPEN4_NOCREATE      = 0,
    OPEN4_CREATE        = 1
};

union openflag4 switch (opentype4 opentype) {
    case OPEN4_CREATE:
        createhow4       how;
    default:

```


Internet-Draft

NFSv4.1

September 2008

```
        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 {
    /*
     * Not a reclaim.
     */
    CLAIM_NULL              = 0,

    CLAIM_PREVIOUS          = 1,
    CLAIM_DELEGATE_CUR      = 2,
    CLAIM_DELEGATE_PREV     = 3,

    /*
     * Not a reclaim.
     *
     * Like CLAIM_NULL, but object identified
     * by the current filehandle.
     */
    CLAIM_FH                = 4, /* new to v4.1 */

    /*
     * Like CLAIM_DELEGATE_CUR, but object identified
     * by current filehandle.
     */
}

```

```

CLAIM_DELEG_CUR_FH      = 5, /* new to v4.1 */

/*
 * Like CLAIM_DELEGATE_PREV, but object identified
 * by current filehandle.
 */
CLAIM_DELEG_PREV_FH     = 6 /* new to v4.1 */
};

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;

/*
 * Like CLAIM_NULL. No special rights
 * to file. Ordinary OPEN of the
 * specified file by current filehandle.
 */
case CLAIM_FH: /* new to v4.1 */
    /* CURRENT_FH: regular file to open */
    void;

/*
 * Like CLAIM_DELEGATE_PREV. Right to file based on a
 * delegation granted to a previous boot

```

```

    * instance of the client. File is identified by
    * by filehandle.
    */
case CLAIM_DELEG_PREV_FH: /* new to v4.1 */
    /* CURRENT_FH: file being opened */
    void;

/*
 * Like CLAIM_DELEGATE_CUR. Right to file based on
 * a delegation granted by the server.
 * File is identified by filehandle.
 */
case CLAIM_DELEG_CUR_FH: /* new to v4.1 */
    /* CURRENT_FH: file being opened */
    stateid4      oc_delegate_stateid;

};

/*
 * 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;
};

```

[18.16.2.](#) 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) */

    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,
    WND4_CONTENTION          = 1,
    WND4_RESOURCE            = 2,
    WND4_NOT_SUPP_FTYPE      = 3,
    WND4_WRITE_DELEG_NOT_SUPP_FTYPE = 4,
    WND4_NOT_SUPP_UPGRADE    = 5,
    WND4_NOT_SUPP_DOWNGRADE = 6,
    WND4_CANCELED            = 7,
    WND4_IS_DIR              = 8
};

union open_none_delegation4 /* new to v4.1 */
switch (why_no_delegation4 ond_why) {
    case WND4_CONTENTION:
        bool ond_server_will_push_deleg;
    case WND4_RESOURCE:
        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;
};

/*
 * Result flags
 */

```

```

/* Client must confirm open */
const OPEN4_RESULT_CONFIRM      = 0x00000002;
/* Type of file locking behavior at the server */
const OPEN4_RESULT_LOCKTYPE_POSIX = 0x00000004;
/* Server will preserve file if removed while open */
const OPEN4_RESULT_PRESERVE_UNLINKED = 0x00000008;

/*
 * Server may use CB_NOTIFY_LOCK on locks
 * derived from this open
 */
const OPEN4_RESULT_MAY_NOTIFY_LOCK = 0x00000020;

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:
        /* New CURRENT_FH: opened file */
        OPEN4resok      resok4;
    default:
        void;
};

```

[18.16.3.](#) DESCRIPTION

The OPEN operation opens a regular file in a directory with the provided name or filehandle. OPEN can also create a file if a name is provided, and the client specifies it wants to create a file. Specification whether a file is be created or not, and the method of creation is via the openhow parameter. The openhow parameter consists of a switched union (data type.opengflag4), which switches on the value of opentype (OPEN4_NOCREATE or OPEN4_CREATE). If OPEN4_CREATE is specified, this leads to another switched union (data

type createhow4) that supports four cases of creation methods: UNCHECKED4, GUARDED4, EXCLUSIVE4, or EXCLUSIVE4_1. If opentype is OPEN4_CREATE, then the claim field of the claim field (sic) MUST be one of CLAIM_NULL, CLAIM_DELEGATE_CUR, or CLAIM_DELEGATE_PREV, because these claim methods include a component of a file name.

Upon success (which might entail creation of a new file), the current filehandle is replaced by that of the created or existing object.

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 or EXCLUSIVE4_1 and the current filehandle is a named attribute directory, the server will return EINVAL.

UNCHECKED4 means that the file should be created if a file of that name does not exist and encountering an existing regular file of that name is not an error. For this type of create, createattrs specifies the initial set of attributes for the file. The set of attributes may include any writable attribute valid for regular files. When an UNCHECKED4 create encounters an existing file, the attributes specified by createattrs are not used, except that when createattrs specifies the size attribute with a size of zero, the existing file is truncated.

If GUARDED4 is specified, the server checks for the presence of a duplicate object by name before performing the create. If a duplicate exists, NFS4ERR_EXIST is returned. If the object does not exist, the request is performed as described for UNCHECKED4.

For the UNCHECKED4 and GUARDED4 cases, where the operation is successful, the server will return to the client an attribute mask signifying which attributes were successfully set for the object.

EXCLUSIVE4_1 and EXCLUSIVE4 specify 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.

If using EXCLUSIVE4, and if the server uses attributes to store the exclusive create verifier, the server will signify which attributes it used by setting the appropriate bits in the attribute mask that is returned in the results. Unlike UNCHECKED4, GUARDED4, and

Internet-Draft

NFSv4.1

September 2008

EXCLUSIVE4_1, EXCLUSIVE4 does not support the setting of attributes at file creation, and after a successful OPEN via EXCLUSIVE4, the client MUST send a SETATTR to set attributes to a known state.

In NFSv4.1, EXCLUSIVE4 has been deprecated in favor of EXCLUSIVE4_1. Unlike EXCLUSIVE4, attributes may be provided in the EXCLUSIVE4_1 case, but because the server may use attributes of the target object to store the verifier, the set of allowable attributes may be fewer than the set of attributes SETATTR allows. The allowable attributes for EXCLUSIVE4_1 are indicated in the `suppattr_exclcreat` ([Section 5.8.1.14](#)) attribute. If the client attempts to set in `cva_attrs` an attribute that is not in `suppattr_exclcreat`, the server MUST return NFS4ERR_INVAL. The response field, `attrset` indicates both which attributes the server set from `cva_attrs`, and which attributes the server used to store the verifier. As described in [Section 18.16.4](#), the client can compare `cva_attrs.attrmask` with `attrset` to determine which attributes were used to store the verifier.

With the addition of persistent sessions and pNFS, under some conditions EXCLUSIVE4 MUST NOT be used by the client or supported by the server. The following table summarizes the appropriate and mandated exclusive create methods for implementations of NFSv4.1:

Required methods for exclusive create

Persistent Reply Cache Enabled	Server Supports pNFS	Server REQUIRED	Client Allowed
no	no	EXCLUSIVE4_1 and EXCLUSIVE4	EXCLUSIVE4_1 (SHOULD) or EXCLUSIVE4 (SHOULD NOT)
no	yes	EXCLUSIVE4_1	EXCLUSIVE4_1
yes	no	GUARDED4	GUARDED4
yes	yes	GUARDED4	GUARDED4

Table 10

If CREATE_SESSION4_FLAG_PERSIST is set in the results of

CREATE_SESSION the reply cache is persistent (see [Section 18.36](#)). If the EXCHGID4_FLAG_USE_PNFS_MDS flag is set in the results from EXCHANGE_ID, the server is a pNFS server (see [Section 18.35](#)). If the client attempts to use EXCLUSIVE4 on a persistent session, or a session derived from a EXCHGID4_FLAG_USE_PNFS_MDS client ID, the

server MUST return NFS4ERR_INVALID.

With persistent sessions, exclusive create semantics are fully achievable via GUARDED4, and so EXCLUSIVE4 or EXCLUSIVE4_1 MUST NOT be used. When pNFS is being used, the layout_hint attribute might not be supported after the file is created. Only the EXCLUSIVE4_1 and GUARDED methods of exclusive file creation allow the atomic setting of attributes.

For the target directory, the server returns change_info4 information in cinfo. With the atomic field of the change_info4 data type, the server will indicate if the before and after change attributes were obtained atomically with respect to the link creation.

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 additional discussion of SHARE semantics see [Section 9.7](#).

For each OPEN, the client provides a value for the owner field of the OPEN argument. The owner field is of data type open_owner4, and contains a field called clientid and a field called owner. The client can set the clientid field to any value and the server MUST ignore it. Instead the server MUST derive the client ID from the session ID of the SEQUENCE operation of the COMPOUND request.

The seqid field of the request is not used in NFSv4.1, but it MAY be any value and the server MUST ignore it.

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:

open type	description
CLAIM_NULL, CLAIM_FH	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 NFSv4.1) the file is identified by just the current filehandle.
CLAIM_PREVIOUS	The client is claiming basic OPEN state for a file that was held previous to a server restart. Generally used when a server is returning persistent filehandles; the client may not have the file name to reclaim the OPEN.
CLAIM_DELEGATE_CUR, CLAIM_DELEG_CUR_FH	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_CUR_FH (new to NFSv4.1), the file is identified by just the current filehandle.
CLAIM_DELEGATE_PREV, CLAIM_DELEG_PREV_FH	The client is claiming a delegation granted to a previous client instance; used after the client restarts. The server

From this the client can choose to manage file locking state in a way to handle a mis-match of file locking management.

- o 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 restart, thereby giving the client the opportunity to reclaim its open.
- o 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_INVALID will be returned. The component is also subject to the normal UTF-8, character support, and name checks. See [Section 14.5](#) for further discussion.

When an OPEN is done and the specified open-owner already has the resulting filehandle open, the result is to "OR" together the new share and deny status together with the existing status. In this case, only a single CLOSE need be done, even though multiple OPENS were completed. When such an OPEN is done, checking of share reservations for the new OPEN proceeds normally, with no exception for the existing OPEN held by the same open-owner. In this case, the stateid returned as an "other" field that matches that of the previous open while the "seqid" field is incremented to reflect the

change status due to the new open.

If the underlying file system at the server is only accessible in a read-only mode and the OPEN request has specified ACCESS_WRITE or ACCESS_BOTH, the server will return NFS4ERR_ROFS to indicate a read-only file system.

As with the CREATE operation, the server MUST derive the owner, owner ACE, group, or group ACE if any of the four attributes are required and supported by the server's file system. For an OPEN with the EXCLUSIVE4 createmode, the server has no choice, since such OPEN calls do not include the createattrs field. Conversely, if createattrs (UNCHECKED4 or GUARDED4) or cva_attrs (EXCLUSIVE4_1) is specified, and includes an owner, or owner_group, or ACE 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 an OPEN which specifies a size of zero (e.g. truncation) and the file has named attributes, the named attributes are left as is and 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 sends 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`. The field `od_whynone` in the reply indicates why no delegation was returned and will be one of:

`WND4_NOT_WANTED` The client specified `OPEN4_SHARE_ACCESS_WANT_NO_DELEG`.

`WND4_CONTENTION` There is a conflicting delegation or open on the file.

`WND4_RESOURCE` Resource limitations prevent the server from granting a delegation.

`WND4_NOT_SUPP_FTYPE` The server does not support delegations on this file type.

`WND4_WRITE_DELEG_NOT_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`,
`OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG`, or
`OPEN4_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_whynone` 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:

- o `OPEN4_SHARE_ACCESS_WANT_READ_DELEG`
- o `OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG`
- o `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`.

18.16.4. IMPLEMENTATION

In absence of a persistent session, the client invokes exclusive create by setting the `how` parameter to `EXCLUSIVE4` or `EXCLUSIVE4_1`. In these cases, the client provides a verifier that can reasonably be expected to be unique. A combination of a client identifier, perhaps the client network address, and a unique number generated by the client, perhaps the RPC transaction identifier, may be appropriate.

If the object does not exist, the server creates the object and stores the verifier in stable storage. For file systems that do not provide a mechanism for the storage of arbitrary file attributes, the server may use one or more elements of the object metadata 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 server's reply cache for storage of the verifier. A nonpersistent reply cache does not survive a crash and the session and reply cache may be deleted after a network partition that exceeds the lease time, thus opening failure windows.

An NFSv4.1 server SHOULD NOT store the verifier in any of the file's `RECOMMENDED` or `REQUIRED` attributes. If it does, the server SHOULD use `time_modify_set` or `time_access_set` to store the verifier. The server SHOULD NOT store the verifier in the following attributes: `acl` (it is desirable for access control to be established at creation), `dacl` (ditto), `mode` (ditto), `owner` (ditto), `owner_group` (ditto),

retentevt_set (it may be desired to establish retention at creation)
retention_hold (ditto), retention_set (ditto), sacl (it is desirable

for auditing control to be established at creation), size (on some servers, size may have a limited range of values), mode_set_masked (as with mode), and time_creation (a meaningful file creation should be set when the file is created). Another alternative for the server is to use a named attribute to store the verifier.

Because the EXCLUSIVE4 create method does not specify initial attributes, when processing an EXCLUSIVE4 create, the server

- o SHOULD set the owner of the file to that corresponding to the credential of request's RPC header.
- o SHOULD NOT leave the file's access control to anyone but the owner of the file.

If the server cannot support 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.

After the client has performed a successful exclusive create, the attrset response indicates which attributes were used to store the verifier. If EXCLUSIVE4 was used, the attributes set in attrset were used for the verifier. If EXCLUSIVE4_1 was used, the client determines the attributes used for the verifier by comparing attrset with cva_attrs.attrmask; any bits set in the former but not the latter identify the attributes used to store the verifier. The client MUST immediately send a SETATTR to set attributes used to store the verifier. Until it does so, the attributes used to store the verifier cannot be relied upon. The subsequent SETATTR MUST NOT occur in the same COMPOUND request as the OPEN.

Unless a persistent session is used, use of the GUARDED4 attribute does not provide exactly-once semantics. In particular, if a reply is lost and the server does not detect the retransmission of the request, the operation can fail with NFS4ERR_EXIST, even though the create was performed successfully. The client would use this behavior in the case that the application has not requested an exclusive create but has asked to have the file truncated when the file is opened. In the case of the client timing out and

retransmitting the create request, the client can use GUARDED4 to prevent against a sequence like: create, write, create (retransmitted) from occurring.

For SHARE reservations, the client MUST specify a value for share_access that is one of READ, WRITE, or BOTH. For share_deny, the client MUST specify one of NONE, READ, WRITE, or BOTH. If the client fails to do this, the server MUST return NFS4ERR_INVALID.

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 if the client ID was not created with EXCHGID4_FLAG_BIND_PRINC_STATEID set in the reply to EXCHANGE_ID, then the server MUST NOT impose any requirement that READs and WRITEs sent for an open file have the same credentials as the OPEN itself, and the server is REQUIRED to perform access checking on the READs and WRITEs themselves. Otherwise, if the reply to EXCHANGE_ID did have EXCHGID4_FLAG_BIND_PRINC_STATEID set, then with one exception, the credentials used in the OPEN request MUST match those used in the READs and WRITEs, and the stateids in the READs and WRITEs MUST match, or be derived from the stateid from the reply to OPEN. The exception is if SP4_SSV or SP4_MACH_CRED state protection is used, and the spo_must_allow result of EXCHANGE_ID includes the READ and/or

WRITE operations. In that case, the machine or SSV credential will be allowed to issue READ and/or WRITE. See [Section 18.35](#).

If the component provided to OPEN is a symbolic link, the error NFS4ERR_SYMLINK will be returned to the client, while if it is a directory the error NFS4ERR_ISDIR. If the component is neither of those but not an ordinary file, the error NFS4ERR_WRONG_TYPE is returned. 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 sends a REMOVE operation that would reduce the file's link count to zero, the server SHOULD report a value of zero for the numlinks attribute on

the file.

If another client has a delegation of the file being opened that conflicts with open being done (sometimes depending of the share_access or share_deny value specified), the delegation(s) MUST be recalled, and the operation cannot proceed until each such delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while delegation remains outstanding. In the case of a write delegation, any open by a different client will conflict, while for a read delegation only opens with one of the following characteristics will be considered conflicting:

- o The value of share_access includes the bit OPEN4_SHARE_ACCESS_WRITE.
- o The value of share_deny specifies READ or BOTH.
- o OPEN4_CREATE is specified together with UNCHECKED4, the size attribute is specified as zero (for truncation) and an existing file is truncated.

If OPEN4_CREATE is specified and the file does not exist and the current filehandle designates a directory for which another client holds a directory delegation, then, unless the delegation is such

that the situation can be resolved by sending a notification, the delegation MUST be recalled, and the operation cannot proceed until the delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while delegation remains outstanding.

If OPEN4_CREATE is specified and the file does not exist and the current filehandle designates a directory for which one or more directory delegations exist, then, when those delegations request such notifications, NOTIFY4_ADD_ENTRY will be generated as a result of this operation.

[18.16.4.1](#). 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.

See also [Section 2.10.6.4](#).

[18.17](#). Operation 19: OPENATTR – Open Named Attribute Directory

[18.17.1](#). ARGUMENTS

```
struct OPENATTR4args {  
    /* CURRENT_FH: object */  
    bool    createdir;  
};
```

[18.17.2](#). RESULTS

```

struct OPENATTR4res {
    /*
     * If status is NFS4_OK,
     *   new CURRENT_FH: named attribute
     *                   directory
     */
    nfsstat4      status;
};

```

[18.17.3.](#) 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 designate objects of 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 and its filehandle becomes the current filehandle. On the other hand, if createdir has a value of TRUE and

the named attribute directory already exists, no error results and the filehandle of the existing directory becomes the current filehandle. 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.

If the current file handle designates an object of type NF4NAMEDATTR (a named attribute) or NF4ATTRDIR (a named attribute directory), an error of NFS4ERR_WRONG_TYPE is returned to the client. Name attributes or a named attribute directory may have their own named attributes.

[18.17.4.](#) IMPLEMENTATION

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

[18.18.](#) Operation 21: OPEN_DOWNGRADE - Reduce Open File Access

[18.18.1.](#) ARGUMENTS

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

[18.18.2.](#) RESULTS

```
struct OPEN_DOWNGRADE4resok {
    stateid4      open_stateid;
};

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

[18.18.3.](#) DESCRIPTION

This operation is used to adjust the access and deny states for a given open. This is necessary when a given open-owner opens the same

file multiple times with different access and deny values. 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.

Valid values for the share_access field are: OPEN4_SHARE_ACCESS_READ, OPEN4_SHARE_ACCESS_WRITE, or OPEN4_SHARE_ACCESS_BOTH. If the client specifies other values, the server MUST reply with NFS4ERR_INVALID.

Valid values for the share_deny field are: OPEN4_SHARE_DENY_NONE, OPEN4_SHARE_DENY_READ, OPEN4_SHARE_DENY_WRITE, or OPEN4_SHARE_DENY_BOTH. If the client specifies other values, the server MUST reply with NFS4ERR_INVALID.

After checking for valid values of share_access and share_deny, the server replaces the current access and deny modes on the file with share_access and share_deny subject to the following constraints:

- o The bits in share_access SHOULD equal the union of the share_access bits (not including OPEN4_SHARE_WANT_* bits) specified for some subset of the OPENS in effect for the current open-owner on the current file.
- o The bits in share_deny SHOULD equal the union of the share_deny bits specified for some subset of the OPENS in effect for the current open-owner on the current file.

If the above constraints are not respected, the server SHOULD return the error NFS4ERR_INVALID. Since share_access and share_deny bits should be subsets of those already granted, short of a defect in the client or server implementation, it is not possible for the OPEN_DOWNGRADE request to be denied because of conflicting share reservations.

The seqid argument is not used in NFSv4.1, MAY be any value, and MUST be ignored by the server.

On success, the current filehandle retains its value.

[18.18.4.](#) IMPLEMENTATION

An OPEN_DOWNGRADE operation may make read delegations grantable where they were not previously. Servers may choose to respond immediately if there are pending delegation want requests or may respond to the

situation at a later time.

[18.19.](#) Operation 22: PUTFH – Set Current Filehandle

[18.19.1.](#) ARGUMENTS

```
struct PUTFH4args {  
    nfs_fh4      object;  
};
```

[18.19.2.](#) RESULTS

```
struct PUTFH4res {  
    /*  
     * If status is NFS4_OK,  
     *   new CURRENT_FH: argument to PUTFH  
     */  
    nfsstat4      status;  
};
```

[18.19.3.](#) DESCRIPTION

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

If the security mechanism used by the requester does not meet the requirements of the filehandle provided to this operation, the server MUST return NFS4ERR_WRONGSEC.

See [Section 16.2.3.1.1](#) for more details on the current filehandle.

See [Section 16.2.3.1.2](#) for more details on the current stateid.

[18.19.4.](#) IMPLEMENTATION

Commonly used as the second operator (after SEQUENCE) in a COMPOUND request to set the context for following operations.

[18.20.](#) Operation 23: PUTPUBFH – Set Public Filehandle

[18.20.1.](#) ARGUMENT

```
void;
```

Internet-Draft

NFSv4.1

September 2008

[18.20.2](#). RESULT

```
struct PUTPUBFH4res {  
    /*  
     * If status is NFS4_OK,  
     *   new CURRENT_FH: public fh  
     */  
    nfsstat4      status;  
};
```

[18.20.3](#). 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.

PUTPUBFH also clears the current stateid.

The public filehandle represents the concepts embodied in [RFC2054](#) [31], [RFC2055](#) [32], [RFC2224](#) [41]. The intent for NFSv4.1 is that the public filehandle (represented by the PUTPUBFH operation) be used as a method of providing WebNFS server compatibility with NFSv3.

The public filehandle and the root filehandle (represented by the PUTROOTFH operation) SHOULD be equivalent. If the public and root filehandles are not equivalent, then the public filehandle MUST be a descendant of the root filehandle.

See [Section 16.2.3.1.1](#) for more details on the current filehandle.

See [Section 16.2.3.1.2](#) for more details on the current stateid.

[18.20.4](#). IMPLEMENTATION

Used as the second operator (after SEQUENCE) in an NFS request to set the context for file accessing operations that follow in the same COMPOUND request.

With the NFSv3 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](#) [41] contains further discussion of the functionality. With NFSv4.1, that type of specification is not directly available in the LOOKUP operation. The reason for this is because the component separators needed to specify absolute vs. relative are not allowed in NFSv4. Therefore, the client is

responsible for constructing its request such that the use of either PUTROOTFH or PUTPUBFH are used to signify absolute or relative evaluation of an NFS URL respectively.

Note that there are warnings mentioned in [RFC2224](#) [41] with respect to the use of absolute evaluation and the restrictions the server may place on that evaluation with respect to how much of its namespace has been made available. These same warnings apply to NFSv4.1. It is likely, therefore that because of server implementation details, an NFSv3 absolute public filehandle lookup may behave differently than an NFSv4.1 absolute resolution.

There is a form of security negotiation as described in [RFC2755](#) [42] that uses the public filehandle and an overloading of the pathname. This method is not available with NFSv4.1 as filehandles are not overloaded with special meaning and therefore do not provide the same framework as NFSv3. Clients should therefore use the security negotiation mechanisms described in [Section 2.6](#).

[18.21](#). Operation 24: PUTROOTFH - Set Root Filehandle

[18.21.1](#). ARGUMENTS

void;

[18.21.2](#). RESULTS

```
struct PUTROOTFH4res {  
    /*  
     * If status is NFS4_OK,  
     *   new CURRENT_FH: root fh  
     */  
    nfsstat4      status;
```

```
};
```

[18.21.3.](#) 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.

PUTROOTFH also clears the current stateid.

See [Section 16.2.3.1.1](#) for more details on the current filehandle.

Shepler, et al.

Expires March 9, 2009

[Page 453]

Internet-Draft

NFSv4.1

September 2008

See [Section 16.2.3.1.2](#) for more details on the current stateid.

[18.21.4.](#) IMPLEMENTATION

Commonly used as the second operator (after SEQUENCE) in an NFS request to set the context for file accessing operations that follow in the same COMPOUND request.

[18.22.](#) Operation 25: READ - Read from File

[18.22.1.](#) ARGUMENTS

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

[18.22.2.](#) RESULTS

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

```

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

```

[18.22.3.](#) 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.

Except when special stateids are used, the stateid value for a READ request represents a value returned from a previous byte-range lock or share reservation request or the stateid associated with a delegation. The stateid identifies the associated owners if any and is used by the server to verify that the associated locks are still valid (e.g. have not been revoked).

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 an ordinary file, an error will be returned to the client. In the case that the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. If the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases, NFS4ERR_WRONG_TYPE 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.

[18.22.4.](#) 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 send another READ to get the remaining data. A server may return less data than requested under several circumstances. The file may have been truncated by another client or perhaps on the server itself, changing the file size from what the requesting client believes to be the case. This would reduce the actual amount of data available to the client. It is possible that the server 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 in effect 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 byte-range lock via the LOCK operation before re-attempting the READ. When the READ completes, the client should release the byte-range lock via LOCKU.

If another client has a write delegation for the file being read, the delegation must be recalled, and the operation cannot proceed until that delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while the delegation remains outstanding. Normally,

delegations will not be recalled as a result of a READ operation since the recall will occur as a result of an earlier OPEN. However, since it is possible for a READ to be done with a special stateid, the server needs to check for this case even though the client should have done an OPEN previously.

[18.23.](#) Operation 26: READDIR – Read Directory

[18.23.1.](#) ARGUMENTS

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

[18.23.2.](#) 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;
};

```

[18.23.3.](#) 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 request's cookieverf field should be set to 0 (zero) when the request's cookie field is 0 (zero) (first directory read). On subsequent requests, the cookieverf field must match the cookieverf 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` field of the request is a hint of the maximum number of bytes of directory information that should be returned. This value represents the total 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` field of the request represents the maximum total size of all of the data being returned within the `READDIR4resok` structure and includes the XDR overhead. The server MAY return less data. If the server is unable to return a single directory entry within the `maxcount` limit, the error NFS4ERR_TOOSMALL MUST be returned to the client.

Finally, the request's `attr_request` field represents the list of attributes to be returned for each directory entry supplied by the server.

A successful reply consists of 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 cursor 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 MUST 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 `rdattr_error` ([Section 5.8.1.12](#)). With this, the server is able to communicate the failure to the client and not fail the entire operation in the instance of what might be a transient failure. Obviously, the client must request the `fattr4_rdattr_error` attribute for this method to work properly. If the client does not request the attribute, the server has no choice but to return failure for the entire `READDIR` operation.

For some file system environments, the directory entries "." and ".." have special meaning and in other environments, they do not. If the server supports these special entries within a directory, they SHOULD

Internet-Draft

NFSv4.1

September 2008

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.

[18.23.4](#). 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 enable the client to provide hints 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.

If `dircount` is zero, the server bounds the reply's size based on request's `maxcount` field.

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 might be unable to properly handle this type of failure.

The use of the `cookieverf` will also protect the client from using READDIR cookie values that might be stale. For example, if the file system has been migrated, the server might or might 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 accepts a cookie value but the underlying directory has changed and the response is invalid from the client's

context of its previous READDIR.

Since some servers will not be returning "." and ".." entries as has been done with previous versions of the NFS protocol, the client that requires these entries be present in READDIR responses must fabricate them.

[18.24.](#) Operation 27: READLINK – Read Symbolic Link

[18.24.1.](#) ARGUMENTS

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

[18.24.2.](#) RESULTS

```
struct READLINK4resok {  
    linktext4    link;  
};  
  
union READLINK4res switch (nfsstat4 status) {  
    case NFS4_OK:  
        READLINK4resok resok4;  
    default:  
        void;  
};
```

[18.24.3.](#) DESCRIPTION

READLINK reads the data associated with a symbolic link. Depending on the value of the UTF-8 capability attribute ([Section 14.4](#)), the data is encoded in UTF-8. Whether created by an NFS client or created locally on the server, the data in a symbolic link is not interpreted (except possibly to check for proper UTF-8 encoding) when created, but is simply stored.

On success, the current filehandle retains its value.

[18.24.4.](#) 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_WRONG_TYPE if the object is not of type NF4LNK.

[18.25](#). Operation 28: REMOVE - Remove File System Object

[18.25.1](#). ARGUMENTS

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

[18.25.2](#). RESULTS

```
struct REMOVE4resok {
    change_info4    cinfo;
};

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

[18.25.3](#). 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. The directory may be either of type NF4DIR or NF4ATTRDIR.

For the directory where the filename was removed, the server returns change_info4 information in cinfo. With the atomic field of the change_info4 data type, 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 (and the server is enforcing UTF-8 encoding, see [Section 14.4](#)), the error NFS4ERR_INVALID will be returned.

On success, the current filehandle retains its value.

[18.25.4](#). IMPLEMENTATION

NFSv3 required a different operator RMDIR for directory removal and REMOVE for non-directory removal. This allowed clients to skip

checking the file type when being passed a non-directory delete system call (e.g. unlink() in POSIX) to remove a directory, as well as the converse (e.g. a rmdir() on a non-directory) because they knew the server would check the file type. NFSv4.1 REMOVE can be used to delete any directory entry independent of its file type. The implementor of an NFSv4.1 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. While the traditional mechanism used is to RENAME the

file from its old name to a new hidden name, the NFSv4.1 OPEN operation MAY return a result flag, OPEN4_RESULT_PRESERVE_UNLINKED, which indicates to the client that the file will be preserved if the file has an outstanding open (see [Section 18.16](#)).

If the server finds that the file is still open when the REMOVE arrives:

- o The server SHOULD NOT delete the file's directory entry if the file was opened with OPEN4_SHARE_DENY_WRITE or OPEN4_SHARE_DENY_BOTH.
- o If the file was not opened with OPEN4_SHARE_DENY_WRITE or OPEN4_SHARE_DENY_BOTH, the server SHOULD delete the file's directory entry. However, until last CLOSE of the file, the server MAY continue to allow access to the file via its filehandle.
- o The server MUST NOT delete the directory entry if the reply from OPEN had the flag OPEN4_RESULT_PRESERVE_UNLINKED set.

The server MAY implement its own restrictions on removal of a file while it is open. The server might disallow such a REMOVE (or a removal that occurs as part of RENAME). The conditions that influence the restrictions on removal of a file while it is still open include:

- o Whether certain access protocols (i.e. not just NFS) are holding the file open.
- o Whether particular options, access modes, or policies on the server are enabled.

In all cases in which a decision is made to not allow the file's directory entry be removed because of an open, the error NFS4ERR_FILE_OPEN is returned.

Where the determination above cannot be made definitively because delegations are being held, they MUST be recalled to allow processing of the REMOVE to continue. When a delegation is held, the server's knowledge of the status of opens for that client is not to be relied

on, so that unless there are files opened with the particular deny modes by clients without delegations, the determination cannot be made until delegations are recalled, and the operation cannot proceed until each sufficient delegations have been returned or revoked to allow the server to make a correct determination.

In all cases in which delegations are recalled, the server is likely to return one or more NFS4ERR_DELAY errors while delegations remain outstanding.

If the current filehandle designates a directory for which another client holds a directory delegation, then, unless the situation can be resolved by sending a notification, the directory delegation MUST be recalled, and the operation MUST NOT proceed until the delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while delegation remains outstanding.

When the current filehandle designates a directory for which one or more directory delegations exist, then, when those delegations request such notifications, NOTIFY4_REMOVE_ENTRY will be generated as a result of this operation.

Note that when a remove occurs as a result of a RENAME, NOTIFY4_REMOVE_ENTRY will only be generated if the removal happens as a separate operation. In the case in which the removal is integrated and atomic with RENAME, the notification of the removal is integrated with notification for the RENAME. See the discussion of the NOTIFY4_RENAME_ENTRY notification in [Section 20.4](#).

[18.26](#). Operation 29: RENAME - Rename Directory Entry

[18.26.1](#). ARGUMENTS

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

```
};
```

[18.26.2.](#) RESULTS

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

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

[18.26.3.](#) 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 or preferably as part of the rename and atomic with it. See [Section 18.25.4](#) for client and server actions whenever a target is removed. Note however that when the removal is performed atomically with the rename, certain parts of the removal described there are integrated with the rename. For example, notification of the removal will not be via a NOTIFY4_REMOVE_ENTRY but will be indicated as part of the NOTIFY4_ADD_ENTRY or NOTIFY4_RENAME_ENTRY generated by the rename.

If the source object and the target 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 (e.g. they might be hard links of each other), then unless the file is open (see [Section 18.26.4](#)), RENAME MUST perform no action and return NFS4_OK.

For both directories involved in the RENAME, the server returns change_info4 information. With the atomic field of the change_info4 data type, the server will indicate if the before and after change attributes were obtained atomically with respect to the rename.

If 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 oldname or newname have a length of 0 (zero), or if oldname or newname do not obey the UTF-8 definition, the error NFS4ERR_INVALID will be returned.

[18.26.4](#). IMPLEMENTATION

The server MAY impose restrictions on the RENAME operation such that RENAME may not be done when the file being renamed is open or when that open is done by particular protocols, or with particular options or access modes. Similar restrictions may be applied when a file exists with the target name and is open. When RENAME is rejected because of such restrictions, the error NFS4ERR_FILE_OPEN is returned.

When oldname and rename refer to the same file and that file is open in a fashion such that RENAME would normally be rejected with NFS4ERR_FILE_OPEN if oldname and newname were different files, then RENAME SHOULD be rejected with NFS4ERR_FILE_OPEN.

If a server does implement such restrictions and those restrictions include cases of NFSv4 opens preventing successful execution of a rename, the server needs to recall any delegations which could hide the existence of opens relevant to that decision. This is because when a client holds a delegation, the server might not have an accurate account of the opens for that client, since the client may execute OPENS and CLOSEs locally. The RENAME operation need only be delayed until a definitive result can be obtained. For example, if there are multiple delegations and one of them establishes an open whose presence would prevent the rename, given the server's semantics, NFS4ERR_FILE_OPEN may be returned to the caller as soon as

Internet-Draft

NFSv4.1

September 2008

that delegation is returned without waiting for other delegations to be returned. Similarly, if such opens are not associated with delegations, NFS4ERR_FILE_OPEN can be returned immediately with no delegation recall being done.

If the current filehandle or the saved filehandle designate a directory for which another client holds a directory delegation, then, unless the situation can be resolved by sending a notification, the delegation **MUST** be recalled, and the operation cannot proceed until the delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while delegation remains outstanding.

When the current and saved filehandles are the same and they designate a directory for which one or more directory delegations exist, then, when those delegations request such notifications, a notification of type NOTIFY4_RENAME_ENTRY will be generated as a result of this operation. When oldname and rename refer to the same file, no notification is generated (because as [Section 18.26.3](#) states, the server **MUST** take no action). When a file is removed because it has the same name as the target, if that removal is done atomically with the rename, a NOTIFY4_REMOVE_ENTRY notification will not be generated. Instead, the deletion of the file will be reported as part of the NOTIFY4_RENAME_ENTRY notification.

When the current and saved filehandles are not the same:

- o If the current filehandle designates a directory for which one or more directory delegations exist, then, when those delegations request such notifications, NOTIFY4_ADD_ENTRY will be generated as a result of this operation. When a file is removed because it has the same name as the target, if that removal is done atomically with the rename, a NOTIFY4_REMOVE_ENTRY notification will not be generated. Instead, the deletion of the file will be reported as part of the NOTIFY4_ADD_ENTRY notification.
- o If the saved filehandle designates a directory for which one or more directory delegations exist, then, when those delegations request such notifications, NOTIFY4_REMOVE_ENTRY will be generated as a result of this operation.

If the object being renamed has file delegations held by clients other than the one doing the RENAME, the delegations **MUST** be

recalled, and the operation cannot proceed until each such delegation is returned or revoked. Note that in the case of multiply linked files, the delegation recall requirement applies even if the delegation was obtained through a different name than the one being renamed. In all cases in which delegations are recalled, the server

is likely to return one or more NFS4ERR_DELAY error while the delegation(s) remains outstanding, although it may, if the returns happen quickly, not do that.

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_INVALID in these cases.

If either of the source or target filehandles are not directories, the server will return NFS4ERR_NOTDIR.

[18.27](#). Operation 31: RESTOREFH - Restore Saved Filehandle

[18.27.1](#). ARGUMENTS

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

[18.27.2](#). RESULTS

```
struct RESTOREFH4res {  
    /*  
     * If status is NFS4_OK,
```

```

        *      new CURRENT_FH: value of saved fh
        */
    nfsstat4      status;
};

```

[18.27.3.](#) DESCRIPTION

Set the current filehandle and stateid to the values in the saved filehandle and stateid. If there is no saved filehandle then the server will return the error NFS4ERR_NOFILEHANDLE.

Shepler, et al.

Expires March 9, 2009

[Page 467]

Internet-Draft

NFSv4.1

September 2008

See [Section 16.2.3.1.1](#) for more details on the current filehandle.

See [Section 16.2.3.1.2](#) for more details on the current stateid.

[18.27.4.](#) IMPLEMENTATION

Operations like OPEN and LOOKUP use the current filehandle to represent a directory and replace it with a new filehandle. Assuming the previous filehandle was saved with a SAVEFH operator, the previous filehandle can be restored as the current filehandle. This is commonly used to obtain post-operation attributes for the directory, e.g.

```

    PUTFH (directory filehandle)
    SAVEFH
    GETATTR attrbits      (pre-op dir attrs)
    CREATE optbits "foo" attrs
    GETATTR attrbits      (file attributes)
    RESTOREFH
    GETATTR attrbits      (post-op dir attrs)

```

[18.28.](#) Operation 32: SAVEFH – Save Current Filehandle

[18.28.1.](#) ARGUMENTS

```

/* CURRENT_FH: */
void;

```

[18.28.2.](#) RESULTS

```

struct SAVEFH4res {
    /*
     * If status is NFS4_OK,
     *   new SAVED_FH: value of current fh
     */
    nfsstat4      status;
};

```

[18.28.3.](#) DESCRIPTION

Save the current filehandle and stateid. 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.

See [Section 16.2.3.1.1](#) for more details on the current filehandle.

See [Section 16.2.3.1.2](#) for more details on the current stateid.

[18.28.4.](#) IMPLEMENTATION

[18.29.](#) Operation 33: SECINFO - Obtain Available Security

[18.29.1.](#) ARGUMENTS

```

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

```

[18.29.2](#). 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:
        /* CURRENTFH: consumed */
        SECINFO4resok resok4;
    default:
        void;
};

```

[18.29.3](#). 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](#) [3]), or RPCSEC_GSS (as defined in [RFC2203](#) [4]). The field flavor can also be 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 identifier (OID, as defined in [RFC2743](#) [7]), the quality of protection (as defined in [RFC2743](#) [7]) and the service type (as defined in [RFC2203](#) [4]). 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 is consumed (see [Section 2.6.3.1.1.8](#)), and if the next operation after SECINFO tries to use the current filehandle, that operation will fail with the status NFS4ERR_NOFILEHANDLE.

If the name has a length of 0 (zero), or if name does not obey the UTF-8 definition (assuming UTF-8 capabilities are enabled, see [Section 14.4](#)), the error NFS4ERR_INVALID will be returned.

See [Section 2.6](#) for additional information on the use of SECINFO.

[18.29.4](#). 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. See Table 8 for a list of operations which can return NFS4ERR_WRONGSEC. In addition, when READDIR returns attributes, the `rdattr_error` ([Section 5.8.1.12](#)) can contain NFS4ERR_WRONGSEC. Note that CREATE and REMOVE MUST NOT return NFS4ERR_WRONGSEC. The rationale for CREATE is that unless the target name exists it cannot have a separate security policy from the parent directory, and the security policy of the parent was checked when its filehandle was injected into the COMPOUND request's operations stream (for similar reasons, an OPEN operation that

creates the target MUST NOT return NFS4ERR_WRONGSEC). If the target

name exists, while it might have a separate security policy, that is irrelevant because CREATE MUST return NFS4ERR_EXIST. The rationale for REMOVE is that while that target might have separate security policy, the target is going to be removed, and so the security policy of the parent trumps that of the object being removed. RENAME and LINK MAY return NFS4ERR_WRONGSEC, but the NFS4ERR_WRONGSEC error applies only to the saved filehandle (see [Section 2.6.3.1.2](#)). Any NFS4ERR_WRONGSEC error on the current filehandle used by LINK and RENAME MUST be returned by the PUTFH, PUTPUBFH, PUTROOTFH, or RESTOREFH operation that injected the current filehandle.

With the exception of LINK and RENAME, the set of operations that can return NFS4ERR_WRONGSEC represent the point at which the client can inject a filehandle into the "current filehandle" at the server. The filehandle is either provided by the client (PUTFH, PUTPUBFH, PUTROOTFH), generated as a result of a name to filehandle translation (LOOKUP and OPEN), or generated from the saved filehandle via RESTOREFH. As [Section 2.6.3.1.1.1](#) states, a put filehandle operation followed by SAVEFH MUST NOT return NFS4ERR_WRONGSEC. Thus the RESTOREFH operation, under certain conditions (see [Section 2.6.3.1.1](#)) is permitted to return NFS4ERR_WRONGSEC so that security policies can be honored.

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 rdattrib_error attribute, the server will return the NFS4ERR_WRONGSEC error in rdattrib_error for the entry.

To resolve an error return of NFS4ERR_WRONGSEC, the client does the following:

- o For LOOKUP and OPEN, the client will use SECINFO with the same current filehandle and name as provided in the original LOOKUP or OPEN to enumerate the available security triples.
- o For the rdattrib_error, the client will use SECINFO with the same current filehandle as provided in the original READDIR. The name passed to SECINFO will be that of the directory entry (as returned from READDIR) that had the NFS4ERR_WRONGSEC error in the rdattrib_error attribute.
- o For PUTFH, PUTROOTFH, PUTPUBFH, RESTOREFH, LINK, and RENAME, 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 operation.

NOTE: In NFSv4.0, the client was required to use SECINFO, and had to reconstruct the parent of the original filehandle, and the component name of the original filehandle. The introduction in NFSv4.1 of SECINFO_NO_NAME obviates the need for reconstruction.

- o For LOOKUPP, the client will use SECINFO_NO_NAME { style = SECINFO_STYLE4_PARENT } and provide the filehandle which equals the filehandle originally provided to LOOKUPP.

See [Section 21](#) for a discussion on the recommendations for the security flavor used by SECINFO and SECINFO_NO_NAME.

[18.30.](#) Operation 34: SETATTR - Set Attributes

[18.30.1.](#) ARGUMENTS

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

[18.30.2.](#) RESULTS

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

[18.30.3.](#) 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 byte-range locks, for those cases in which a server is implementing mandatory byte-range locking behavior. A valid stateid SHOULD always be specified. When the file size attribute is not set, the special stateid consisting of all bits zero MAY be passed.

On either success or failure of the operation, the server will return the attrset bitmask to represent what (if any) attributes were successfully set. The attrset in the response is a subset of the attrmask field of the obj_attributes field in the argument.

On success, the current filehandle retains its value.

[18.30.4](#). IMPLEMENTATION

If the request specifies the owner attribute to be set, the server SHOULD allow the operation to succeed if the current owner of the object matches the value specified in the request. Some servers may be implemented in a way as to prohibit the setting of the owner attribute unless the requester has privilege to do so. If the server is lenient in this one case of matching owner values, the client implementation may be simplified in cases of creation of an object (e.g. an exclusive create via OPEN) followed by a SETATTR.

The file size attribute is used to request changes to the size of a file. A value of zero causes the file to be truncated, a value less than the current size of the file causes data from new size to the end of the file to be discarded, and a size greater than the current size of the file causes logically zeroed data bytes to be added to the end of the file. Servers are free to implement this using unallocated bytes (holes) or allocated data bytes set to zero. Clients should not make any assumptions regarding a server's implementation of this feature, beyond that the bytes in affected region returned by READ will be zeroed. Servers MUST support extending the file size via SETATTR.

SETATTR is not guaranteed to be atomic. A failed SETATTR may partially change a file's attributes, hence the reason why the reply

always includes the status and the list of attributes that were set.

If the object whose attributes are being changed has a file delegation which is held by a client other than the one doing the SETATTR, the delegation(s) must be recalled, and the operation cannot proceed to actually change an attribute until each such delegation is returned or revoked. In all cases in which delegations are recalled, the server is likely to return one or more NFS4ERR_DELAY error while the delegation(s) remains outstanding, although it may, if the

returns happen quickly, not do that.

If the object whose attributes are being set is a directory and another client holds a directory delegation for that directory, then if enabled, asynchronous notifications will be generated when the set of attributes changed has a non-null intersection with the set of attributes for which notification is requested. Notifications of type NOTIFY4_CHANGE_DIR_ATTRS will be sent to the appropriate client(s), but the SETATTR is not delayed by waiting for these notifications to be sent.

If the object whose attributes are being set is a member of directory for which another client holds a directory delegation, then asynchronous notifications will be generated when the set of attributes changed has a non-null intersection with the set of attributes for which notification is requested. Notifications of type NOTIFY4_CHANGE_CHILD_ATTRS will be sent to the appropriate clients, but the SETATTR is not delayed by waiting for these notifications to be sent.

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

The attributes time_access_set and time_modify_set are write-only attributes constructed as a switched union so the client can direct the server in setting the time values. If the switched union specifies SET_TO_CLIENT_TIME4, the client has provided an nfstime4 to be used for the operation. If the switch union does not specify SET_TO_CLIENT_TIME4, the server is to use its current time for the SETATTR operation.

If server and client times differ, programs that compare client time to file times can break. A time synchronization protocol should be used to limit client/server time skew.

Use of a COMPOUND containing a VERIFY operation specifying only the change attribute, immediately followed by a SETATTR, provides a means whereby a client may specify a request that emulates the functionality of the SETATTR guard mechanism of NFSv3. Since the function of the guard mechanism is to avoid changes to the file attributes based on stale information, delays between checking of the guard condition and the setting of the attributes have the potential to compromise this function, as would the corresponding delay in the NFSv4 emulation. Therefore, NFSv4.1 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. If the attribute masks in the request and reply are equal, the status field in the reply MUST be NFS4_OK.

[18.31](#). Operation 37: VERIFY - Verify Same Attributes

[18.31.1](#). ARGUMENTS

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

[18.31.2](#). RESULTS

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

[18.31.3.](#) DESCRIPTION

The VERIFY operation is used to verify that attributes have the value assumed by the client before proceeding with following operations in the COMPOUND request. If any of the attributes do not match then the error NFS4ERR_NOT_SAME must be returned. The current filehandle retains its value after successful completion of the operation.

[18.31.4.](#) IMPLEMENTATION

One possible use of the VERIFY operation is the following series of operations. With this the client is attempting to verify that the file being removed will match what the client expects to be removed. This series 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 series 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 rdattrib_error or any set-only attribute (e.g. time_modify_set) is specified, the error NFS4ERR_INVALID is returned to the client.

[18.32.](#) Operation 38: WRITE - Write to File

[18.32.1.](#) ARGUMENTS

```
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<>;
};

```

[18.32.2.](#) RESULTS

```

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

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

```

[18.32.3.](#) 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 write fewer bytes than requested by the client.

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 NFSv2 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 `writeverf` and that it will not commit the data and metadata at a level less than that requested by the client.

Except when special stateids are used, the `stateid` value for a `WRITE` request represents a value returned from a previous byte-range `LOCK` or `OPEN` request or the `stateid` associated with a delegation. The `stateid` identifies the associated owners if any and is used by the server to verify that the associated locks are still valid (e.g. have not been revoked).

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`. Per Table 11,

- o The server **MAY** commit the data at a stronger level than requested.

- o The server **MUST** commit the data at a level at least as high as that committed.

Valid combinations of the fields `stable` in the request and `committed`

in the reply.

+-----+-----+	
stable	committed
+-----+-----+	
UNSTABLE4	FILE_SYNC4, DATA_SYNC4, UNSTABLE4
DATA_SYNC4	FILE_SYNC4, DATA_SYNC4
FILE_SYNC4	FILE_SYNC4
+-----+-----+	

Table 11

The final portion of the result is the field `writeverf`. This field is the write verifier and is a cookie that the client can use to determine whether a server has changed instance state (e.g. server restart) between a call to `WRITE` and a subsequent call to either `WRITE` or `COMMIT`. This cookie **MUST** be unchanged during a single instance of the NFSv4.1 server and **MUST** be unique between instances of the NFSv4.1 server. If the cookie changes, then the client **MUST** assume that any data written with an `UNSTABLE4` value for committed and an old `writeverf` in the reply has been lost and will need to be recovered.

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 otherwise is treated as if a `stateid` of all bits 0 were used.

On success, the current filehandle retains its value.

[18.32.4](#). 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 will need to send another WRITE to write the remaining data.

It is assumed that the act of writing data to a file will cause the time_modified and change attributes of the file to be updated. However, these attributes 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 and change attributes of the file to be updated.

Stable storage is persistent storage that survives:

1. Repeated power failures.
2. Hardware failures (of any board, power supply, etc.).
3. Repeated software crashes and restarts.

This definition does not address failure of the stable storage module itself.

The verifier is defined to allow a client to detect different instances of an NFSv4.1 protocol server over which cached, uncommitted data may be lost. In the most likely case, the verifier allows the client to detect server restarts. 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) the server might 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 last started (if restarting the server results in lost buffers).

The reply's committed field 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.

NFS4ERR_DQUOT when a user's quota is exceeded.

In the case that the current filehandle is of type NF4DIR, the server will return NFS4ERR_ISDIR. If the current file is a symbolic link, the error NFS4ERR_SYMLINK will be returned. Otherwise, if the current filehandle does not designate an ordinary file, the server will return NFS4ERR_WRONG_TYPE.

If mandatory file locking is in effect for the file, and the corresponding byte-range of the data to be written to the file is read or write locked by an owner that is not associated with the stateid, the server MUST return NFS4ERR_LOCKED. If so, the client MUST check if the owner corresponding to the stateid used with the WRITE operation has a conflicting read lock that overlaps with the region that was to be written. If the stateid's owner has no conflicting read lock, then the client SHOULD try to get the appropriate write byte-range lock via the LOCK operation before re-attempting the WRITE. When the WRITE completes, the client SHOULD release the byte-range lock via LOCKU.

If the stateid's owner had a conflicting read lock, then the client has no choice but to return an error to the application that attempted the WRITE. The reason is that since the stateid's owner had a read lock, the server either attempted to temporarily effectively upgrade this read lock to a write lock, or the server has no upgrade capability. If the server attempted to upgrade the read lock and failed, it is pointless for the client to re-attempt the upgrade via the LOCK operation, because there might be another client also trying to upgrade. If two clients are blocked trying upgrade the same lock, the clients deadlock. If the server has no upgrade capability, then it is pointless to try a LOCK operation to upgrade.

If one or more other clients have delegations for the file being written, those delegations MUST be recalled, and the operation cannot proceed until those delegations are returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while the delegation remains outstanding. Normally, delegations will not be recalled as a result of a WRITE operation since the recall will occur as a result of an earlier OPEN. However, since it is possible for a WRITE to be done

with a special stateid, the server needs to check for this case even though the client should have done an OPEN previously.

[18.33.](#) Operation 40: BACKCHANNEL_CTL - Backchannel Control

[18.33.1.](#) ARGUMENT

```
typedef opaque gsshandle4_t<>;

struct gss_cb_handles4 {
    rpc_gss_svc_t          gcbp_service; /* RFC 2203 */
    gsshandle4_t           gcbp_handle_from_server;
    gsshandle4_t           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<>;
};
```

[18.33.2.](#) RESULT

```
struct BACKCHANNEL_CTL4res {
    nfsstat4               bcr_status;
};
```

[18.33.3.](#) DESCRIPTION

The BACKCHANNEL_CTL operation replaces the backchannel's callback program number and adds (not replaces) RPCSEC_GSS contexts for use by the backchannel.

The arguments of the BACKCHANNEL_CTL call are a subset of the CREATE_SESSION parameters. In the arguments of BACKCHANNEL_CTL, the bca_cb_program field and bca_sec_parms fields correspond respectively to the csa_cb_program and csa_sec_parms fields of the arguments of CREATE_SESSION ([Section 18.36](#)).

BACKCHANNEL_CTL MUST appear in a COMPOUND that starts with SEQUENCE.

If the RPCSEC_GSS handle identified by gcbp_handle_from_server does not exist on the server, the server MUST return NFS4ERR_NOENT.

[18.34](#). Operation 41: BIND_CONN_TO_SESSION - Associate Connection with Session

[18.34.1](#). ARGUMENT

```
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;

    channel_dir_from_client4
                    bctsa_dir;

    bool            bctsa_use_conn_in_rdma_mode;
};
```

[18.34.2](#). RESULT

```
enum channel_dir_from_server4 {
    CDFS4_FORE      = 0x1,
    CDFS4_BACK      = 0x2,
    CDFS4_BOTH      = 0x3
};

struct BIND_CONN_TO_SESSION4resok {
    sessionid4      bctsr_sessid;

    channel_dir_from_server4
                    bctsr_dir;

    bool            bctsr_use_conn_in_rdma_mode;
};

union BIND_CONN_TO_SESSION4res
    switch (nfsstat4 bctsr_status) {
```

```

case NFS4_OK:
    BIND_CONN_TO_SESSION4resok
        bctsr_resok4;

default:    void;
};

```

[18.34.3](#). DESCRIPTION

BIND_CONN_TO_SESSION is used to associate additional connections with a session. It MUST be used on the connection being associated with the session. It MUST be the only operation in the COMPOUND procedure. If SP4_NONE ([Section 18.35](#)) state protection is used, any principal, security flavor, or RPCSEC_GSS context MAY be used to invoke the operation. If SP4_MACH_CRED is used, RPCSEC_GSS MUST be used with the integrity or privacy services, using the principal that created the client ID. If SP4_SSV is used, RPCSEC_GSS with the SSV GSS mechanism ([Section 2.10.9](#)) and integrity or privacy MUST be used.

If, when the client ID was created, the client opted for SP4_NONE state protection, the client is not required to use BIND_CONN_TO_SESSION to associate the connection with the session, unless the client wishes to associate the connection with the backchannel. When SP4_NONE protection is used, simply sending a COMPOUND request with a SEQUENCE operation is sufficient to associate the connection with the session specified in SEQUENCE.

The field bctsa_dir indicates whether the client wants to associate the connection with the fore channel or the backchannel or both channels. The value CDFC4_FORE_OR_BOTH indicates the client wants to associate the connection with both the fore channel and backchannel, but will accept the connection being associated to just the fore channel. The value CDFC4_BACK_OR_BOTH indicates the client wants to associate with both the fore and backchannel, but will accept the connection being associated with just the backchannel. The server replies in bctsr_dir which channel(s) the connection is associated with. 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

server MUST return CDFS4_FORE or CDFS4_BOTH. If the client specified CDFS4_BACK_OR_BOTH, the server MUST return CDFS4_BACK or CDFS4_BOTH.

See the CREATE_SESSION operation ([Section 18.36](#)), 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.

Invoking BIND_CONN_TO_SESSION on a connection already associated with the specified session has no effect, and the server MUST respond with NFS4_OK, unless the client is demanding changes to the set of channels the connection is associated with. If so, the server MUST return NFS4ERR_INVALID.

[18.34.4](#). IMPLEMENTATION

If a session's channel loses all connections, depending on the client ID's state protection and type of channel, the client might need to use BIND_CONN_TO_SESSION to associate a new connection. If the server restarted and does not keep the reply cache in stable storage, the server will not recognize the session ID. The client will ultimately have to invoke EXCHANGE_ID to create a new client ID and session.

Suppose SP4_SSV state protection is being used, and BIND_CONN_TO_SESSION is among the operations included in the spo_must_enforce set when the client ID was created ([Section 18.35](#)). If so, there is an issue if SET_SSV is sent, no response is returned, and the last connection associated with the client ID drops. The client, per the sessions model, MUST retry the SET_SSV. But it needs a new connection to do so, and MUST associate that connection with the session via a BIND_CONN_TO_SESSION authenticated with the SSV GSS mechanism. The problem is that the RPCSEC_GSS message integrity codes use a subkey derived from the SSV as the key and the SSV may have changed. While there are multiple recovery strategies, a single, general strategy is described here.

- o The client reconnects.
- o The client assumes the SET_SSV was executed, and so sends BIND_CONN_TO_SESSION with the subkey (derived from the new SSV, i.e., what SET_SSV would have set the SSV to) used as the key for

the RPCSEC_GSS credential message integrity codes.

- o If the request succeeds, this means the original attempted SET_SSV did execute successfully. The client re-sends the original SET_SSV, which the server will reply to via the reply cache.
- o If the server returns an RPC authentication error, this means the server's current SSV was not changed, (and the SET_SSV was likely not executed). The client then tries BIND_CONN_TO_SESSION with the subkey derived from the old SSV as the key for the RPCSEC_GSS message integrity codes.
- o The attempted BIND_CONN_TO_SESSION with the old SSV should succeed. If so the client re-sends the original SET_SSV. If the original SET_SSV was not executed, then the server executes it. If the original SET_SSV was executed, but failed, the server will return the SET_SSV from the reply cache.

[18.35](#). Operation 42: EXCHANGE_ID - Instantiate Client ID

Exchange long hand client and server identifiers (owners), and create a client ID

[18.35.1](#). ARGUMENT

```

const EXCHGID4_FLAG_SUPP_MOVED_REFER      = 0x00000001;
const EXCHGID4_FLAG_SUPP_MOVED_MIGR      = 0x00000002;

const EXCHGID4_FLAG_BIND_PRINC_STATEID    = 0x00000100;

const EXCHGID4_FLAG_USE_NON_PNFS          = 0x00010000;
const EXCHGID4_FLAG_USE_PNFS_MDS         = 0x00020000;
const EXCHGID4_FLAG_USE_PNFS_DS          = 0x00040000;

const EXCHGID4_FLAG_MASK_PNFS             = 0x00070000;

const EXCHGID4_FLAG_UPD_CONFIRMED_REC_A   = 0x40000000;
const EXCHGID4_FLAG_CONFIRMED_R          = 0x80000000;

struct state_protect_ops4 {
    bitmap4 spo_must_enforce;
    bitmap4 spo_must_allow;
};

struct ssv_sp_parms4 {
    state_protect_ops4    ssp_ops;
    sec_oid4              ssp_hash_algs<>;
    sec_oid4              ssp_encr_algs<>;
    uint32_t              ssp_window;
    uint32_t              ssp_num_gss_handles;
};

enum state_protect_how4 {
    SP4_NONE = 0,
    SP4_MACH_CRED = 1,
    SP4_SSV = 2
};

union state_protect4_a switch(state_protect_how4 spa_how) {
    case SP4_NONE:
        void;
    case SP4_MACH_CRED:
        state_protect_ops4    spa_mach_ops;
    case SP4_SSV:
        ssv_sp_parms4         spa_ssv_parms;
};

struct EXCHANGE_ID4args {
    client_owner4           eia_clientowner;
    uint32_t               eia_flags;
    state_protect4_a        eia_state_protect;
    nfs_impl_id4           eia_client_impl_id<1>;
};

```

Internet-Draft

NFSv4.1

September 2008

[18.35.2](#). RESULT

```
struct ssv_prot_info4 {
    state_protect_ops4    spi_ops;
    uint32_t              spi_hash_alg;
    uint32_t              spi_encr_alg;
    uint32_t              spi_ssv_len;
    uint32_t              spi_window;
    gsshandle4_t          spi_handles<>;
};

union state_protect4_r switch(state_protect_how4 spr_how) {
    case SP4_NONE:
        void;
    case SP4_MACH_CRED:
        state_protect_ops4    spr_mach_ops;
    case SP4_SSV:
        ssv_prot_info4        spr_ssv_info;
};

struct EXCHANGE_ID4resok {
    clientid4              eir_clientid;
    sequenceid4            eir_sequenceid;
    uint32_t               eir_flags;
    state_protect4_r        eir_state_protect;
    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;
};
```

[18.35.3](#). DESCRIPTION

The client uses the EXCHANGE_ID operation to register a particular

client owner 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 must first be used, along with the returned `eir_sequenceid`, as arguments to `CREATE_SESSION`. If the flag `EXCHGID4_FLAG_CONFIRMED_R` is set in the

result, `eir_flags`, then `eir_sequenceid` MUST be ignored, as it has no relevancy.

`EXCHANGE_ID` MAY be sent in a `COMPOUND` procedure that starts with `SEQUENCE`. However, when a client communicates with a server for the first time, it will not have a session, so using `SEQUENCE` will not be possible. If `EXCHANGE_ID` is sent without a preceding `SEQUENCE`, then it MUST be the only operation in the `COMPOUND` procedure's request. If is not, the server MUST return `NFS4ERR_NOT_ONLY_OP`.

The `eia_clientowner` field is composed of a `co_verifier` field and a `co_ownerid` string. As noted in [Section 2.4](#), the `co_ownerid` describes the client, and the `co_verifier` is the incarnation of the client. An `EXCHANGE_ID` sent with a new incarnation of the client will lead to the server removing lock state of the old incarnation. Whereas an `EXCHANGE_ID` sent with the current incarnation and `co_ownerid` will result in an error or an update of the client ID's properties, depending on the arguments to `EXCHANGE_ID`.

A server MUST NOT use the same client ID for two different incarnations of an `eir_clientowner`.

In addition to the client ID and sequence ID, the server returns a server owner (`eir_server_owner`) and server scope (`eir_server_scope`). The former field is used for network trunking as described in [Section 2.10.5](#). The latter field is used to allow clients to determine when client IDs sent by one server may be recognized by another in the event of file system migration (see [Section 11.7.7](#)).

The client ID returned by `EXCHANGE_ID` is only unique relative to the combination of `eir_server_owner.so_major_id` and `eir_server_scope`. Thus if two servers return the same client ID, the onus is on the client to distinguish the client IDs on the basis of `eir_server_owner.so_major_id` and `eir_server_scope`. In the event two different server's claim matching `server_owner.so_major_id` and

eir_server_scope, the client can use the verification techniques discussed in [Section 2.10.5](#) to determine if the servers are distinct. If they are distinct, then the client will need to note the destination network addresses of the connections used with each server, and use the network address as the final discriminator.

The server, as defined by the unique identity expressed in the so_major_id of the server owner and the server scope, needs to track several properties of each client ID it hands out. The properties apply to the client ID and all sessions associated with the client ID. The properties are derived from the arguments and results of EXCHANGE_ID. The client ID properties include:

- o The capabilities expressed by the following bits, which come from the results of EXCHANGE_ID:

- * EXCHGID4_FLAG_SUPP_MOVED_REFERER
- * EXCHGID4_FLAG_SUPP_MOVED_MIGR
- * EXCHGID4_FLAG_BIND_PRINC_STATEID
- * EXCHGID4_FLAG_USE_NON_PNFS
- * EXCHGID4_FLAG_USE_PNFS_MDS
- * EXCHGID4_FLAG_USE_PNFS_DS

These properties may be updated by subsequent EXCHANGE_ID requests on confirmed client IDs though the server MAY refuse to change them.

- o The state protection method used, one of SP4_NONE, SP4_MACH_CRED, or SP4_SSV, as set by the spa_how field of the arguments to EXCHANGE_ID. Once the client ID is confirmed, this property cannot be updated by subsequent EXCHANGE_ID requests.
- o For SP4_MACH_CRED or SP4_SSV state protection:
 - * The list of operations that MUST use the specified state protection: spo_must_enforce, which come from the results of

EXCHANGE_ID.

- * The list of operations that MAY use the specified state protection: `spo_must_allow`, which come from the results of EXCHANGE_ID.

Once the client ID is confirmed, these properties cannot be updated by subsequent EXCHANGE_ID requests.

- o For SP4_SSV protection:

- * The OID of the hash algorithm. This property is represented by one of the algorithms in the `ssp_hash_algs` field of the EXCHANGE_ID arguments. Once the client ID is confirmed, this property cannot be updated by subsequent EXCHANGE_ID requests.
- * The OID of the encryption algorithm. This property is represented by one of the algorithms in the `ssp_encr_algs` field of the EXCHANGE_ID arguments. Once the client ID is confirmed, this property cannot be updated by subsequent EXCHANGE_ID

requests.

- * The length of the SSV. This property is represented by the `spi_ssv_len` field in the EXCHANGE_ID results. Once the client ID is confirmed, this property cannot be updated by subsequent EXCHANGE_ID requests. The length of SSV MUST be equal to the length of the key used by the negotiated encryption algorithm.
- * Number of concurrent versions of the SSV the client and server will support ([Section 2.10.9](#)). This property is represented by `spi_window`, in the EXCHANGE_ID results. The property may be updated by subsequent EXCHANGE_ID requests.
- o The client's implementation ID as represented by the `eia_client_impl_id` field of the arguments. The property may be updated by subsequent EXCHANGE_ID requests.
- o The server's implementation ID as represented by the `eir_server_impl_id` field of the reply. The property may be updated by replies to subsequent EXCHANGE_ID requests.

The `eia_flags` passed as part of the arguments and the `eir_flags` results allow the client and server to 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 cannot be set in the `eia_flags` field. If they are, the server MUST reject the operation with `NFS4ERR_INVALID`.

The `EXCHGID4_FLAG_UPD_CONFIRMED_REC_A` bit can only be set in `eia_flags`; it is always off in `eir_flags`. The `EXCHGID4_FLAG_CONFIRMED_R` bit can only be set in `eir_flags`; it is always off in `eia_flags`. If the server recognizes the `co_ownerid` and `co_verifier` as mapping to a confirmed client ID, it sets `EXCHGID4_FLAG_CONFIRMED_R` in `eir_flags`. The `EXCHGID4_FLAG_CONFIRMED_R` flag allows a client to tell if the client ID it is trying to create already exists and is confirmed.

If `EXCHGID4_FLAG_UPD_CONFIRMED_REC_A` is set in `eia_flags`, this means the client is attempting to update properties of an existing confirmed client ID (if the client wants to update properties of an unconfirmed client ID, it MUST NOT set `EXCHGID4_FLAG_UPD_CONFIRMED_REC_A`). If so, it is RECOMMENDED the client send the update `EXCHANGE_ID` operation in the same COMPOUND as a SEQUENCE so that the `EXCHANGE_ID` is executed exactly once. Whether the client can update the properties of client ID depends on the state protection it selected when the client ID was created, and the principal and security flavor it uses when sending the `EXCHANGE_ID`

request. The situations described in Sub-Paragraph 6, Sub-Paragraph 7, Sub-Paragraph 8, or Sub-Paragraph 9, of Paragraph 6 in [Section 18.35.4](#) will apply. Note that if the operation succeeds and returns a client ID that is already confirmed, the server MUST set the `EXCHGID4_FLAG_CONFIRMED_R` bit in `eir_flags`.

If `EXCHGID4_FLAG_UPD_CONFIRMED_REC_A` is not set in `eia_flags`, this means the client is trying to establish a new client ID; it is attempting to trunk data communication to the server ([Section 2.10.5](#)); or it is attempting to update properties of an unconfirmed client ID. The situations described in Sub-Paragraph 1, Sub-Paragraph 2, Sub-Paragraph 3, Sub-Paragraph 4, or Sub-Paragraph 5 of Paragraph 6 in [Section 18.35.4](#) will apply. Note that if the operation succeeds and returns a client ID that was previously

confirmed, the server MUST set the EXCHGID4_FLAG_CONFIRMED_R bit in `eir_flags`.

When the EXCHGID4_FLAG_SUPP_MOVED_REFER flag bit 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_MOVED_REFER in `eir_flags`.

When the EXCHGID4_FLAG_SUPP_MOVED_MIGR 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_MOVED_MIGR in `eir_flags`.

When EXCHGID4_FLAG_BIND_PRINC_STATEID is set, the client indicates it wants the server to bind the stateid to the principal. This means that when a principal creates a stateid, it has to be the one to use the stateid. If the server will perform binding it will return EXCHGID4_FLAG_BIND_PRINC_STATEID. The server MAY return EXCHGID4_FLAG_BIND_PRINC_STATEID even if the client does not request

it. If an update to the client ID changes the value of EXCHGID4_FLAG_BIND_PRINC_STATEID's client ID property, the effect applies only to new stateids. Existing stateids (and all stateids with the same "other" field) that were created with stateid to principal binding in force will continue to have binding in force. Existing stateids (and all stateids with same "other" field) that were created with stateid to principal not in force will continue to

have binding not in force.

The EXCHGID4_FLAG_USE_NON_PNFS, EXCHGID4_FLAG_USE_PNFS_MDS, and EXCHGID4_FLAG_USE_PNFS_DS bits are described in [Section 13.1](#) and convey roles the client ID is to be used for in a pNFS environment. The server MUST set one of the acceptable combinations of these bits (roles) in eir_flags, as specified in [Section 13.1](#). Note that the same client owner/server owner pair can have multiple roles. Multiple roles can be associated with the same client ID or with different client IDs. Thus, if a client sends EXCHANGE_ID from the same client owner to the same server owner multiple times, but specifies different pNFS roles each time, the server might return different client IDs. Given that different pNFS roles might have different client IDs, the client may ask for different properties for each role/client ID.

The spa_how field of the eia_state_protect field specifies how the client wants to protect its client, locking and session state from unauthorized changes ([Section 2.10.8.3](#)):

- o SP4_NONE. The client does not request the NFSv4.1 server to enforce state protection. The NFSv4.1 server MUST NOT enforce state protection for the returned client ID.
- o SP4_MACH_CRED. This choice is only valid if the client sent the request with RPCSEC_GSS as the security flavor, and with a service of RPC_GSS_SVC_INTEGRITY or RPC_GSS_SVC_PRIVACY. The client wants to use an RPCSEC_GSS-based machine credential to protect its state. The server MUST note the principal the EXCHANGE_ID operation was sent with, and the GSS mechanism used. These notes collectively comprise the machine credential.

After the client ID is confirmed, as long as the lease associated with the client ID is unexpired, a subsequent EXCHANGE_ID operation that uses the same eia_clientowner.co_owner as the first EXCHANGE_ID, MUST also use the same machine credential as the first EXCHANGE_ID. The server returns the same client ID for the subsequent EXCHANGE_ID as that returned from the first EXCHANGE_ID.

- o SP4_SSV. This choice is only valid if the client sent the request with RPCSEC_GSS as the security flavor, and with a service of RPC_GSS_SVC_INTEGRITY or RPC_GSS_SVC_PRIVACY. This choice indicates the client wants to use the SSV to protect state. The server records the credential used in the request as the machine credential (as defined above) for the eia_clientowner.co_owner. The CREATE_SESSION operation that confirms the client ID MUST use the same machine credential.

When a client specifies SP4_MACH_CRED or SP4_SSV, it also provides two lists of operations (each expressed as a bit map). The first list is spo_must_enforce and consists of those operations the client MUST send (subject to the server confirming the list of operations in the result of EXCHANGE_ID) with the machine credential (if SP4_MACH_CRED protection is specified) or the SSV-based credential (if SP4_SSV protection is used). The client MUST send the operations with RPCSEC_GSS credentials that specify the RPC_GSS_SVC_INTEGRITY or RPC_GSS_SVC_PRIVACY security service. Typically the first list of operations includes EXCHANGE_ID, CREATE_SESSION, DELEGPURGE, DESTROY_SESSION, BIND_CONN_TO_SESSION, and DESTROY_CLIENTID. The client SHOULD NOT specify in this list any operations that require a filehandle because the server's access policies MAY conflict with the client's choice, and thus the client would then be unable to access a subset of the server's namespace.

Note that if SP4_SSV protection is specified, and the client indicates that CREATE_SESSION must be protected with SP4_SSV, because the SSV cannot exist without a confirmed client ID, the first CREATE_SESSION MUST instead be sent using the machine credential, and the server MUST accept the machine credential.

There is a corresponding result, also called spo_must_enforce, of the operations the server will require SP4_MACH_CRED or SP4_SSV protection for. Normally the server's result equals the client's argument, but the result MAY be different. If the client requests one or more operations in the set { EXCHANGE_ID, CREATE_SESSION, DELEGPURGE, DESTROY_SESSION, BIND_CONN_TO_SESSION, DESTROY_CLIENTID }, then the result spo_must_enforce MUST include the operations the client requested from that set.

If spo_must_enforce in the results has BIND_CONN_TO_SESSION set, then connection binding enforcement is enabled, and the client MUST use the machine (if SP4_MACH_CRED protection is used) or SSV (if SP4_SSV protection is used) credential on calls to BIND_CONN_TO_SESSION.

The second list is spo_must_allow and consists of those operations the client wants to have the option of issuing with the machine credential or the SSV-based credential, even if the object the

Internet-Draft

NFSv4.1

September 2008

operations are performed on is not owned by the machine or SSV credential.

The corresponding result, also called `spo_must_allow`, consists of the operations the server will allow the client to use `SP4_SSV` or `SP4_MACH_CRED` credentials with. Normally the server's result equals the client's argument, but the result MAY be different.

The purpose of `spo_must_allow` is to allow clients to solve the following conundrum. Suppose the client ID is confirmed with `EXCHGID4_FLAG_BIND_PRINC_STATEID`, and it calls `OPEN` with the `RPCSEC_GSS` credentials of a normal user. Now suppose the user's credentials expire, and cannot be renewed (e.g. a Kerberos ticket granting ticket expires, and the user has logged off and will not be acquiring a new ticket granting ticket). The client will be unable to send `CLOSE` without the user's credentials, which is to say the client has to either leave the state on the server, or it has to re-send `EXCHANGE_ID` with a new verifier to clear all state. That is, unless the client includes `CLOSE` on the list of operations in `spo_must_allow` and the server agrees.

The `SP4_SSV` protection parameters also have:

`ssp_hash_algs`:

This is the set of algorithms the client supports for the purpose of computing the digests needed for the internal SSV GSS mechanism and for the `SET_SSV` operation. 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` [18]. The algorithm the server selects among the set is indicated in `spi_hash_alg`, a field of `spr_ssv_prot_info`. The field `spi_hash_alg` is an index into the array `ssp_hash_algs`. If the server does not support any of the offered algorithms, it returns `NFS4ERR_HASH_ALG_UNSUPP`. If `ssp_hash_algs` is empty, the server MUST return `NFS4ERR_INVALID`.

`ssp_encr_algs`:

This is the set of algorithms the client supports for the purpose of providing privacy protection for the internal SSV GSS mechanism. Each algorithm is specified as an OID. The REQUIRED algorithm for a server is `id-aes256-CBC`. The RECOMMENDED

algorithms are id-aes192-CBC and id-aes128-CBC [19]. The selected algorithm is returned in `spi_encr_alg`, an index into `ssp_encr_algs`. If the server does not support any of the offered algorithms, it returns `NFS4ERR_ENCR_ALG_UNSUPP`. If `ssp_encr_algs` is empty, the server MUST return `NFS4ERR_INVALID`.

`ssp_window`:

This is the number of SSV versions the client wants the server to maintain (i.e. each successful call to `SET_SSV` produces a new version of the SSV). If `ssp_window` is zero, the server MUST return `NFS4ERR_INVALID`. The server responds with `spi_window`, which MUST NOT exceed `ssp_window`, and MUST be at least one (1). Any requests on the backchannel or fore channel that are using a version of the SSV that is outside the window will fail with an `ONC RPC authentication error`, and the requester will have to retry them with the same slot ID and sequence ID.

`ssp_num_gss_handles`:

This is the number of `RPCSEC_GSS` handles the server should create that are based on the GSS SSV mechanism ([Section 2.10.9](#)). It is not the total number of `RPCSEC_GSS` handles for the client ID. Indeed, subsequent calls to `EXCHANGE_ID` will add `RPCSEC_GSS` handles. The server responds with a list of handles in `spi_handles`. If the client asks for at least one handle and the server cannot create it, the server MUST return an error. The handles in `spi_handles` are not available for use until the client ID is confirmed, which could be immediately if `EXCHANGE_ID` returns `EXCHGID4_FLAG_CONFIRMED_R`, or upon successful confirmation from `CREATE_SESSION`. While a client ID can span all the connections that are connected to a server sharing the same `eir_server_owner.so_major_id`, the `RPCSEC_GSS` handles returned in `spi_handles` can only be used on connections connected to a server that returns the same the `eir_server_owner.so_major_id` and `eir_server_owner.so_minor_id` on each connection. It is permissible for the client to set `ssp_num_gss_handles` to zero (0); the client can create more handles with another `EXCHANGE_ID` call.

The arguments include 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 accept a zero length `eia_client_impl_id` array, and clients MUST accept a zero length `eir_server_impl_id` array.

An example use for implementation identifiers would be diagnostic software that 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 that 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 possible 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.

[18.35.4](#). 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 incarnations (where a client restart represents a new incarnation), from the `eia_clientowner` structure of the `EXCHANGE_ID4args` structure

3. `principal`:

The principal that was defined in the RPC header's credential and/or verifier at the time the client record was established.

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 `SHOULD` 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 principal of a client record as defined by the RPC header's credential or verifier of 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`.

`confirmed`:

The client ID has been confirmed.

`unconfirmed`:

The client ID has not been confirmed.

Since `EXCHANGE_ID` is a non-idempotent operation, we must consider the possibility that retries occur as a result of a client restart, network partition, malfunctioning router, etc. Retries are

identified by the value of the `eia_clientowner` field of `EXCHANGE_ID4args` and the method for dealing with them is outlined in the scenarios below.

The scenarios are described in terms of the client record(s) a server has for a given `co_ownerid`. Note if the client ID was created specifying `SP4_SSV` state protection and `EXCHANGE_ID` as the one of the operations in `spo_must_allow`, then server MUST authorize `EXCHANGE_IDs` with the `SSV` principal in addition to the principal that created the client ID.

1. New Owner ID

If the server has no client records with `eia_clientowner.co_ownerid` matching `ownerid_arg`, and

EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is not set in the EXCHANGE_ID, then a new shorthand client ID (let us call it clientid_ret) is generated, and the following unconfirmed record is added to the server's state.

```
{ ownerid_arg, verifier_arg, principal_arg, clientid_ret,
  unconfirmed }
```

Subsequently, the server returns clientid_ret.

2. Non-Update on Existing Client ID

If the server has the following confirmed record, and the request does not have EXCHGID4_FLAG_UPD_CONFIRMED_REC_A set, then the request is the result of a retried request due to a faulty router or lost connection, or the client is trying to determine if it can perform trunking.

```
{ ownerid_arg, verifier_arg, principal_arg, clientid_ret,
  confirmed }
```

Since the record has been confirmed, the client must have received the server's reply from the initial EXCHANGE_ID request. Since the server has a confirmed record, and since EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is not set, with the possible exception of eir_server_owner.so_minor_id, the server returns the same result it did when the client ID's properties were last updated (or if never updated, the result when the client ID was created). The confirmed record is unchanged.

3. Client Collision

If EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is not set, and 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, old_clientid_ret,
confirmed }
```

If there is currently no state associated with `old_clientid_ret`, or if there is state but the lease has expired, then this case is effectively equivalent to the New Owner ID case of Paragraph 1. The confirmed record is deleted, the `old_clientid_ret` and its lock state are deleted, a new shorthand client ID is generated, and the following unconfirmed record is added to the server's state.

```
{ ownerid_arg, verifier_arg, principal_arg, clientid_ret,
unconfirmed }
```

Subsequently, the server returns `clientid_ret`.

If `old_clientid_ret` has an unexpired lease with state, then no state of `old_clientid_ret` is changed or deleted. The server returns `NFS4ERR_CLID_INUSE` to indicate the client should retry with a different value for the `eia_clientowner.co_ownerid` subfield of `EXCHANGE_ID4args`. The client record is not changed.

4. Replacement of Unconfirmed Record

If the `EXCHGID4_FLAG_UPD_CONFIRMED_REC_A` flag is not set, and the server has the following unconfirmed record then the client is attempting `EXCHANGE_ID` again on an unconfirmed client ID, perhaps due to a retry, or perhaps due to a client restart before client ID confirmation (i.e. before `CREATE_SESSION` was called), or some other reason.

```
{ ownerid_arg, *, *, old_clientid_ret, unconfirmed }
```

It is possible the properties of `old_clientid_ret` are different than those specified in the current `EXCHANGE_ID`. Whether the properties are being updated or not, to eliminate ambiguity, the server deletes the unconfirmed record, generates a new client ID (`clientid_ret`) and establishes the

following unconfirmed record:

```
{ ownerid_arg, verifier_arg, principal_arg, clientid_ret,
unconfirmed }
```

5. Client Restart

If EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is not set, and if the server has the following confirmed client record, then this request is likely from a previously confirmed client which has restarted.

```
{ ownerid_arg, old_verifier_arg, principal_arg,
old_clientid_ret, confirmed }
```

Since the previous incarnation of the same client will no longer be making requests, once the new client ID is confirmed by CREATE_SESSION, 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 restart, this request would not have been sent. 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 [Section 10.2.1](#).

After processing, clientid_ret is returned to the client and this client record is added:

```
{ ownerid_arg, verifier_arg, principal_arg, clientid_ret,
unconfirmed }
```

The previously described confirmed record continues to exist, and thus the same ownerid_arg exists in both a confirmed and unconfirmed state at the same time. The number of states can collapse to one once the server receives an applicable CREATE_SESSION or EXCHANGE_ID.

- + If the server subsequently receives a successful CREATE_SESSION that confirms clientid_ret, then the server atomically destroys the confirmed record and makes the unconfirmed record confirmed as described in [Section 18.36.4](#).

- + If the server instead subsequently receives an EXCHANGE_ID with the client owner equal to ownerid_arg, one strategy is to simply delete the unconfirmed record, and process the EXCHANGE_ID as described in the entirety of [Section 18.35.4](#).

6. Update

If EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is set, and the server has the following confirmed record, then this request is an attempt at an update.

```
{ ownerid_arg, verifier_arg, principal_arg, clientid_ret,
  confirmed }
```

Since the record has been confirmed, the client must have received the server's reply from the initial EXCHANGE_ID request. The server allows the update, and the client record is left intact.

7. Update but No Confirmed Record

If EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is set, and the server has no confirmed record corresponding ownerid_arg, then the server returns NFS4ERR_NOENT and leaves any unconfirmed record intact.

8. Update but Wrong Verifier

If EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is set, and the server has the following confirmed record, then this request is an illegal attempt at an update, perhaps because of a retry from an previous client incarnation.

```
{ ownerid_arg, old_verifier_arg, *, clientid_ret, confirmed }
```

The server returns NFS4ERR_NOT_SAME and leaves the client record intact.

9. Update but Wrong Principal

If EXCHGID4_FLAG_UPD_CONFIRMED_REC_A is set, and the server has the following confirmed record, then this request is an illegal attempt at an update by an unauthorized principal.

```
{ ownerid_arg, verifier_arg, old_principal_arg, clientid_ret,
```

confirmed }

Internet-Draft

NFSv4.1

September 2008

The server returns NFS4ERR_PERM and leaves the client record intact.

[18.36](#). Operation 43: CREATE_SESSION - Create New Session and Confirm Client ID

[18.36.1](#). ARGUMENT

```
struct channel_attrs4 {
    count4          ca_headerpadsize;
    count4          ca_maxrequestsize;
    count4          ca_maxresponsesize;
    count4          ca_maxresponsesize_cached;
    count4          ca_maxoperations;
    count4          ca_maxrequests;
    uint32_t        ca_rdma_ird<1>;
};

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;

    channel_attrs4 csa_fore_chan_attrs;
    channel_attrs4 csa_back_chan_attrs;

    uint32_t       csa_cb_program;
    callback_sec_parms4 csa_sec_parms<>;
};
```

[18.36.2.](#) RESULT

```
struct CREATE_SESSION4resok {
    sessionid4          csr_sessionid;
    sequenceid4         csr_sequence;

    uint32_t            csr_flags;

    channel_attr4        csr_fore_chan_attr;
    channel_attr4        csr_back_chan_attr;
};

union CREATE_SESSION4res switch (nfsstat4 csr_status) {
case NFS4_OK:
    CREATE_SESSION4resok    csr_resok4;
default:
    void;
};
```

[18.36.3.](#) DESCRIPTION

This operation is used by the client to create new session objects on the server.

CREATE_SESSION can be sent with or without a preceding SEQUENCE operation in the same COMPOUND procedure. If CREATE_SESSION is sent with a preceding SEQUENCE operation, any session created by CREATE_SESSION has no direct relation to the session specified in the SEQUENCE operation, although the two sessions might be associated with the same client ID. If CREATE_SESSION is sent without a preceding SEQUENCE, then it MUST be the only operation in the

COMPOUND procedure's request. If is not, the server MUST return NFS4ERR_NOT_ONLY_OP.

In addition to creating a session, CREATE_SESSION has the following effects:

- o The first session created with a new 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.
- o The connection CREATE_SESSION is sent over is associated with the session's fore channel.

The arguments and results of CREATE_SESSION are described as follows:

csa_clientid:

This is the client ID the new session will be associated with. The corresponding result is csr_sessionid, the session ID of the new session.

csa_sequence:

Each client ID serializes CREATE_SESSION via a per client ID sequence number (see [Section 18.36.4](#)). The corresponding result is csr_sequence, which MUST be equal to csa_sequence.

In the next three arguments, the client offers a value that is to be a property of the session. It is RECOMMENDED that the server accept the value. If it is not acceptable, the server MAY use a different value. Regardless, the server MUST return the value the session will use (which will be either what the client offered, or what the server is insisting on). 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 wants the server to provide a persistent reply cache. 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 a persistent reply cache, the server MUST NOT set CREATE_SESSION4_FLAG_PERSIST in the field csr_flags.

If the server is a pNFS metadata server, for reasons described in [Section 12.5.2](#) it SHOULD support CREATE_SESSION4_FLAG_PERSIST if it supports the layout_hint ([Section 5.12.4](#)) 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 backchannel as well as the fore 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, and if the connection CREATE_SESSION is called over is currently in non-RDMA mode, but has the capability to operate in RDMA mode, then 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 ([8]).

csa_fore_chan_attrs, csa_fore_chan_attrs:

The `csa_fore_chan_attrs` and `csa_back_chan_attrs` fields apply to attributes of the fore channel (which conveys requests originating from the client to the server), and the backchannel (the channel that conveys callback requests originating from the server to the client), respectively. The results are in corresponding structures called `csr_fore_chan_attrs` and `csr_back_chan_attrs`. The results establish attributes for each channel, and on all subsequent use of each channel of the session. Each structure has the following fields:

`ca_headerpadsize:`

The maximum amount of padding the requester is willing to apply to ensure that write payloads are aligned on some boundary at the replier. The replier should reply in `ca_headerpadsize` with its preferred value, or zero if padding is not in use. The replier may decrease this value but MUST NOT increase it.

`ca_maxrequestsize:`

The maximum size of a COMPOUND or CB_COMPOUND request that will be sent. This size represents the XDR encoded size of the request, including the RPC headers (including security flavor credentials and verifiers) but excludes any RPC transport framing headers. Imagine a request coming over a non-RDMA TCP/IP connection, and that it has a single Record Marking header preceding it. The maximum allowable count encoded in the header will be `ca_maxrequestsize`. If a requester sends a request that exceeds `ca_maxrequestsize`, the error

NFS4ERR_REQ_TOO_BIG will be returned per the description in [Section 2.10.6.4](#).

`ca_maxresponsesize:`

The maximum size of a COMPOUND or CB_COMPOUND reply that the requester will accept from the replier including RPC headers (see the `ca_maxrequestsize` definition). The NFSv4.1 server MUST NOT increase the value of this parameter in the CREATE_SESSION results. However, if the client selects a value for `ca_maxresponsesize` such that a replier on a channel could

never send a response, the server SHOULD return NFS4ERR_TOOSMALL in the CREATE_SESSION reply. If a requester sends a request for which the size of the reply would exceed this value, the replier will return NFS4ERR_REP_TOO_BIG, per the description in [Section 2.10.6.4](#).

ca_maxresponsesize_cached:

Like ca_maxresponsesize, but the maximum size of a reply that will be stored in the reply cache ([Section 2.10.6.1](#)). If the reply to CREATE_SESSION has ca_maxresponsesize_cached less than ca_maxresponsesize, then this is an indication to the requester on the channel that it needs to be selective about which replies it directs the replier to cache; for example large replies from nonidempotent operations (e.g. COMPOUND requests with a READ operation), should not be cached. The requester decides which replies to cache via an argument to the SEQUENCE (the sa_cachethis field, see [Section 18.46](#)) or CB_SEQUENCE (the csa_cachethis field, see [Section 20.9](#)) operations. If a requester sends a request for which the size of the reply would exceed this value, the replier will return NFS4ERR_REP_TOO_BIG_TOO_LARGE, per the description in [Section 2.10.6.4](#).

ca_maxoperations:

The maximum number of operations the replier will accept in a COMPOUND or CB_COMPOUND. The server MUST NOT increase ca_maxoperations in the reply to CREATE_SESSION. If the requester sends 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 requester will send on the session. Subsequent requests will each be assigned a slot identifier by the requester within the range 0 to ca_maxrequests - 1 inclusive.

ca_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 ONC RPC program number the server must use in any callbacks sent through the backchannel to the client. The server MUST specify an ONC RPC program number equal to csa_cb_program and an ONC RPC version number equal to 4 in callbacks sent to the client. If a CB_COMPOUND is sent to the client, the server MUST use a minor version number of 1. There is no corresponding result.

csa_sec_parms

The field csa_sec_parms is an array of acceptable security credentials the server can use on the session's backchannel. 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 authorizing the server to use AUTH_NONE on all callbacks for the session. If AUTH_SYS is specified, then the client is authorizing the server 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. There is no corresponding result.

The RPCSEC_GSS context for the backchannel is specified via a pair of values of data type gsshandle4_t. The data type gsshandle4_t represents an RPCSEC_GSS handle, and is precisely the same as the data type of the "handle" field of the rpc_gss_init_res data type defined in [Section 5.2.3.1](#), "Context Creation Response - Successful Acceptance" of [4].

The first RPCSEC_GSS handle, gcbp_handle_from_server, is the first handle the server returned to the client (either in the handle field of data type rpc_gss_init_res or one of the elements of the spi_handles field returned in the reply to EXCHANGE_ID) 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 the RPCSEC_GSS context to. The server can immediately use the value of `gcbp_handle_from_client` in the RPCSEC_GSS credential in callback RPCs. I.e., the value in `gcbp_handle_from_client` can be used as the value of the field "handle" in data type `rpc_gss_cred_t` (see [Section 5](#), "Elements of the RPCSEC_GSS Security Protocol" of [4]) in callback RPCs. The server MUST use the RPCSEC_GSS security service specified in `gcbp_service`, i.e. it MUST set the "service" field of the `rpc_gss_cred_t` data type in RPCSEC_GSS credential to the value of `gcbp_service` (see [Section 5.3.1](#), "RPC Request Header", of [4]).

If the RPCSEC_GSS handle identified by `gcbp_handle_from_server` does not exist on the server, the server will return `NFS4ERR_NOENT`.

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 (see the `seq_num` field in the `rpc_gss_cred_t` data type of [Section 5](#) and of [Section 5.3.1](#), "RPC Request Header", of [4]).

Once the session is created, the first `SEQUENCE` or `CB_SEQUENCE` received on a slot MUST have a sequence ID equal to 1; if not the server MUST return `NFS4ERR_SEQ_MISORDERED`.

[18.36.4](#). IMPLEMENTATION

To describe a possible 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 retries may occur as a result of a client restart, network partition, malfunctioning router, etc. For each client ID created by `EXCHANGE_ID`, the server maintains a separate reply cache (called the `CREATE_SESSION` reply cache) similar to the session reply cache used for `SEQUENCE` operations, with two distinctions.

- o First this is a reply cache just for detecting and processing

- o Second, the size of the client ID reply 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.

As previously stated, CREATE_SESSION can be sent with or without a preceding SEQUENCE operation. Even if SEQUENCE precedes CREATE_SESSION, the server MUST maintain the CREATE_SESSION reply cache, which is separate from the reply cache for the session associated with SEQUENCE. If CREATE_SESSION was originally sent by itself, the client MAY send a retry of the CREATE_SESSION operation within a COMPOUND preceded by SEQUENCE. If CREATE_SESSION was originally sent in a COMPOUND that started with SEQUENCE, then the client SHOULD send a retry in a COMPOUND that starts with SEQUENCE that has the same session ID as the SEQUENCE of the original request. However, the client MAY send a retry in a COMPOUND that either has no preceding SEQUENCE, or has a preceding SEQUENCE that refers to a different session than the original CREATE_SESSION. This might be necessary if the client sends a CREATE_SESSION in a COMPOUND preceded by a SEQUENCE with session ID X, and session X no longer exists. Regardless, any retry of CREATE_SESSION, with or without a preceding SEQUENCE, MUST use the same value of csa_sequence as the original.

When a client sends a successful EXCHANGE_ID and it is returned an unconfirmed client ID, the client is also returned eir_sequenceid, and the client is expected to set the value of csa_sequenceid in the client ID-confirming-CREATE_SESSION it sends with that client ID to the value of eir_sequenceid. When EXCHANGE_ID returns a new, unconfirmed client 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. Client record lookup. The server looks up the client ID in its client record table. 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. Otherwise, the server goes to phase 2.

2. Sequence ID processing. If `csa_sequenceid` is equal to the sequence ID in the client ID'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 sequence ID 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 sequence ID + 1 (accounting for wraparound), then the slot's sequence ID is set to `csa_sequenceid`, and the `CREATE_SESSION` processing goes to the next phase. A subsequent new `CREATE_SESSION` call over the same client ID MUST use a `csa_sequenceid` that is one greater than the sequence ID in the slot.

3. Client ID confirmation. If this would be the first session for the client ID, the `CREATE_SESSION` operation serves to confirm the client ID. Otherwise the client ID confirmation phase is skipped and only the session creation phase occurs. Any case in which there is more than one record with identical values for client ID represents a server implementation error. Operation in the potential valid cases is summarized as follows.

* Successful Confirmation

If the server has the following unconfirmed record, then this is the expected confirmation of an unconfirmed record.

```
{ ownerid, verifier, principal_arg, clientid_arg,
  unconfirmed }
```

As noted in [Section 18.35.4](#), the server might also have the following confirmed record.

```
{ ownerid, old_verifier, principal_arg, old_clientid,
  confirmed }
```

The server schedules the replacement of both records with:

```
{ ownerid, verifier, principal_arg, clientid_arg, confirmed
}
```

The processing of CREATE_SESSION continues on to session creation. Once the session is successfully created, the scheduled client record replacement is committed. If the session is not successfully created, then no changes are made to any client records on the server.

* Unsuccessful Confirmation

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, * }
```

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.
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 reply cache (if there is not enough space, the server returns NFS4ERR_NOSPC). For each slot in the reply cache, the server sets the sequence ID to zero (0), and records an entry containing a COMPOUND reply with zero operations and the error NFS4ERR_SEQ_MISORDERED. This way, if the first SEQUENCE request sent has a sequence ID equal to zero, the server can simply return what is in the reply cache: NFS4ERR_SEQ_MISORDERED. The client initializes its reply cache for receiving callbacks in the same way, and similarly, the first CB_SEQUENCE operation on a slot after session creation must have a sequence ID of one.
6. If the session state is created successfully, the server

associates the session with the client ID provided by the client.

7. When a request that had `CREATE_SESSION4_FLAG_CONN_RDMA` set needs to be retried, the retry **MUST** be done on a new connection that is in non-RDMA mode. If properties of the new connection are different enough that the arguments to `CREATE_SESSION` must change, then a non-retry **MUST** be sent. The server will eventually dispose of any session that was created on the original connection.

On the backchannel, the client and server might wish to have many slots, in some cases perhaps more than the fore channel, in to deal with the situations where the network link has high latency and is the primary bottleneck for response to recalls. If so, and if the client provides too few slots to the backchannel, the server might limit the number of recallable objects it gives to the server.

Implementing `RPCSEC_GSS` callback support requires the client and server change their `RPCSEC_GSS` implementations. One possible set of changes includes:

- o Adding a data structure that wraps the GSS-API context with a reference count.
- o 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.
- o 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.
- o Adding a function to map an existing `RPCSEC_GSS` handle to a pointer to the wrapper data structure. The reference count would be incremented.
- o Adding a function to create a new `RPCSEC_GSS` handle from a pointer to the wrapper data structure. The reference count would be incremented.

- o Replacing calls from RPCSEC_GSS that free GSS-API contexts, with calls to decrement the reference count on the wrapper data structure.

[18.37.](#) Operation 44: DESTROY_SESSION – Destroy a Session

[18.37.1.](#) ARGUMENT

```
struct DESTROY_SESSION4args {  
    sessionid4    dsa_sessionid;  
};
```

[18.37.2.](#) RESULT

```
struct DESTROY_SESSION4res {  
    nfsstat4      dsr_status;  
};
```

[18.37.3.](#) DESCRIPTION

The DESTROY_SESSION operation closes the session and discards the session's reply cache, if any. Any remaining connections associated with the session are immediately disassociated. If the connection has no remaining associated sessions, the connection MAY 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.

DESTROY_SESSION MUST be invoked on a connection that is associated with the session being destroyed. In addition if SP4_MACH_CRED state protection was specified when the client ID was created, the RPCSEC_GSS principal that created the session MUST be the one that destroys the session, using RPCSEC_GSS privacy or integrity. If SP4_SSV state protection was specified when the client ID was created, RPCSEC_GSS using the SSV mechanism ([Section 2.10.9](#)) MUST be used, with integrity or privacy.

If the COMPOUND request starts with SEQUENCE, and if the sessionids specified in SEQUENCE and DESTROY_SESSION are the same, then

- o DESTROY_SESSION MUST be the final operation in the COMPOUND request.
- o It is advisable to not place DESTROY_SESSION in a COMPOUND request with other state-modifying operations, because the DESTROY_SESSION will destroy the reply cache.

DESTROY_SESSION MAY be the only operation in a COMPOUND request.

Because the session is destroyed, a client that retries the request may receive an error in reply to the retry, even though the original request was successful.

If there is a backchannel on the session and the server has outstanding CB_COMPOUND operations for the session which have not been replied to, then the server MAY refuse to destroy the session and return an error. In the event the backchannel is down, the server SHOULD return NFS4ERR_CB_PATH_DOWN to inform the client that the backchannel needs to be repaired before the server will allow the session to be destroyed. Otherwise, the error CB_BACK_CHAN_BUSY SHOULD be returned to indicate that there are CB_COMPOUNDS that need to be replied to. The client SHOULD reply to all outstanding CB_COMPOUNDS before re-sending DESTROY_SESSION.

[18.38](#). Operation 45: FREE_STATEID - Free Stateid with No Locks

[18.38.1](#). ARGUMENT

```
struct FREE_STATEID4args {
    stateid4      fsa_stateid;
};
```

[18.38.2](#). RESULT

```
struct FREE_STATEID4res {
    nfsstat4      fsr_status;
};
```

[18.38.3.](#) DESCRIPTION

The FREE_STATEID operation is used to free a stateid which no longer has any associated locks (including opens, byte-range locks, delegations, layouts). This may be because 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. This allows 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 8.4.2](#).

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.

[18.39.](#) Operation 46: GET_DIR_DELEGATION - Get a directory delegation

[18.39.1.](#) ARGUMENT

```
typedef nfstime4 attr_notice4;

struct GET_DIR_DELEGATION4args {
    /* CURRENT_FH: delegated directory */
    bool          gdda_signal_deleg_avail;
    bitmap4       gdda_notification_types;
    attr_notice4  gdda_child_attr_delay;
    attr_notice4  gdda_dir_attr_delay;
    bitmap4       gdda_child_attributes;
    bitmap4       gdda_dir_attributes;
};
```

[18.39.2.](#) RESULT

```
struct GET_DIR_DELEGATION4resok {
    verifier4      gddr_cookieverf;
    /* Stateid for get_dir_delegation */
    stateid4       gddr_stateid;
    /* Which notifications can the server support */
    bitmap4        gddr_notification;
    bitmap4        gddr_child_attributes;
    bitmap4        gddr_dir_attributes;
};

enum 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:
    GET_DIR_DELEGATION4res_non_fatal      gddr_res_non_fatal4;
default:
    void;
};
```

[18.39.3.](#) 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 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.

Internet-Draft

NFSv4.1

September 2008

The server will also return a directory delegation stateid, gddr_stateid, as a result of the GET_DIR_DELEGATION 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 might 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, gddr_notification, contained in the reply. The server MUST NOT add bits to gddr_notification that the client did not request.

The GET_DIR_DELEGATION operation can be used for both normal and named attribute directories.

If client sets gdda_signal_deleg_avail to TRUE, then it is registering with the client a "want" for a directory delegation. If the delegation is not available, and the server supports and will honor the "want", the results will have gddrnf_will_signal_deleg_avail set to TRUE and no error will be indicated on return. If so the client should expect a future CB_RECALLABLE_OBJ_AVAIL operation to indicate that a directory delegation is available. If the server does not wish to honor the "want" or is not able to do so, it returns the error NFS4ERR_DIRDELEG_UNAVAIL. If the delegation is immediately available, the server SHOULD return it with the response to the operation, rather than via a callback.

[18.39.4.](#) IMPLEMENTATION

Directory delegations provide 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 are specified in terms of potential changes to the directory. A client can ask to be notified of events by setting one

or more bits in `gdda_notification_types`. The client can ask for notifications on addition of entries to a directory (by setting the `NOTIFY4_ADD_ENTRY` in `gdda_notification_types`), notifications on entry removal (`NOTIFY4_REMOVE_ENTRY`), renames (`NOTIFY4_RENAME_ENTRY`), directory attribute changes (`NOTIFY4_CHANGE_DIR_ATTRIBUTES`), and cookie verifier changes (`NOTIFY4_CHANGE_COOKIE_VERIFIER`) by setting

one or more corresponding bits in the `gdda_notification_types` field.

The client can also ask for notifications of changes to attributes of directory entries (`NOTIFY4_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_types` appropriately to its particular need 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. The enumeration of 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 synchronized within N seconds.

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 MUST use a security tuple ([Section 2.6.1](#)) that the directory or its applicable ancestor ([Section 2.6](#)) is exported with. If not, the server MUST return `NFS4ERR_WRONGSEC` to the operation that both precedes `GET_DIR_DELEGATION` and sets the current filehandle (see [Section 2.6.3.1](#)).

The directory delegation covers all the entries in the directory except the parent 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 parent entry points to the parent directory.

[18.40](#). Operation 47: GETDEVICEINFO - Get Device Information

[18.40.1](#). ARGUMENT

```
struct GETDEVICEINFO4args {
    deviceid4      gdia_device_id;
    layouttype4    gdia_layout_type;
    count4         gdia_maxcount;
    bitmap4        gdia_notify_types;
};
```

[18.40.2](#). RESULT

```
struct GETDEVICEINFO4resok {
    device_addr4    gdir_device_addr;
    bitmap4         gdir_notification;
};

union GETDEVICEINFO4res switch (nfsstat4 gdir_status) {
case NFS4_OK:
    GETDEVICEINFO4resok    gdir_resok4;
```

```

case NFS4ERR_TOOSMALL:
    count4                                gdir_mincount;
default:
    void;
};

```

[18.40.3](#). DESCRIPTION

Returns pNFS storage device address information for the specified device ID. The client identifies the device information to be returned by providing the `gdia_device_id` and `gdia_layout_type` that uniquely identify the device. The client provides `gdia_maxcount` to limit the number of bytes for the result. This maximum size represents all of the data being returned within the `GETDEVICEINFO4resok` structure and includes the XDR overhead. The server may return less data. If the server is unable to return any information within the `gdia_maxcount` limit, the error `NFS4ERR_TOOSMALL` will be returned. However, if `gdia_maxcount` is zero, `NFS4ERR_TOOSMALL` MUST NOT be returned.

The `da_layout_type` field of the `gdir_device_addr` returned by the server MUST be equal to the `gdia_layout_type` specified by the client. If it is not equal, the client SHOULD ignore the response as invalid and behave as if the server returned an error, even if the client does have support for the layout type returned.

The client also provides a notification bitmap, `gdia_notify_types` for the device ID mapping notification for which it is interested in receiving; the server must support device ID notifications for the notification request to have affect. The notification mask is composed in the same manner as the bitmap for file attributes ([Section 3.3.7](#)). The numbers of bit positions are listed in the `notify_device_type4` enumeration type ([Section 20.12](#)). Only two enumerated values of `notify_device_type4` currently apply to `GETDEVICEINFO`: `NOTIFY_DEVICEID4_CHANGE` and `NOTIFY_DEVICEID4_DELETE` (see [Section 20.12](#)).

The notification bitmap applies only to the specified device ID. If a client issues `GETDEVICEINFO` on a deviceID multiple times, the last notification bitmap is used by the server for subsequent notifications. If the bitmap is zero or empty, then the device ID's

notifications are turned off.

If the client wants to just update or turn off notifications, it MAY issue GETDEVICEINFO with `gdia_maxcount` set to zero. In that event, if the device ID is valid, the reply's `da_addr_body` field of the `gdir_device_addr` field will be of zero length.

If an unknown device ID is given in `gdia_device_id`, the server returns `NFS4ERR_NOENT`. Otherwise, the device address information is returned in `gdir_device_addr`. Finally, if the server supports notifications for device ID mappings, the `gdir_notification` result will contain a bitmap of which notifications it will actually send to the client (via `CB_NOTIFY_DEVICEID`, see [Section 20.12](#)).

If `NFS4ERR_TOOSMALL` is returned, the results also contain `gdir_mincount`. The value of `gdir_mincount` represents the minimum size necessary to obtain the device information.

[18.40.4](#). IMPLEMENTATION

Aside from updating or turning off notifications, another use case for `gdia_maxcount` being set to zero is to validate a device ID.

The client SHOULD request a notification for changes or deletion of a device ID to device address mapping so that the server can allow the client gracefully use a new mapping, without having pending I/O fail abruptly, or force layouts using the device ID to be recalled or

revoked.

It is possible that GETDEVICEINFO (and GETDEVICELIST) will race with `CB_NOTIFY_DEVICEID`, i.e. `CB_NOTIFY_DEVICEID` arrives before the client gets and processes the response to GETDEVICEINFO or GETDEVICELIST. The analysis of the race leverages the fact that the server MUST NOT delete a device ID that is referred to by a layout the client has.

- o `CB_NOTIFY_DEVICEID` deletes a device ID. If the client believes it has layouts that refer to the device ID, then it is possible the layouts have been revoked. The client should send a `TEST_STATEID` request using the `stateid` for each layout that might have been revoked. If `TEST_STATEID` indicates any layouts have been revoked,

the client must recover from layout revocation as described in [Section 12.5.6](#). If TEST_STATEID indicates at least one layout has not been revoked, the client should send a GETDEVICEINFO on the device ID to verify that the device ID has been deleted. If GETDEVICEINFO indicates the device ID does not exist, the client then assumes the server is faulty, and recovers issuing by EXCHANGE_ID. If the client does not have layouts that refer to the device ID, no harm is done. The client should mark the device ID as deleted, and when the GETDEVICEINFO or GETDEVICELIST results are finally received for the device ID, delete the device ID from client's cache.

- o CB_NOTIFY_DEVICEID indicates a device ID's device addressing mappings have changed. The client should assume that the results from the in progress GETDEVICEINFO will be stale for the device ID once received, and so it should send another GETDEVICEINFO on the device ID.

[18.41](#). Operation 48: GETDEVICELIST - Get All Device Mappings for a File System

[18.41.1](#). ARGUMENT

```
struct GETDEVICELIST4args {  
    /* CURRENT_FH: object belonging to the file system */  
    layouttype4      gdla_layout_type;  
  
    /* number of deviceIDs to return */  
    count4           gdla_maxdevices;  
  
    nfs_cookie4      gdla_cookie;  
    verifier4        gdla_cookieverf;  
};
```

[18.41.2](#). RESULT

```
struct GETDEVICELIST4resok {  
    nfs_cookie4      gdlr_cookie;  
    verifier4        gdlr_cookieverf;  
    deviceid4        gdlr_deviceid_list<>;  
    bool              gdlr_eof;
```

```

};

union GETDEVICELIST4res switch (nfsstat4 gdlr_status) {
case NFS4_OK:
    GETDEVICELIST4resok      gdlr_resok4;
default:
    void;
};

```

[18.41.3.](#) DESCRIPTION

This operation is used by the client to enumerate all of the device IDs a server's file system uses.

The client provides a current filehandle of a file object that belongs to the file system (i.e. all file objects sharing the same fsid as that of the current filehandle), and the layout type in `gdlr_layout_type`. Since this operation might require multiple calls to enumerate all the device IDs (and is thus similar to the `READDIR` ([Section 18.23](#)) operation), the client also provides `gdlr_cookie` and `gdlr_cookieverf` to specify the current cursor position in the list. When the client wants to read from the beginning of the file system's device mappings, it sets `gdlr_cookie` to zero. The field `gdlr_cookieverf` MUST be ignored by the server when `gdlr_cookie` is zero. The client provides `gdlr_maxdevices` to limit the number of device IDs in the result. If `gdlr_maxdevices` is zero, the server MUST return `NFS4ERR_INVALID`. The server MAY return fewer device IDs.

The successful response to the operation will contain the cookie, `gdlr_cookie`, and cookie verifier, `gdlr_cookieverf`, to be used on the subsequent `GETDEVICELIST`. A `gdlr_eof` value of `TRUE` signifies that there are no remaining entries in the server's device list. Each element of `gdlr_deviceid_list` contains a device ID.

[18.41.4.](#) IMPLEMENTATION

An example of the use of this operation is for pNFS clients and servers that use `LAYOUT4_BLOCK_VOLUME` layouts. In these environments it may be helpful for a client to determine device accessibility upon first file system access.

[18.42.](#) Operation 49: LAYOUTCOMMIT - Commit Writes Made Using a Layout

[18.42.1.](#) 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;
    stateid4         loca_stateid;
    newoffset4       loca_last_write_offset;
    newtime4         loca_time_modify;
    layoutupdate4    loca_layoutupdate;
};
```

Internet-Draft

NFSv4.1

September 2008

[18.42.2](#). 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;
};
```

[18.42.3](#). DESCRIPTION

Commits changes in the layout represented by the current filehandle, client ID (derived from the session ID in the preceding SEQUENCE operation), byte range, and stateid. Since layouts are sub-dividable, a smaller portion of a layout, retrieved via LAYOUTGET, can be committed. The region being committed is specified through the byte range (loca_offset and loca_length). This region MUST overlap with one or more existing layouts previously granted via LAYOUTGET ([Section 18.43](#)), each with an iomode of LAYOUTIOMODE4_RW. In the case where the iomode of any held layout segment is not LAYOUTIOMODE4_RW, the server should return the error NFS4ERR_BAD_IOMODE. For the case where the client does not hold matching layout segment(s) for the defined region, the server should return the error NFS4ERR_BAD_LAYOUT.

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 referenced by LAYOUTCOMMIT is still valid after the operation

completes and can be continued to be referenced by the client ID, filehandle, byte range, layout type, and stateid.

If the `loca_reclaim` field is set to `TRUE`, this indicates that the client is attempting to commit changes to a layout after the restart

of the metadata server during the metadata server's recovery grace period (see [Section 12.7.4](#)). 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 restart 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 restart 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 for the file. It simply commits the changes to the layout specified in the `loca_layoutupdate` field. To obtain a layout for the file the client must send 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 restart. If the client is committing a layout that was acquired during the metadata server's grace period, it **MUST** set the "reclaim" field to `FALSE`.

The `loca_stateid` is a layout stateid value as returned by previously successful layout operations (see [Section 12.5.3](#)).

The `loca_last_write_offset` field specifies the offset of the last byte written by the client previous to the `LAYOUTCOMMIT`. Note that this value is never equal to the file's size (at most it is one byte less than the file's size) and **MUST** be less than or equal to `NFS4_MAXFILEOFF`. Also, `loca_last_write_offset` **MUST** overlap the range described by `loca_offset` and `loca_length`. 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 field allows the client to suggest a modification time it would like the metadata server to set. The metadata server may use the suggestion or it may use the time of the LAYOUTCOMMIT operation to set the modification time. If the metadata server uses the client provided modification time, 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 12.5.4](#) for more details. If the client desires the resultant modification time 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's modification time. For block/volume layouts, it needs to specify precisely which blocks have been used.

If the layout identified in the arguments does not exist, the error NFS4ERR_BADLAYOUT is returned. The layout 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 and the current stateid retains its value.

The client MAY 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 restart. 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.

[18.43](#). Operation 50: LAYOUTGET – Get Layout Information

[18.43.1](#). 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;
    stateid4            loga_stateid;
    count4              loga_maxcount;
};
```

[18.43.2](#). RESULT

```
struct LAYOUTGET4resok {
    bool                logr_return_on_close;
    stateid4            logr_stateid;
    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;
};

```

[18.43.3](#). DESCRIPTION

Requests a layout from the metadata server for reading or writing the file given by the filehandle at the byte range specified by offset and length. Layouts are identified by the client ID (derived from the session ID in the preceding SEQUENCE operation), current filehandle, layout type (`loga_layout_type`), and the layout stateid (`loga_stateid`). The use of the `loga_iomode` field 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 layouts and device ID to device address mappings, then it MUST return NFS4ERR_GRACE (see [Section 8.4.2.1](#)).

The LAYOUTGET operation returns layout information for the specified byte range: a layout. The client actually specifies two ranges, both

starting at the offset in the `loga_offset` field. The first range is between `loga_offset` and `loga_offset + loga_length - 1` inclusive. This range indicates the desired range the client wants the layout to cover. The second range is between `loga_offset` and `loga_offset + loga_minlength - 1` inclusive. This range indicates the required range the client needs the layout to cover. Thus, `loga_minlength` MUST be less than or equal to `loga_length`.

When a length field is set to NFS4_UINT64_MAX, this indicates a desire (when `loga_length` is NFS4_UINT64_MAX) or requirement (when `loga_minlength` is NFS4_UINT64_MAX) to get a layout from `loga_offset` through the end-of-file, regardless of the file's length.

The following rules govern the relationships among, and the minima of `loga_length`, `loga_minlength`, and `loga_offset`.

- o If `loga_length` is less than `loga_minlength`, the metadata server MUST return `NFS4ERR_INVALID`.
- o If `loga_minlength` is zero, this is an indication to the metadata server that the client desires any layout at offset `loga_offset` or less that the metadata server has "readily available". Readily is subjective, and depends on the layout type and the pNFS server implementation. For example, some metadata servers might have to pre-allocate stable storage when they receive a request for a range of a file that goes beyond the file's current length. If `loga_minlength` is zero and `loga_length` is greater than zero, this tells the metadata server what range of the layout the client would prefer to have. If `loga_length` and `loga_minlength` are both zero, then the client is indicating it desires a layout of any length with the ending offset of the range no less than specified `loga_offset`, and the starting offset at or below `loga_offset`. If the metadata server does not have a layout that is readily available, then it MUST return `NFS4ERR_LAYOUTTRYLATER`.
- o If the sum of `loga_offset` and `loga_minlength` exceeds `NFS4_UINT64_MAX`, and `loga_minlength` is not `NFS4_UINT64_MAX`, the error `NFS4ERR_INVALID` MUST result.
- o If the sum of `loga_offset` and `loga_length` exceeds `NFS4_UINT64_MAX`, and `loga_length` is not `NFS4_UINT64_MAX`, the error `NFS4ERR_INVALID` MUST result.

After the metadata server has performed the above checks on `loga_offset`, `loga_minlength`, and `loga_offset`, the metadata server MUST return a layout according to the rules in Table 12.

Acceptable layouts based on `loga_minlength`. Note: `u64m` = `NFS4_UINT64_MAX`; `a_off` = `loga_offset`; `a_minlen` = `loga_minlength`.

Layout iomode of request	Layout a_minlen of request	Layout iomode of reply	Layout offset of reply	Layout length of reply

_READ	u64m	MAY be _READ	MUST be ≤ a_off	MUST be ≥ file length - layout offset
_READ	u64m	MAY be _RW	MUST be ≤ a_off	MUST be u64m
_READ	> 0 and < u64m	MAY be _READ	MUST be ≤ a_off	MUST be ≥ MIN(file length, a_minlen + a_off) - layout offset
_READ	> 0 and < u64m	MAY be _RW	MUST be ≤ a_off	MUST be ≥ a_off - layout offset + a_minlen
_READ	0	MAY be _READ	MUST be ≤ a_off	MUST be > 0
_READ	0	MAY be _RW	MUST be ≤ a_off	MUST be > 0
_RW	u64m	MUST be _RW	MUST be ≤ a_off	MUST be u64m
_RW	> 0 and < u64m	MUST be _RW	MUST be ≤ a_off	MUST be ≥ a_off - layout offset + a_minlen
_RW	0	MUST be _RW	MUST be ≤ a_off	MUST be > 0

Table 12

If loga_minlength is not zero and the metadata server cannot return a layout according to the rules in Table 12, then the metadata server MUST return the error NFS4ERR_BADLAYOUT. If loga_minlength is zero and the metadata server cannot or will not return a layout according to the rules in Table 12, then the metadata server MUST return the error NFS4ERR_LAYOUTTRYLATER. Assuming loga_length is greater than loga_minlength or equal to zero, the metadata server SHOULD return a layout according to the rules in Table 13.

Desired layouts based on loga_length. The rules of Table 12 MUST be

applied first. Note: `u64m = NFS4_UINT64_MAX`; `a_off = loga_offset`;
`a_len = loga_length`.

Layout iomode of request	Layout a_len of request	Layout iomode of reply	Layout offset of reply	Layout length of reply
<code>_READ</code>	<code>u64m</code>	MAY be <code>_READ</code>	MUST be <code><= a_off</code>	SHOULD be <code>u64m</code>
<code>_READ</code>	<code>u64m</code>	MAY be <code>_RW</code>	MUST be <code><= a_off</code>	SHOULD be <code>u64m</code>
<code>_READ</code>	<code>> 0 and < u64m</code>	MAY be <code>_READ</code>	MUST be <code><= a_off</code>	SHOULD be <code>>=</code> <code>a_off - layout</code> <code>offset + a_len</code>
<code>_READ</code>	<code>> 0 and < u64m</code>	MAY be <code>_RW</code>	MUST be <code><= a_off</code>	SHOULD be <code>>=</code> <code>a_off - layout</code> <code>offset + a_len</code>
<code>_READ</code>	<code>0</code>	MAY be <code>_READ</code>	MUST be <code><= a_off</code>	SHOULD be <code>></code> <code>a_off - layout</code> <code>offset</code>
<code>_READ</code>	<code>0</code>	MAY be <code>_READ</code>	MUST be <code><= a_off</code>	SHOULD be <code>></code> <code>a_off - layout</code> <code>offset</code>
<code>_RW</code>	<code>u64m</code>	MUST be <code>_RW</code>	MUST be <code><= a_off</code>	SHOULD be <code>u64m</code>
<code>_RW</code>	<code>> 0 and < u64m</code>	MUST be <code>_RW</code>	MUST be <code><= a_off</code>	SHOULD be <code>>=</code> <code>a_off - layout</code> <code>offset + a_len</code>
<code>_RW</code>	<code>0</code>	MUST be <code>_RW</code>	MUST be <code><= a_off</code>	SHOULD be <code>></code> <code>a_off - layout</code> <code>offset</code>

Table 13

The `loga_stateid` field specifies a valid stateid. If a layout is not currently held by the client, the `loga_stateid` field represents a stateid reflecting the correspondingly valid open, byte-range lock, or delegation stateid. Once a layout is held on the file by the client, the `loga_stateid` field MUST be a stateid as returned from a previous `LAYOUTGET` or `LAYOUTRETURN` operation or provided by a `CB_LAYOUTRECALL` operation (see [Section 12.5.3](#)).

The `loga_maxcount` field specifies the maximum layout size (in bytes) 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.

The returned layout is expressed as an array, `logr_layout`, with each element of type `layout4`. If a file has a single striping pattern, then `logr_layout` SHOULD contain just one entry. Otherwise, if the requested range overlaps more than one striping pattern, `logr_layout` will contain the required number of entries. The elements of `logr_layout` MUST be sorted in ascending order of the value of the `lo_offset` field of each element. There MUST be no gaps or overlaps in the range between two successive elements of `logr_layout`. The `lo_iomode` field in each element of `logr_layout` MUST be the same.

Table 12 and Table 13 both refer to a returned layout `iomode`, `offset`, and `length`. Because the returned layout is encoded in the `logr_layout` array, more description is required.

`iomode`

The value of the returned layout `iomode` listed in Table 12 and Table 13 is equal to the value of the `lo_iomode` field in each element of `logr_layout`. As shown in Table 12 and Table 13, the metadata server MAY return a layout with an `lo_iomode` different from the requested `iomode` (field `loga_iomode` of the request). If it does so, it MUST ensure that the `lo_iomode` is more permissive than the `loga_iomode` requested. For example, this behavior 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.

`offset`

The value of the returned layout `offset` listed in Table 12 and Table 13 is always equal to the `lo_offset` field of the first element `logr_layout`.

`length`

When setting the value of the returned layout `length`, the situation is complicated by the possibility that the special layout `length` value `NFS4_UINT64_MAX` is involved. For a `logr_layout` array of `N` elements, the `lo_length` field in the first `N-1` elements MUST NOT be `NFS4_UINT64_MAX`. The `lo_length` field of the last element of `logr_layout` can be `NFS4_UINT64_MAX` under some conditions as described in the following list.

- * If an applicable rule of Table 12 states the metadata server MUST return a layout of length NFS4_UINT64_MAX, then lo_length

field of the last element of log_layout MUST be NFS4_UINT64_MAX.

- * If an applicable rule of Table 12 states the metadata server MUST NOT return a layout of length NFS4_UINT64_MAX, then lo_length field of the last element of log_layout MUST NOT be NFS4_UINT64_MAX.
- * If an applicable rule of Table 13 states the metadata server SHOULD return a layout of length NFS4_UINT64_MAX, then lo_length field of the last element of log_layout SHOULD be NFS4_UINT64_MAX.
- * When the value of the returned layout length of Table 12 and Table 13 is not NFS4_UINT64_MAX, then the returned layout length is equal to the sum of the lo_length fields of each element of log_layout.

The log_return_on_close result field is a directive to return the layout before closing the file. When the metadata 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 metadata 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 remove one extra step from the client's and metadata server's interaction.

The log_stateid stateid is returned to the client for use in subsequent layout related operations. See [Section 8.2](#), [Section 12.5.3](#), and [Section 12.5.5.2](#) for a further discussion and requirements.

The format of the returned layout (lo_content) is specific to the layout type. The value of the layout type (lo_content.loc_type) for each of the elements of the array of layouts returned by the metadata server (log_layout) MUST be equal to the log_layout_type specified by the client. If it is not equal, the client SHOULD ignore the response as invalid and behave as if the metadata server returned an error, even if the client does have support for the layout type

returned.

If layouts are not supported for the requested file or its containing file system the metadata server MUST return NFS4ERR_LAYOUTUNAVAILABLE. If the layout type is not supported, the metadata server MUST return NFS4ERR_UNKNOWN_LAYOUTTYPE. If layouts are supported but no layout matches the client provided layout identification, the metadata server MUST return NFS4ERR_BADLAYOUT. If an invalid loga_iomode is specified, or a loga_iomode of

LAYOUTIOMODE4_ANY is specified, the metadata server MUST return NFS4ERR_BADIOMODE.

If the layout for the file is unavailable due to transient conditions, e.g. file sharing prohibits layouts, the metadata server MUST return NFS4ERR_LAYOUTTRYLATER.

If the layout request is rejected due to an overlapping layout recall, the metadata server MUST return NFS4ERR_RECALLCONFLICT. See [Section 12.5.5.2](#) for details.

If the layout conflicts with a mandatory byte 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 metadata 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 and the current stateid is updated to match the value as returned in the results.

[18.43.4](#). IMPLEMENTATION

Typically, LAYOUTGET will be called as part of a COMPOUND request after an OPEN operation and results in the client having location information for the file; this requires that loga_stateid be set to

the special stateid that tells the metadata server to use the current stateid, which is set by OPEN (see [Section 16.2.3.1.2](#)) . A client may also hold a layout across multiple OPENS. The client specifies a layout type that limits what kind of layout the metadata server will return. This prevents metadata servers from granting layouts that are unusable by the client.

As indicated by Table 12 and Table 13 the specification of LAYOUTGET allows a pNFS client and server considerable flexibility. A pNFS client can take several strategies for sending LAYOUTGET. Some examples are as follows.

- o If LAYOUTGET is preceded by OPEN in the same COMPOUND request, and the OPEN requests read access, the client might opt to request a _READ layout with loga_offset set to zero, loga_minlength set to zero, and loga_length set to NFS4_UINT64_MAX. If the file has

space allocated to it, that space is striped over one or more storage devices, and there is either no conflicting layout, or the concept of a conflicting layout does not apply to the pNFS server's layout type or implementation, then the metadata server might return a layout with a starting offset of zero, and a length equal to the length of the file, if not NFS4_UINT64_MAX. If the length of the file is not a multiple of the pNFS server's stripe width (see [Section 13.2](#) for a formal definition), the metadata server might round the returned layout's length up.

- o If LAYOUTGET is preceded by OPEN in the same COMPOUND request, and the OPEN does not truncate the file, and requests write access, the client might opt to request a _RW layout with loga_offset set to zero, loga_minlength set to zero, and loga_length set to the file's current length (if known), or NFS4_UINT64_MAX. As with the previous case, under some conditions the metadata server might return a layout that covers the entire length of the file or beyond.
- o As above, but the OPEN truncates the file. In this case, client might anticipate it will be writing to the file from offset zero, and so loga_offset and loga_minlength are set to zero, and loga_length is set to the value of threshold4_write_iosize. The metadata server might return a layout from offset zero with a length at least as long as threshold4_write_iosize.

- o A process on the client invokes a request to read from offset 10000 for length 50000. The client is using buffered I/O, and has buffer sizes of 4096 bytes. The client intends to map the request of the process into a series of READ requests starting at offset 8192. The end offset needs to be higher than $10000 + 50000 = 60000$, and the next offset that is a multiple of 4096 is 61440. The difference between 61440 and that starting offset of the layout is 53248 (which is the product of 4096 and 15). The value of `threshold4_read_iosize` is less than 53248, so the client sends a LAYOUTGET request with `loga_offset` set to 8192, `loga_minlength` set to 53248, and `loga_length` set to the file's length (if known) minus 8192 or `NFS4_UINT64_MAX` (if the file's length is not known). Since this LAYOUTGET request exceeds the metadata server's threshold, it grants the layout, possibly with an initial offset of 0, with an end offset of at least $8192 + 53248 - 1 = 61439$, but preferably a layout with an offset aligned on the stripe width and a length that is a multiple of the stripe width.
- o As above, but the client is not using buffered I/O, and instead all internal I/O requests are sent directly to the server. The LAYOUTGET request has `loga_offset` equal to 10000, and `loga_minlength` set to 50000. The value of `loga_length` is set to

the length of the file. The metadata server is free to return a layout that fully overlaps the requested range, with a starting offset and length aligned on the stripe width.

- o Again a process on the client invokes a request to read from offset 10000 for length 50000, and buffered I/O is in use. The client is expecting that the server might not be able to return the layout for the full I/O range, with `loga_offset` set to 8192 and `loga_minlength` set to 53248. The client intends to map the request of the process into a series of READ requests starting at offset 8192, each with length 4096, with a total length of 53248 (which equals $13 * 4096$). Because the value of `threshold4_read_iosize` is equal to 4096, it is practical and reasonable for the client to use several LAYOUTGETs to complete the series of READs. The client sends a LAYOUTGET request with `loga_offset` set to 8192, `loga_minlength` set to 4096, and `loga_length` set to 53248 or higher. The server will grant a layout possibly with an initial offset of 0, with an end offset of

at least $8192 + 4096 - 1 = 12287$, but preferably a layout with an offset aligned on the stripe width and a length that is a multiple of the stripe width. This will allow the client to make forward progress, possibly having to issue more LAYOUTGET requests for the remainder of the range.

- o An NFS client detects a sequential read pattern, and so issues a LAYOUTGET that goes well beyond any current or pending read requests to the server. The server might likewise detect this pattern, and grant the LAYOUTGET request. The client continues to send LAYOUTGET requests once it has read from an offset of the file that represents 50% of the way through the last layout it received.
- o As above but the client fails to detect the pattern, but the server does. The next time the metadata server gets a LAYOUTGET, it returns a layout with a length that is well beyond `loga_minlength`.
- o A client is using buffered I/O, and has a long queue of write behinds to process and also detects a sequential write pattern. It issues a LAYOUTGET for a layout that spans the range of the queued write behinds and well beyond, including ranges beyond the filer's current length. The client continues to issue LAYOUTGETs once the write behind queue reaches 50% of the maximum queue length.

Once the client has obtained a layout referring to a particular device ID, the metadata server MUST NOT delete the device ID until the layout is returned or revoked.

CB_NOTIFY_DEVICEID can race with LAYOUTGET. One race scenario is that LAYOUTGET returns a device ID the client does not have device address mappings for, and the metadata server sends a CB_NOTIFY_DEVICEID to add the device ID to the client's awareness and meanwhile the client sends GETDEVICEINFO on the device ID. This scenario is discussed in [Section 18.40.4](#). Another scenario is that the CB_NOTIFY_DEVICEID is processed by the client before it processes the results from LAYOUTGET. The client will send a GETDEVICEINFO on the device ID. If the results from GETDEVICEINFO are received before the client gets results from LAYOUTGET, then there is no longer a race. If the results from LAYOUTGET are received before the results

from GETDEVICEINFO, the client can either wait for results of GETDEVICEINFO, or send another one to get possibly more up to date device address mappings for the device ID.

[18.44.](#) Operation 51: LAYOUTRETURN - Release Layout Information

[18.44.1.](#) ARGUMENT

```
/* Constants used for LAYOUTRETURN and CB_LAYOUTRECALL */
const LAYOUT4_RET_REC_FILE      = 1;
const LAYOUT4_RET_REC_FSID     = 2;
const LAYOUT4_RET_REC_ALL      = 3;

enum layoutreturn_type4 {
    LAYOUTRETURN4_FILE = LAYOUT4_RET_REC_FILE,
    LAYOUTRETURN4_FSID = LAYOUT4_RET_REC_FSID,
    LAYOUTRETURN4_ALL  = LAYOUT4_RET_REC_ALL
};

struct layoutreturn_file4 {
    offset4      lrf_offset;
    length4      lrf_length;
    stateid4     lrf_stateid;
    /* layouttype4 specific data */
    opaque       lrf_body<>;
};

union layoutreturn4 switch(layoutreturn_type4 lr_returntype) {
    case LAYOUTRETURN4_FILE:
        layoutreturn_file4      lr_layout;
    default:
        void;
};
```

```
struct LAYOUTRETURN4args {
    /* CURRENT_FH: file */
    bool                lora_reclaim;
    layouttype4         lora_layout_type;
```

```

        layoutiomode4          lora_iomode;
        layoutreturn4         lora_layoutreturn;
};

```

[18.44.2.](#) RESULT

```

union layoutreturn_stateid switch (bool lrs_present) {
case TRUE:
    stateid4          lrs_stateid;
case FALSE:
    void;
};

union LAYOUTRETURN4res switch (nfsstat4 lorr_status) {
case NFS4_OK:
    layoutreturn_stateid    lorr_stateid;
default:
    void;
};

```

[18.44.3.](#) DESCRIPTION

This operation returns from the client to the server one or more layouts represented by the client ID (derived from the session ID in the preceding SEQUENCE operation), `lora_layout_type`, and `lora_iomode`. When `lr_returntype` is `LAYOUTRETURN4_FILE`, the returned layout is further identified by the current filehandle, `lrf_offset`, `lrf_length`, and `lrf_stateid`. If the `lrf_length` field is `NFS4_UINT64_MAX`, all bytes of the layout, starting at `lrf_offset` are returned. When `lr_returntype` is `LAYOUTRETURN4_FSID`, the current filehandle is used to identify the file system and all layouts matching the client ID, the `fsid` of the file system, `lora_layout_type`, and `lora_iomode` are returned. When `lr_returntype` is `LAYOUTRETURN4_ALL`, all layouts 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(s) and the associated storage protocol to access the file data.

If the set of layouts designated in the case of `LAYOUTRETURN4_FSID` or `LAYOUTRETURN4_ALL` is empty, then no error results. In the case of `LAYOUTRETURN4_FILE`, the byte range specified is returned even if it

is a subdivision of a layout previously obtained with LAYOUTGET, a combination of multiple layouts previously obtained with LAYOUTGET, or a combination including some layouts previously obtained with LAYOUTGET, and one or more subdivisions of such layouts. When the byte range does not designate any bytes for which a layout is held for the specified file, client ID, layout type and mode, no error results. See [Section 12.5.5.2.1.5](#) for considerations with "bulk" return of layouts.

The layout being returned may be a subset or superset of a layout specified by CB_LAYOUTRECALL. However, if it is a subset, the recall is not complete until the full recalled scope has been returned. Recalled scope refers to the byte range in the case of LAYOUTRETURN4_FILE, use of LAYOUTRETURN4_FSID, or the use of LAYOUTRETURN4_ALL. There must be a LAYOUTRETURN with a matching scope to complete the return even if all current layout ranges have been previously individually returned.

For all lr_returntype values, 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.

In the case that lr_returntype is LAYOUTRETURN4_FILE, the lrf_stateid provided by the client is a layout stateid as returned from previous layout operations. Note that the "seqid" field of lrf_stateid MUST NOT be zero. See [Section 8.2](#), [Section 12.5.3](#), and [Section 12.5.5.2](#) for a further discussion and requirements.

Return of a layout or all layouts does not invalidate the mapping of storage device ID to storage device address which remains in effect until specifically changed or deleted via device ID notification callbacks.

If the lora_reclaim field is set to TRUE, the client is attempting to return a layout that was acquired before the restart of the metadata server during the metadata server's grace period. When returning layouts that were acquired during the metadata server's grace period, the client 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 18.42](#)) 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 the client returns the layout in response to a CB_LAYOUTRECALL where the `lor_recalltype` field of the `clora_recall` field was `LAYOUTRECALL4_FILE`, the client should use the `lor_stateid` value from `CB_LAYOUTRECALL` as the value for `lrf_stateid`. Otherwise, it should use `logr_stateid` (from a previous `LAYOUTGET` result) or `lorr_stateid` (from a previous `LAYRETURN` result). This is done to indicate the point in time (in terms of layout stateid transitions) when the recall was sent. The client uses the precise `lora_recallstateid` value and MUST NOT set the stateid's `seqid` to zero; otherwise `NFS4ERR_BAD_STATEID` MUST be returned. `NFS4ERR_OLD_STATEID` can be returned if the client is using an old `seqid`, and the server knows the client should not be using the old `seqid`. E.g. the client uses the `seqid` on slot 1 of the session, received the response with the new `seqid`, and uses the slot to send another request with the old `seqid`.

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 12.5.5](#) for more details.

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_INVALID` is returned.

If the client sets the `lr_returntype` field to `LAYOUTRETURN4_FILE`, then the `lrs_stateid` field will represent the layout stateid as updated for this operation's processing; the current stateid will also be updated to match the returned value. If the last byte of any layout for the current file, client ID, and layout type is being returned and there are no remaining pending `CB_LAYOUTRECALL` operations for which a `LAYOUTRETURN` operation must be done, `lrs_present` MUST be `FALSE`, and no stateid will be returned. In addition, the `COMPOUND` request's current stateid will be set to all-zeroes special stateid (see [Section 16.2.3.1.2](#)). The server MUST

reject with NFS4ERR_BAD_STATEID any further use of the current stateid in that COMPOUND until the current stateid is re-established by a later stateid-returning operation.

On success, the current filehandle retains its value.

If the EXCHGID4_FLAG_BIND_PRINC_STATEID capability is set on the client ID (see [Section 18.35](#)), the server will require that the

principal, security flavor, and if applicable, the GSS mechanism, combination that acquired the layout also be the one to send LAYOUTRETURN. This might not be possible if credentials for the principal are no longer available. The server will allow the machine credential or SSV credential (see [Section 18.35](#)) to send LAYOUTRETURN if LAYOUTRETURN's operation code was set in the spo_must_allow result of EXCHANGE_ID.

[18.44.4](#). 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.

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 12.5.5.1](#) for more details.

Once the client has returned all layouts referring to a particular device ID, the server MAY delete the device ID.

[18.45](#). Operation 52: SECINFO_NO_NAME - Get Security on Unnamed Object

[18.45.1](#). ARGUMENT

```
enum secinfo_style4 {
    SECINFO_STYLE4_CURRENT_FH      = 0,
    SECINFO_STYLE4_PARENT          = 1
};

/* CURRENT_FH: object or child directory */
typedef secinfo_style4 SECINFO_NO_NAME4args;
```

[18.45.2.](#) RESULT

```
/* CURRENTFH: consumed if status is NFS4_OK */
typedef SECINFO4res SECINFO_NO_NAME4res;
```

Shepler, et al.

Expires March 9, 2009

[Page 540]

Internet-Draft

NFSv4.1

September 2008

[18.45.3.](#) 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 filehandle'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.

If PUTFH, PUTPUBFH, PUTROOTFH, or RESTOREFH return NFS4ERR_WRONGSEC, then the client resolves the situation by sending a COMPOUND request that consists of PUTFH, PUTPUBFH, or PUTROOTFH immediately followed by SECINFO_NO_NAME, style SECINFO_STYLE4_CURRENT_FH. See [Section 2.6](#) for instructions on dealing with NFS4ERR_WRONGSEC error returns from PUTFH, PUTROOTFH, PUTPUBFH, or RESTOREFH.

If SECINFO_STYLE4_PARENT is specified and there is no parent directory, SECINFO_NO_NAME MUST return NFS4ERR_NOENT.

On success, the current filehandle is consumed (see [Section 2.6.3.1.1.8](#)), and if the next operation after SECINFO_NO_NAME tries to use the current filehandle, that operation will fail with the status NFS4ERR_NOFILEHANDLE.

Everything else about SECINFO_NO_NAME is the same as SECINFO. See the discussion on SECINFO ([Section 18.29.3](#)).

[18.45.4](#). IMPLEMENTATION

See the discussion on SECINFO ([Section 18.29.4](#)).

[18.46](#). Operation 53: SEQUENCE – Supply Per-Procedure Sequencing and Control

[18.46.1](#). ARGUMENT

```
struct SEQUENCE4args {
    sessionid4    sa_sessionid;
    sequenceid4   sa_sequenceid;
    slotid4       sa_slotid;
    slotid4       sa_highest_slotid;
    bool          sa_cachethis;
};
```

[18.46.2](#). 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;
const SEQ4_STATUS_CB_PATH_DOWN_SESSION          = 0x00000200;
const SEQ4_STATUS_BACKCHANNEL_FAULT              = 0x00000400;
const SEQ4_STATUS_DEVID_CHANGED                  = 0x00000800;
const SEQ4_STATUS_DEVID_DELETED                  = 0x00001000;

```

```

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;
};

```

[18.46.3](#). DESCRIPTION

The SEQUENCE operation is used by the server to implement session request control and the reply cache semantics.

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, MUST NOT appear as the first operation in a COMPOUND. Such operations MUST yield 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 associated with the session via CREATE_SESSION or BIND_CONN_TO_SESSION, and connection association enforcement is enabled (see [Section 18.35](#)), then the server returns NFS4ERR_CONN_NOT_BOUND_TO_SESSION.

The sa_sessionid argument identifies the session this request applies to. The sr_sessionid result MUST equal sa_sessionid.

The sa_slotid argument is the index in the reply cache for the request. The sa_sequenceid field is the sequence number of the request for the reply cache entry (slot). The sr_slotid result MUST equal sa_slotid. The sr_sequenceid result MUST equal sa_sequenceid.

The sa_highest_slotid argument is the highest slot ID the client has a request outstanding for; it could be equal to sa_slotid. The server returns two "highest_slotid" values: sr_highest_slotid, and sr_target_highest_slotid. The former is the highest slot ID the server will accept in future SEQUENCE operation, and SHOULD NOT be less than the value of sa_highest_slotid. (but see [Section 2.10.6.1](#) for an exception). The latter is the highest slot ID the server would prefer the client use on a future SEQUENCE operation.

If sa_cachethis is TRUE, then the client is requesting that the server cache the entire reply in the server's reply cache; therefore the server MUST cache the reply (see [Section 2.10.6.1.3](#)). The server MAY cache the reply if sa_cachethis is FALSE. If the server does not cache the entire reply, it MUST still record that it executed the request at the specified slot and sequence ID.

The response to the SEQUENCE operation contains a word of status flags (sr_status_flags) 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 MAY be reset when lock state is lost due to a server restart (even if the session

is persistent across restarts; session persistence does not imply lock state persistence) or the establishment of a new client instance.

SEQ4_STATUS_CB_PATH_DOWN

When set, indicates that the client has no operational backchannel path for any session associated with the client ID, making it necessary for the client to re-establish one. This bit remains set on all SEQUENCE responses on all sessions associated with the client ID until at least one backchannel is available on any session associated with the client ID. If the client fails to re-establish a backchannel for the client ID, it is subject to having recallable state revoked.

SEQ4_STATUS_CB_PATH_DOWN_SESSION

When set, indicates that the session has no operational backchannel. There are two reasons why SEQ4_STATUS_CB_PATH_DOWN_SESSION may be set and not SEQ4_STATUS_CB_PATH_DOWN. First is that a callback operation that applies specifically to the session (e.g. CB_RECALL_SLOT, see [Section 20.8](#)) needs to be sent. Second is that the server did send a callback operation, but the connection was lost before the reply. The server cannot be sure whether the client received the callback operation or not, and so, per rules on request retry, the server MUST retry the callback operation over the same session. The SEQ4_STATUS_CB_PATH_DOWN_SESSION bit is the indication to the client that it needs to associate a connection to the session's backchannel. This bit remains set on all SEQUENCE responses on the session until a backchannel on the session the path is available. If the client fails to re-establish a backchannel for the session, it is subject to having recallable state revoked.

SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRING

When set, indicates that all GSS contexts assigned to the session's backchannel will expire within a period equal to the lease time. This bit remains set on all SEQUENCE replies until the expiration time of at least one context is beyond the lease period from the current time (relative to the time of when a SEQUENCE response was sent) or until all GSS contexts for the session's backchannel have expired.

SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRED

When set, indicates all GSS contexts assigned to the session's backchannel have expired. This bit remains set on all SEQUENCE replies until at least one non-expired context for the session's backchannel has been established.

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 on all SEQUENCE replies until the loss of all such locks has been acknowledged by use of FREE_STATEID (see [Section 18.38](#)), or by establishing a new client instance by destroying all sessions (via DESTROY_SESSION), the client ID (via DESTROY_CLIENTID), 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 on all SEQUENCE replies until the loss of all such locks has been acknowledged by use of FREE_STATEID.

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 on all SEQUENCE replies until the loss of all such locks has been acknowledged by use of FREE_STATEID.

SEQ4_STATUS_RECALLABLE_STATE_REVOKED

When set indicates that one or more recallable objects have been revoked without expiration of the lease period, due to the client's failure to return them when recalled which may be a consequence of there being no working backchannel and the client failing to reestablish a backchannel per the SEQ4_STATUS_CB_PATH_DOWN, SEQ4_STATUS_CB_PATH_DOWN_SESSION, or SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRED status flags. This status bit remains set on all SEQUENCE replies until the loss of all such locks has been acknowledged by use of FREE_STATEID.

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 NFS4ERR_MOVED 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. See [Section 11.7.7.1](#).

SEQ4_STATUS_RESTART_RECLAIM_NEEDED

When set indicates that due to server restart the client must reclaim locking state. Until the client sends a global RECLAIM_COMPLETE ([Section 18.51](#)), every SEQUENCE operation will

Internet-Draft

NFSv4.1

September 2008

return SEQ4_STATUS_RESTART_RECLAIM_NEEDED.

SEQ4_STATUS_BACKCHANNEL_FAULT

The server has encountered an unrecoverable fault with the backchannel (e.g. it has lost track of the sequence ID for a slot in the backchannel). The client MUST stop sending more requests on the session's fore channel, wait for all outstanding requests to complete on the fore and back channel, and then destroy the session.

SEQ4_STATUS_DEVID_CHANGED

The client is using device ID notifications and the server has changed a device ID mapping held by the client. This flag will stay present until the client has obtained the new mapping with GETDEVICEINFO.

SEQ4_STATUS_DEVID_DELETED

The client is using device ID notifications and the server has deleted a device ID mapping held by the client. This flag will stay in effect until the client sends a GETDEVICEINFO on the device ID with a null value in the argument `gdia_notify_types`.

The value of the `sa_sequenceid` argument relative to the cached sequence ID on the slot falls into one of three cases.

- o If the difference between `sa_sequenceid` and the server's cached sequence ID at the slot ID is two (2) or more, or if `sa_sequenceid` is less than the cached sequence ID (accounting for wraparound of the unsigned sequence ID value), then the server MUST return `NFS4ERR_SEQ_MISORDERED`.
- o If `sa_sequenceid` and the cached sequence ID are the same, this is a retry, and the server replies with the COMPOUND reply that is stored the reply cache. The lease is possibly renewed as described below.
- o If `sa_sequenceid` is one greater (accounting for wraparound) than the cached sequence ID, then this is a new request, and the slot's sequence ID 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 possibly renew

the lease.

If the client reuses a slot ID and sequence ID for a completely different request, the server MAY treat the request as if it is a retry of what it has already executed. The server MAY however detect the client's illegal reuse and return NFS4ERR_SEQ_FALSE_RETRY.

If SEQUENCE returns an error, then the state of the slot (sequence ID, cached reply) MUST NOT change, and the associated lease MUST NOT be renewed.

If SEQUENCE returns NFS4_OK, then the associated lease MUST be renewed (see [Section 8.3](#)), except if SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED is returned in sr_status_flags.

[18.46.4](#). IMPLEMENTATION

The server MUST maintain a mapping of session ID 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 a SEQUENCE done after a server restart may encounter requests performed and recorded in a persistent reply cache before the server restart. In this case, SEQUENCE will be processed successfully, while requests which were not processed previously are rejected with NFS4ERR_DEADSESSION.

Depending on which of the operations within the COMPOUND were successfully performed before the server restart, these operations will also have replies sent from the server reply cache. Note that when these operations establish locking state it is locking state that applies to the previous server instance and to the previous client ID, even though the server restart, which logically happened after these operations, eliminated that state. In the case of a partially executed COMPOUND, processing may reach an operation not processed during the earlier server instance, making this operation a new one and not performable on the existing session. In this case, NFS4ERR_DEADSESSION will be returned from that operation.

[18.47](#). Operation 54: SET_SSV - Update SSV for a Client ID

[18.47.1](#). ARGUMENT

```

struct ssa_digest_input4 {
    SEQUENCE4args sdi_seqargs;
};

struct SET_SSV4args {
    opaque          ssa_ssv<>;
    opaque          ssa_digest<>;
};

```

[18.47.2.](#) 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;
};

```

[18.47.3.](#) DESCRIPTION

This operation is used to update the SSV for a client ID. Before SET_SSV is called the first time on a client ID, the SSV is zero (0). The SSV is the key used for the SSV GSS mechanism ([Section 2.10.9](#))

SET_SSV MUST be preceded by a SEQUENCE operation in the same COMPOUND. It MUST NOT be used if the client did not opt for SP4_SSV state protection when the client ID was created (see [Section 18.35](#)); the server returns NFS4ERR_INVALID in that case.

The field `ssa_digest` is computed as the output of the HMAC [RFC2104 \[11\]](#) using the subkey derived from the `SSV4_SUBKEY_MIC_I2T` and current SSV as the key (See [Section 2.10.9](#) for a description of subkeys), and an XDR encoded value of data type `ssa_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 argument `ssa_ssv` is XORed with the current SSV to produce the new SSV. The argument `ssa_ssv` SHOULD be generated randomly.

In the response, `ssr_digest` is the output of the HMAC using the subkey derived from `SSV4_SUBKEY_MIC_T2I` and 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.

As noted in [Section 18.35](#), the client and server can maintain multiple concurrent versions of the SSV. The client and server each MUST maintain an internal SSV version number, which is set to one (1)

the first time `SET_SSV` executes on the server and the client receives the first `SET_SSV` reply. Each subsequent `SET_SSV` increases the internal SSV version number by one (1). The value of this version number corresponds to the `smpt_ssv_seq`, `smt_ssv_seq`, `sspt_ssv_seq`, and `ssct_ssv_seq` fields of the SSV GSS mechanism tokens (see [Section 2.10.9](#)).

[18.47.4](#). 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`. This error might be the result of another `SET_SSV` from the same client ID changing the SSV. If so, the client recovers by issuing `SET_SSV` again with a recomputed digest based on the subkey of the new SSV. If the transport connection is dropped after the `SET_SSV` request is sent, but before the `SET_SSV` reply is received, then there are special considerations for recovery if the client has no more connections associated with sessions associated with the client ID of the SSV. See [Section 18.34.4](#).

Clients SHOULD NOT send an ssa_ssv that is equal to a previous ssa_ssv, nor equal to a previous or current SSV (including an ssa_ssv equal to zero since the SSV is initialized to zero when the client ID is created).

Clients SHOULD send SET_SSV with RPCSEC_GSS privacy. Servers MUST support RPCSEC_GSS with privacy for any COMPOUND that has { SEQUENCE, SET_SSV }.

A client SHOULD NOT send SET_SSV with the SSV GSS mechanism's credential because the purpose of SET_SSV is to seed the SSV from non-SSV credentials. Instead SET_SSV SHOULD be sent with the credential of a user that is accessing the client ID for the first time ([Section 2.10.8.3](#)). However if the client does send SET_SSV with SSV credentials, the digest protecting the arguments uses the value of the SSV before ssa_ssv is XORed in, and the digest protecting the results uses the value of the SSV after the ssa_ssv is XORed in.

[18.48](#). Operation 55: TEST_STATEID - Test Stateids for Validity

[18.48.1](#). ARGUMENT

```
struct TEST_STATEID4args {  
    stateid4          ts_stateids<>;  
};
```

[18.48.2](#). RESULT

```
struct TEST_STATEID4resok {  
    nfsstat4          tsr_status_codes<>;  
};  
  
union TEST_STATEID4res switch (nfsstat4 tsr_status) {  
    case NFS4_OK:  
        TEST_STATEID4resok tsr_resok4;  
    default:  
        void;  
};
```

18.48.3. DESCRIPTION

The TEST_STATEID operation is used to check the validity of a set of stateids. It can be used at any time but the client should definitely use it when it receives an indication that one or more of its stateids have been invalidated due to lock revocation. This occurs when the SEQUENCE operation returns with one of the following sr_status_flags set:

- o SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED
- o SEQ4_STATUS_EXPIRED_ADMIN_STATE_REVOKED
- o SEQ4_STATUS_EXPIRED_RECALLABLE_STATE_REVOKED

The client can use TEST_STATEID one or more times to test the validity of its stateids. Each use of 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 returns the status code that would be returned if that stateid were to be used in normal operation. Returning such a status indication is not an error and does not cause compound processing to terminate. Checks for the validity of the stateid proceed as they would for normal operations with a number of exceptions:

- o There is no check for the type of stateid object, as would be the case for normal use of a stateid.
- o There is no reference to the current filehandle.

- o Special stateids are always considered invalid (they result in the error code NFS4ERR_BAD_STATEID).

All stateids are interpreted as being associated with the client for the current session. Any possible association with a previous instance of the client (as stale stateids) is not considered.

The errors which are validly returned within the `status_code` array are: `NFS4ERR_OK`, `NFS4ERR_BAD_STATEID`, `NFS4ERR_OLD_STATEID`, `NFS4ERR_EXPIRED`, `NFS4ERR_ADMIN_REVOKED`, and `NFS4ERR_DELEG_REVOKED`.

[18.48.4](#). IMPLEMENTATION

See [Section 8.2.2](#) and [Section 8.2.4](#) for a discussion of `stateid` structure, lifetime, and validation.

[18.49](#). Operation 56: `WANT_DELEGATION` – Request Delegation

[18.49.1.](#) 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 */
    /* CURRENT_FH: object being delegated */
    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: object being delegated */
    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: object being reclaimed */
    open_delegation_type4    dc_delegate_type;
};

struct WANT_DELEGATION4args {
    uint32_t        wda_want;
    deleg_claim4    wda_claim;
};
```

[18.49.2.](#) RESULT

```
union WANT_DELEGATION4res switch (nfsstat4 wdr_status) {
case NFS4_OK:
    open_delegation4 wdr_resok4;
default:
    void;
};
```

Internet-Draft

NFSv4.1

September 2008

[18.49.3](#). 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:

- o Get a delegation on all types of files except directories.
- o 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, and instead MUST return with `ond_why` set to `WND4_CONTENTION` or `WND4_RESOURCE` and `ond_server_will_push_deleg` or `ond_server_will_signal_avail` set to `FALSE`. When the server indicates that it will notify the client by means of a callback, it will either provide the delegation using a `CB_PUSH_DELEG` operation, or cancel its promise by sending a `CB_WANTS_CANCELLED` operation.
- o Cancel a want for a delegation.

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`:

- o `OPEN4_SHARE_ACCESS_WANT_READ_DELEG`
- o `OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG`
- o `OPEN4_SHARE_ACCESS_WANT_ANY_DELEG`
- o `OPEN4_SHARE_ACCESS_WANT_NO_DELEG`
- o `OPEN4_SHARE_ACCESS_WANT_CANCEL`
- o `OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL`

- o OPEN4_SHARE_ACCESS_WANT_PUSH_DELEG_WHEN_UNCONTENDED

The handling of the above flags in WANT_DELEGATION is the same as in OPEN. Information about the delegation and/or the promises the server is making regarding future callbacks are the same as those

described in the open_delegation4 structure.

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 (see [Section 18.16.3](#)). The server constructs wdr_resok4 the same way it constructs OPEN's "delegation" with one difference: WANT_DELEGATION MUST NOT return a delegation type of OPEN_DELEGATE_NONE.

If (wda_want & OPEN4_SHARE_ACCESS_WANT_DELEG_MASK) is zero then the client is indicating no desire for a delegation and the server MUST return NFS4ERR_INVAL.

The client uses the OPEN4_SHARE_ACCESS_WANT_NO_DELEG flag in the WANT_DELEGATION operation to cancel a previously requested want for a delegation. Note that if the server is in the process of sending the delegation (via CB_PUSH_DELEG) at the time the client sends a cancellation of the want, the delegation might still be pushed to the client.

If WANT_DELEGATION fails to return a delegation, and the server returns NFS4_OK, the server MUST set the delegation type to OPEN4_DELEGATE_NONE_EXT, and set od_whynone, as described in [Section 18.16](#). Write delegations are not available for file types that are not writable. This includes file objects of types: NF4BLK, NF4CHR, NF4LNK, NF4SOCK, and NF4FIFO. If the client requests OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG without OPEN4_SHARE_ACCESS_WANT_READ_DELEG on an object with one of the aforementioned file types, the server must set WND4_WRITE_DELEG_NOT_SUPP_FTYPE.

[18.49.4](#). IMPLEMENTATION

A request for a conflicting delegation is not normally intended to trigger the recall of the existing delegation. Servers may choose to treat some clients as having higher priority such that their wants

will trigger recall of an existing delegation, although that is expected to be an unusual situation.

Servers will generally recall delegations assigned by WANT_DELEGATION on the same basis as those assigned by OPEN. CB_RECALL will generally be done only when other clients perform operations inconsistent with the delegation. The normal response to aging of delegations is to use CB_RECALL_ANY, in order to give the client the opportunity to keep the delegations most useful from its point of view.

[18.50](#). Operation 57: DESTROY_CLIENTID - Destroy a Client ID

[18.50.1](#). ARGUMENT

```
struct DESTROY_CLIENTID4args {
    clientid4      dca_clientid;
};
```

[18.50.2](#). RESULT

```
struct DESTROY_CLIENTID4res {
    nfsstat4      dcr_status;
};
```

[18.50.3](#). DESCRIPTION

The DESTROY_CLIENTID operation destroys the client ID. If there are sessions (both idle and non-idle), opens, locks, delegations, layouts, and/or wants ([Section 18.49](#)) associated with the unexpired lease of the client ID, the server MUST return NFS4ERR_CLIENTID_BUSY. DESTROY_CLIENTID MAY be preceded with a SEQUENCE operation as long as the client ID derived from the session ID of SEQUENCE is not the same as the client ID to be destroyed. If the client IDs are the same, then the server MUST return NFS4ERR_CLIENTID_BUSY.

If DESTROY_CLIENTID is not prefixed by SEQUENCE, it MUST be the only operation in the COMPOUND request (otherwise the server MUST return

NFS4ERR_NOT_ONLY_OP). If the operation is sent without a SEQUENCE preceding it, a client that retransmits the request may receive an error in response, because the original request might have been successfully executed.

[18.50.4.](#) 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 client ID the server can safely reuse the client ID on a future EXCHANGE_ID operation.

[18.51.](#) Operation 58: RECLAIM_COMPLETE - Indicates Reclaims Finished

[18.51.1.](#) ARGUMENT

```
struct RECLAIM_COMPLETE4args {
    /*
     * If rca_one_fs TRUE,
     *
     * CURRENT_FH: object in
     * filesystem reclaim is
     * complete for.
     */
    bool          rca_one_fs;
};
```

[18.51.2.](#) RESULTS

```
struct RECLAIM_COMPLETE4res {
    nfsstat4      rcr_status;
};
```

[18.51.3.](#) DESCRIPTION

A RECLAIM_COMPLETE operation is used to indicate that the client has reclaimed all of the locking state that it will recover, when it is recovering state due to either a server restart or the transfer of a file system to another server. There are two types of RECLAIM_COMPLETE operations:

- o When `rca_one_fs` is FALSE, a global RECLAIM_COMPLETE is being done. This indicates that recovery of all locks that the client held on the previous server instance have been completed.
- o When `rca_one_fs` is TRUE, a file system-specific RECLAIM_COMPLETE is being done. This indicates that recovery of locks for a single fs (the one designated by the current filehandle) due to a file system transition have been completed. Presence of a current filehandle is only required when `rca_one_fs` is set to TRUE.

Once a RECLAIM_COMPLETE is done, there can be no further reclaim operations for locks whose scope is defined as having completed recovery. Once the client sends RECLAIM_COMPLETE, the server will not allow the client to do subsequent reclaims of locking state for that scope and if these are attempted, will return NFS4ERR_NO_GRACE.

Whenever a client establishes a new client ID and before it does the first non-reclaim operation that obtains a lock, it MUST do a global RECLAIM_COMPLETE, even if there are no locks to reclaim. If non-

reclaim locking operations are done before the RECLAIM_COMPLETE, an NFS4ERR_GRACE error will be returned.

Similarly, when the client accesses a file system on a new server, before it sends the first non-reclaim operation that obtains a lock on this new server, it must do a RECLAIM_COMPLETE with `rca_one_fs` set to TRUE and current filehandle within that file system, even if there are no locks to reclaim. If non-reclaim locking operations are done on that file system before the RECLAIM_COMPLETE, an NFS4ERR_GRACE error will be returned.

Any locks not reclaimed at the point at which RECLAIM_COMPLETE is done become non-reclaimable. The client MUST NOT attempt to reclaim them, either during the current server instance or in any subsequent server instance, or on another server to which responsibility for

that file system is transferred. If the client were to do so, it would be violating the protocol by representing itself as owning locks that it does not own, and so has no right to reclaim. See [Section 8.4.3](#) for a discussion of edge conditions related to lock reclaim.

By sending a RECLAIM_COMPLETE, the client indicates readiness to proceed to do normal non-reclaim locking operations. The client should be aware that such operations may temporarily result in NFS4ERR_GRACE errors until the server is ready to terminate its grace period.

[18.51.4](#). IMPLEMENTATION

Servers will typically use the information as to when reclaim activity is complete to reduce the length of the grace period. When the server maintains in persistent storage a list of clients that might have had locks, it is in a position to use the fact that all such clients have done a RECLAIM_COMPLETE to terminate the grace period and begin normal operations (i.e. grant requests for new locks) sooner than it might otherwise.

Latency can be minimized by doing a RECLAIM_COMPLETE as part of the COMPOUND request in which the last lock-reclaiming operation is done. When there are no reclaims to be done, RECLAIM_COMPLETE should be done immediately in order to allow the grace period to end as soon as possible.

RECLAIM_COMPLETE should only be done once for each server instance, or occasion of the transition of a file system. If it is done a second time, the error NFS4ERR_COMPLETE_ALREADY will result. Note that because of the session feature's retry protection, retries of COMPOUND requests containing RECLAIM_COMPLETE operation will not

result in this error.

When a RECLAIM_COMPLETE is done, the client effectively acknowledges any locks not yet reclaimed as lost. This allows the server to again mark this client as able to subsequently recover locks if it had been prevented from doing so, be by logic to prevent the occurrence of edge conditions, as described in [Section 8.4.3](#).

[18.52.](#) Operation 10044: ILLEGAL – Illegal operation

[18.52.1.](#) ARGUMENTS

void;

[18.52.2.](#) RESULTS

```
struct ILLEGAL4res {  
    nfsstat4      status;  
};
```

[18.52.3.](#) 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.

[18.52.4.](#) 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.

[19.](#) NFSv4.1 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.

Both procedures, CB_NULL and CB_COMPOUND, MUST be implemented.

[19.1.](#) Procedure 0: CB_NULL - No Operation

[19.1.1.](#) ARGUMENTS

void;

[19.1.2.](#) RESULTS

void;

[19.1.3.](#) DESCRIPTION

CB_NULL is the standard ONC RPC NULL procedure, with the standard void argument and 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 the server to client.

[19.1.4.](#) ERRORS

None.

[19.2.](#) Procedure 1: CB_COMPOUND - Compound Operations

[19.2.1.](#) ARGUMENTS

```
enum nfs_cb_opnum4 {
    OP_CB_GETATTR          = 3,
    OP_CB_RECALL           = 4,
    /* Callback operations new to NFSv4.1 */
    OP_CB_LAYOUTRECALL     = 5,
    OP_CB_NOTIFY           = 6,
    OP_CB_PUSH_DELEG       = 7,
    OP_CB_RECALL_ANY       = 8,
    OP_CB_RECALLABLE_OBJ_AVAIL = 9,
    OP_CB_RECALL_SLOT      = 10,
    OP_CB_SEQUENCE         = 11,
    OP_CB_WANTS_CANCELLED  = 12,
    OP_CB_NOTIFY_LOCK      = 13,
    OP_CB_NOTIFY_DEVICEID  = 14,

    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_LAYOUTRECALL:
    CB_LAYOUTRECALL4args     opcblayoutrecall;
  case OP_CB_NOTIFY:
    CB_NOTIFY4args           opcbnotify;
  case OP_CB_PUSH_DELEG:
    CB_PUSH_DELEG4args       opcbpush_deleg;
  case OP_CB_RECALL_ANY:
    CB_RECALL_ANY4args       opcbrecall_any;
  case OP_CB_RECALLABLE_OBJ_AVAIL:
    CB_RECALLABLE_OBJ_AVAIL4args opcbrecallable_obj_avail;
  case OP_CB_RECALL_SLOT:
    CB_RECALL_SLOT4args      opcbrecall_slot;
  case OP_CB_SEQUENCE:
    CB_SEQUENCE4args         opcbsequence;
  case OP_CB_WANTS_CANCELLED:
    CB_WANTS_CANCELLED4args  opcbwants_cancelled;
  case OP_CB_NOTIFY_LOCK:
    CB_NOTIFY_LOCK4args      opcbnotify_lock;
  case OP_CB_NOTIFY_DEVICEID:
    CB_NOTIFY_DEVICEID4args  opcbnotify_deviceid;
  case OP_CB_ILLEGAL:
    void;
};
```

```
struct CB_COMPOUND4args {
  utf8str_cs    tag;
  uint32_t      minorversion;
  uint32_t      callback_ident;
  nfs_cb_argop4 argarray<>;
};
```

Internet-Draft

NFSv4.1

September 2008

[19.2.2.](#) RESULTS

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

  /* new NFSv4.1 operations */
  case OP_CB_LAYOUTRECALL:
                                CB_LAYOUTRECALL4res
                                opcblayoutrecall;

  case OP_CB_NOTIFY:       CB_NOTIFY4res   opcbnotify;

  case OP_CB_PUSH_DELEG:   CB_PUSH_DELEG4res
                                opcbpush_deleg;

  case OP_CB_RECALL_ANY:   CB_RECALL_ANY4res
                                opcbrecall_any;

  case OP_CB_RECALLABLE_OBJ_AVAIL:
                                CB_RECALLABLE_OBJ_AVAIL4res
                                opcbrecallable_obj_avail;

  case OP_CB_RECALL_SLOT:
                                CB_RECALL_SLOT4res
                                opcbrecall_slot;

  case OP_CB_SEQUENCE:     CB_SEQUENCE4res opcbsequence;

  case OP_CB_WANTS_CANCELLED:
                                CB_WANTS_CANCELLED4res
                                opcbwants_cancelled;

  case OP_CB_NOTIFY_LOCK:
                                CB_NOTIFY_LOCK4res
                                opcbnotify_lock;

  case OP_CB_NOTIFY_DEVICEID:
                                CB_NOTIFY_DEVICEID4res
```

```

                                opcbnotify_deviceid;

/* Not new operation */
case OP_CB_ILLEGAL:    CB_ILLEGAL4res  opcbillegal;
};

```

```

struct CB_COMPOUND4res {
    nfsstat4 status;
    utf8str_cs    tag;
    nfs_cb_resop4  resarray<>;
};

```

[19.2.3.](#) 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.

During 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. Refer to [Section 2.10.6.4](#) for details.

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.

The "tag" field is handled the same way as that of COMPOUND procedure (see [Section 16.2.3](#)).

Illegal operation codes are handled in the same way as they are

handled for the COMPOUND procedure.

[19.2.4.](#) 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.

[19.2.5.](#) ERRORS

CB_COMPOUND will of course return every error that each operation on the backchannel can return (see Table 7). However if CB_COMPOUND

returns zero operations, obviously the error returned by COMPOUND has nothing to do with an error returned by an operation. The list of errors CB_COMPOUND will return if it processes zero operations include:

CB_COMPOUND error returns

Error	Notes
NFS4ERR_BADCHAR	The tag argument has a character the replier does not support.
NFS4ERR_BADXDR	
NFS4ERR_DELAY	The tag argument is not in UTF-8 encoding.
NFS4ERR_INVALID	
NFS4ERR_MINOR_VERSION_MISMATCH	
NFS4ERR_SERVERFAULT	
NFS4ERR_TOO_MANY_OPS	
NFS4ERR_REP_TOO_BIG	
NFS4ERR_REP_TOO_BIG_TO_CACHE	
NFS4ERR_REQ_TOO_BIG	

Table 14

[20.](#) NFSv4.1 Callback Operations

[20.1.](#) Operation 3: CB_GETATTR - Get Attributes

[20.1.1.](#) ARGUMENT

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

[20.1.2.](#) RESULT

```
struct CB_GETATTR4resok {
    fattr4 obj_attributes;
};

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

[20.1.3.](#) 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 [Section 10.4.3](#) for a full description of how the client and server are to interact with the use of CB_GETATTR.

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

[20.1.4.](#) 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).

[20.2.](#) Operation 4: CB_RECALL - Recall a Delegation

[20.2.1.](#) ARGUMENT

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

[20.2.2.](#) RESULT

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

[20.2.3.](#) DESCRIPTION

The CB_RECALL operation is used to begin the process of recalling a delegation and returning it to the server.

The truncate flag is used to optimize recall for a file object which is a regular file and is about to be truncated to zero. When it is TRUE, the client is freed of the 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 a

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.

[20.2.4.](#) IMPLEMENTATION

The client SHOULD reply to the callback immediately. Replying does not complete the recall except when the value of the reply's status field is neither NFS4ERR_DELAY nor NFS4_OK. The recall is not complete until the delegation is returned using a DELEGRETURN operation.

[20.3.](#) Operation 5: CB_LAYOUTRECALL - Recall Layout from Client

[20.3.1.](#) ARGUMENT

/*

* NFSv4.1 callback arguments and results

```

*/

enum layoutrecall_type4 {
    LAYOUTRECALL4_FILE = LAYOUT4_RET_REC_FILE,
    LAYOUTRECALL4_FSID = LAYOUT4_RET_REC_FSID,
    LAYOUTRECALL4_ALL = LAYOUT4_RET_REC_ALL
};

struct layoutrecall_file4 {
    nfs_fh4        lor_fh;
    offset4        lor_offset;
    length4        lor_length;
    stateid4       lor_stateid;
};

union layoutrecall4 switch(layoutrecall_type4 lor_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;
};

```

[20.3.2.](#) RESULT

```

struct CB_LAYOUTRECALL4res {
    nfsstat4        clorr_status;
};

```

[20.3.3.](#) DESCRIPTION

The CB_LAYOUTRECALL operation is used by the server to recall layouts from the client; as a result, the client will begin the process of returning layouts via LAYOUTRETURN. The CB_LAYOUTRECALL operation specifies one of three forms of recall processing with the value of

layoutrecall_type4. The recall is either for a specific layout (by file), for an entire file system (FSID), or for all file systems (ALL).

The behavior of the operation varies based on the value of the layoutrecall_type4. The value and behaviors are:

LAYOUTRECALL4_FILE

For a layout to match the recall request, the values of the following fields must match those of the layout: clora_type, clora_iomode, lor_fh, and the byte range specified by lor_offset and lor_length. The clora_iomode field may have a special value of LAYOUTIOMODE4_ANY. The special value LAYOUTIOMODE4_ANY will match any iomode originally returned in a layout; therefore it acts as a wild card. The other special value used is for lor_length. If lor_length has a value of NFS4_UINT64_MAX, the lor_length field means the maximum possible file size. If a matching layout is found, it MUST be returned using the LAYOUTRETURN operation (see [Section 18.44](#)). An example of the field's special value use is if clora_iomode is LAYOUTIOMODE4_ANY, lor_offset is zero, and lor_length is NFS4_UINT64_MAX, then the entire layout is to be returned.

The NFS4ERR_NOMATCHING_LAYOUT error is only returned when the client does not hold layouts for the file or if the client does not have any overlapping layouts for the specification in the layout recall.

LAYOUTRECALL4_FSID and LAYOUTRECALL4_ALL

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. See [Section 12.5.5.2.1.5](#) for considerations with "bulk" recall of layouts.

The NFS4ERR_NOMATCHING_LAYOUT error is only returned when the client does not hold layouts and does not have valid deviceid mappings.

In processing the layout recall request, the client also varies its

behavior based on the value of the `clora_changed` field. This field

is used by the server to provide additional context for the reason why the layout is being recalled. A FALSE value for `clora_changed` indicates that no change in the layout is expected and the client may write modified data to the storage devices involved; this must be done prior to returning the layout via LAYOUTRETURN. A TRUE value for `clora_changed` indicates that the server is changing the layout. Examples of layout changes and reasons for a TRUE indication are: the metadata server is restriping the file or a permanent error has occurred on a storage device and the metadata server would like to provide a new layout for the file. Therefore, a `clora_changed` value of TRUE indicates some level of change for the layout and the client SHOULD NOT write and commit modified data to the storage devices. In this case, the client writes and commits data through the metadata server.

See [Section 12.5.3](#) for a description of how the `lor_stateid` field in the arguments is to be constructed. Note that the "seqid" field of `lor_stateid` MUST NOT be zero. See [Section 8.2](#), [Section 12.5.3](#), and [Section 12.5.5.2](#) for a further discussion and requirements.

[20.3.4](#). IMPLEMENTATION

The client's processing for CB_LAYOUTRECALL is similar to CB_RECALL (recall of file delegations) in that the client responds to the request before actually returning layouts via the LAYOUTRETURN operation. While the client responds to the CB_LAYOUTRECALL immediately, the operation is not considered complete (i.e. considered pending) until all affected layouts are returned to the server via the LAYOUTRETURN operation.

Before returning the layout to the server via LAYOUTRETURN, the client should wait for the response from in-process or in-flight READ, WRITE, or COMMIT operations that use the recalled layout.

If the client is holding modified data which is affected by a recalled layout, the client has various options for writing the data to the server. As always, the client may write the data through the metadata server. In fact, the client may not have a choice other than writing to the metadata server when the `clora_changed` argument is TRUE and a new layout is unavailable from the server. However,

the client may be able to write the modified data to the storage device if the `clora_changed` argument is `FALSE`; this needs to be done before returning the layout via `LAYOUTRETURN`. If the client were to obtain a new layout covering the modified data's range, then writing to the storage devices is an available alternative. Note that before obtaining a new layout, the client must first return the original layout.

In the case of modified data being written while the layout is held, the client must use `LAYOUTCOMMIT` operations at the appropriate time; as required `LAYOUTCOMMIT` must be done before the `LAYOUTRETURN`. If a large amount of modified data is outstanding, the client may send `LAYOUTRETURNS` for portions of the recalled layout; this allows the server to monitor the client's progress and adherence to the original recall request. However, the last `LAYOUTRETURN` in a sequence of returns, **MUST** specify the full range being recalled (see [Section 12.5.5.1](#) for details).

If a server needs to delete a device ID, and there are layouts referring to the device ID, `CB_LAYOUTRECALL` **MUST** be invoked to cause the client to return all layouts referring to device ID before the server can delete the device ID. If the client does not return the affected layouts, the server **MAY** revoke the layouts.

[20.4](#). Operation 6: `CB_NOTIFY` - Notify Client of Directory Changes

[20.4.1](#). ARGUMENT

```
/*
 * Directory notification types.
 */
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
};

/* 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_remove4 {
    notify_entry4 nrm_old_entry;
    nfs_cookie4   nrm_old_entry_cookie;
};

```

```

};

struct notify_add4 {
    /*
     * Information on object
     * possibly renamed over.
     */
    notify_remove4 nad_old_entry<1>;
    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_rename4 {
    notify_remove4 nrn_old_entry;
    notify_add4    nrn_new_entry;
};

struct notify_verifier4 {
    verifier4      nv_old_cookieverf;
};

```



```

        verifier4      nv_new_cookieverf;
};

/*
 * Objects of type notify_<>4 and
 * notify_device_<>4 are encoded in this.
 */
typedef opaque notifylist4<>;

struct notify4 {
    /* composed from notify_type4 or notify_deviceid_type4 */
    bitmap4      notify_mask;
    notifylist4  notify_vals;
};

struct CB_NOTIFY4args {
    stateid4     cna_stateid;
    nfs_fh4      cna_fh;
    notify4      cna_changes<>;
};

```

[20.4.2.](#) RESULT

```

struct CB_NOTIFY4res {
    nfsstat4     cnr_status;
};

```

[20.4.3.](#) DESCRIPTION

The CB_NOTIFY operation is used by the server to send notifications to clients about changes to delegated directories. The registration of notifications for the directories occurs when the delegation is established using GET_DIR_DELEGATION. These notifications are sent over the backchannel. The notification is sent once the original request has been processed on the server. The server will send an array of notifications for changes that might have occurred in the directory. The notifications are sent as list of pairs of bitmaps and values. See [Section 3.3.7](#) for a description of how NFSv4.1 bitmaps work.

If the server has more notifications than can fit in the CB_COMPOUND request, it SHOULD send a sequence of serial CB_COMPOUND requests so that the client's view of the directory does not become confused. E.g. If the server indicates a file named "foo" is added, and that the file "foo" is removed, the order in which the client receives these notifications needs to be the same as the order in which corresponding operations occurred on the server.

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:

NOTIFY4_ADD_ENTRY

The server will send information about the new directory 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. The server will send this type of entry when a file is actually being created, when an entry is being added to a directory as a result of a rename across directories (see below), and when a hard link is being created to an existing file. 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 file name caches and directory caches where this entry should be cached. If the new entry's cookie is available, it will be in the nad_new_entry_cookie (another variable length array of up to one element) field. If the addition of the entry causes another entry to be deleted (which can only happen in the rename case) atomically with the addition, then information on this entry is reported in nad_old_entry.

NOTIFY4_REMOVE_ENTRY

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.

NOTIFY4_RENAME_ENTRY

The server will send information about both the old entry and the new entry. This includes name and attributes for each entry. In addition, if the rename causes the deletion of an entry (i.e. the case of a file renamed over) then this is reported in `nrn_new_new_entry.nad_old_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.

NOTIFY4_CHANGE_CHILD_ATTRS/NOTIFY4_CHANGE_DIR_ATTRS

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's attributes in the directory by using two separate attribute masks. The client cannot ask for change attribute notification for a specific 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.

NOTIFY4_CHANGE_COOKIE_VERIFIER

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 re-send a READDIR to get the new set of cookies.

[20.5.](#) Operation 7: CB_PUSH_DELEG - Offer Previously Requested Delegation to Client

[20.5.1.](#) ARGUMENT

```
struct CB_PUSH_DELEG4args {  
    nfs_fh4          cpda_fh;  
    open_delegation4 cpda_delegation;  
  
};
```

[20.5.2.](#) RESULT

```
struct CB_PUSH_DELEG4res {  
    nfsstat4 cpdr_status;  
  
};
```

[20.5.3.](#) DESCRIPTION

CB_PUSH_DELEG is used by the server to both signal to the client that the delegation it wants (previously indicated via a want established from an OPEN or WANT_DELEGATION operation) 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 by returning NFS4ERR_REJECT_DELEG. When a delegation is rejected in this fashion, the want previously established is permanently deleted and the delegation is subject to acquisition by another client.

[20.5.4.](#) IMPLEMENTATION

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 be processed behind other delegation requests or registered wants.

When a client returns a status other than NFS4_OK, NFS4ERR_DELAY, or

NFS4ERR_REJECT_DELAY, the want remains pending, although servers may decide to cancel the want by sending a CB_WANTS_CANCELLED.

[20.6.](#) Operation 8: CB_RECALL_ANY - Keep Any N Recallable Objects

[20.6.1.](#) 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     = 4;
const RCA4_TYPE_MASK_OBJ_LAYOUT_MIN = 8;
const RCA4_TYPE_MASK_OBJ_LAYOUT_MAX = 9;
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;
};
```

[20.6.2.](#) RESULT

```
struct CB_RECALL_ANY4res {
    nfsstat4      crar_status;
};
```

[20.6.3.](#) 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 metadata 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

Internet-Draft

NFSv4.1

September 2008

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 send 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.

A number of 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 reserved for object-based storage protocols and for other experimental storage protocols. An 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.

RCA4_TYPE_MASK_RDATA_DLG

The client is to return read delegations on non-directory file objects.

RCA4_TYPE_MASK_WDATA_DLG

The client is to return write delegations on regular file objects.

RCA4_TYPE_MASK_DIR_DLG

The client is to return directory delegations.

RCA4_TYPE_MASK_FILE_LAYOUT

The client is to return layouts of type LAYOUT4_NFSV4_1_FILES.

Internet-Draft

NFSv4.1

September 2008

See [\[30\]](#) for a description.

RCA4_TYPE_MASK_OBJ_LAYOUT_MIN to RCA4_TYPE_MASK_OBJ_LAYOUT_MAX

See [\[29\]](#) for a description.

RCA4_TYPE_MASK_OTHER_LAYOUT_MIN to RCA4_TYPE_MASK_OTHER_LAYOUT_MAX

This range is reserved for telling the client to recall layouts of experimental or site specific layout types (see [Section 3.3.13](#)).

When a bit is set in the type mask that corresponds to an undefined type of recallable object, NFS4ERR_INVALID MUST be returned. When a bit is set that corresponds to a defined type of object, but the client does not support an object of the type, NFS4ERR_INVALID MUST 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.

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.

[20.6.4.](#) IMPLEMENTATION

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.

[20.7.](#) Operation 9: CB_RECALLABLE_OBJ_AVAIL - Signal Resources for Recallable Objects

[20.7.1.](#) ARGUMENT

```
typedef CB_RECALL_ANY4args CB_RECALLABLE_OBJ_AVAIL4args;
```

[20.7.2.](#) RESULT


```

struct CB_RECALLABLE_OBJ_AVAIL4res {
    nfsstat4      croa_status;
};

```

[20.7.3.](#) 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 `craa_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.

The server is not obligated to reserve the difference between the

number of the objects the client currently has and the value of `craa_objects_to_keep`, nor does delaying the reply to CB_RECALLABLE_OBJ_AVAIL prevent the server from using the resources of the recallable objects for another purpose. Indeed, if a client responds slowly to CB_RECALLABLE_OBJ_AVAIL, the server might interpret the client as having reduced capability to manage recallable objects, and so cancel or reduce any reservation it is maintaining on behalf of the client. Thus if the client desires to acquire more recallable objects, it needs to reply quickly to CB_RECALLABLE_OBJ_AVAIL, and then send the appropriate operations to acquire recallable objects.

[20.8.](#) Operation 10: CB_RECALL_SLOT - Change Flow Control Limits

[20.8.1.](#) ARGUMENT

```

struct CB_RECALL_SLOT4args {
    slotid4      rsa_target_highest_slotid;
};

```

[20.8.2.](#) RESULT

```
struct CB_RECALL_SLOT4res {  
    nfsstat4    rsr_status;  
};
```

[20.8.3.](#) DESCRIPTION

The CB_RECALL_SLOT operation requests the client to return session slots, and if applicable, transport credits (e.g. RDMA credits for connections associated with the operations channel) of the session's fore channel. CB_RECALL_SLOT specifies `rsa_target_highest_slotid`, the value of the target highest slot ID the server wants for the session. The client MUST then progress toward reducing the session's highest slot ID to the target value.

If the session has only non-RDMA connections associated with its operations channel, then the client need only wait for all outstanding requests with a slot ID > `rsa_target_highest_slotid` to complete, then send a single COMPOUND consisting of a single SEQUENCE operation, with the `sa_highestslot` field set to `rsa_target_highest_slotid`. If there are RDMA-based connections associated with operation channel, then the client needs to also send enough zero-length RDMA Sends to take the total RDMA credit count to `rsa_target_highest_slotid + 1` or below.

[20.8.4.](#) IMPLEMENTATION

If the client fails to reduce highest slot it has on the fore channel to what the server requests, the server can force the issue by asserting flow control on the receive side of all connections bound to the fore channel, and then finish servicing all outstanding requests that are in slots greater than `rsa_target_highest_slotid`. Once that is done, the server can then open the flow control, and any time the client sends a new request on a slot greater than `rsa_target_highest_slotid`, the server can return NFS4ERR_BADSLLOT.

[20.9.](#) Operation 11: CB_SEQUENCE - Supply Backchannel Sequencing and Control

[20.9.1.](#) 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<>;
};
```

[20.9.2.](#) RESULT

```
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;
};

```

[20.9.3.](#) DESCRIPTION

The CB_SEQUENCE operation is used to manage operational accounting for the backchannel of the session on which a request is sent. The contents include the session ID to which this request belongs, the slot ID and sequence ID used by the server to implement session request control and exactly once semantics, and exchanged slot ID maxima which are used to adjust the size of the reply cache. This operation will appear once as the first operation in each CB_COMPOUND request or a protocol error MUST result. See [Section 18.46.3](#) 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 2.10.6.1.3](#)).

The `csa_referring_call_lists` array is the list of COMPOUND requests, identified by session ID, slot ID and sequence ID. These are requests that the client previously sent to the server. These previous requests created state that some operation(s) in the same CB_COMPOUND as the `csa_referring_call_lists` are identifying. A session ID is included because leased state is tied to a client ID, and a client ID can have multiple sessions. See [Section 2.10.6.3](#).

The value of the `csa_sequenceid` argument relative to the cached sequence ID on the slot falls into one of three cases.

- o If the difference between `csa_sequenceid` and the client's cached sequence ID at the slot ID is two (2) or more, or if `csa_sequenceid` is less than the cached sequence ID (accounting for

wraparound of the unsigned sequence ID value), then the client MUST return NFS4ERR_SEQ_MISORDERED.

- o If csa_sequenceid and the cached sequence ID are the same, this is a retry, and the client returns the CB_COMPOUND request's cached reply.
- o If csa_sequenceid is one greater (accounting for wraparound) than the cached sequence ID, then this is a new request, and the slot's sequence ID 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, maintain the session's activity, and when the server receives the CB_SEQUENCE reply, renew the lease of state related to the client ID.

If the server reuses a slot ID and sequence ID for a completely different request, the client MAY treat the request as if it is retry of what it has already executed. The client MAY however detect the server's illegal reuse and return NFS4ERR_SEQ_FALSE_RETRY.

If CB_SEQUENCE returns an error, then the state of the slot (sequence ID, cached reply) MUST NOT change.

The client returns two "highest_slotid" values: csr_highest_slotid, and csr_target_highest_slotid. The former is the highest slot ID the client will accept in a future CB_SEQUENCE operation, and SHOULD NOT be less than the value of csa_highest_slotid (but see [Section 2.10.6.1](#) for an exception). The latter is the highest slot ID the client would prefer the server use on a future CB_SEQUENCE operation.

[20.10](#). Operation 12: CB_WANTS_CANCELLED - Cancel Pending Delegation Wants

[20.10.1](#). ARGUMENT

```
struct CB_WANTS_CANCELLED4args {
    bool cwca_contended_wants_cancelled;
    bool cwca_resourced_wants_cancelled;
};
```

Internet-Draft

NFSv4.1

September 2008

[20.10.2.](#) RESULT

```
struct CB_WANTS_CANCELLED4res {  
    nfsstat4      cwcs_status;  
};
```

[20.10.3.](#) 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 cwca_contended_wants_cancelled is 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 to 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.

[20.10.4.](#) IMPLEMENTATION

When a client has an OPEN, WANT_DELEGATION, or GET_DIR_DELEGATION request outstanding, when a CB_WANTS_CANCELLED is sent, the server may need to make clear to the client whether a promise to signal delegation availability happened before the CB_WANTS_CANCELLED and is thus covered by it, or after the CB_WANTS_CANCELLED in which case it was not covered by it. The server can make this distinction by putting the appropriate requests into the list of referring calls in the associated CB_SEQUENCE.

[20.11.](#) Operation 13: CB_NOTIFY_LOCK - Notify Client of Possible Lock Availability

[20.11.1.](#) ARGUMENT

```
struct CB_NOTIFY_LOCK4args {  
    nfs_fh4      cnla_fh;
```

```
    lock_owner4 cnla_lock_owner;
};
```

[20.11.2.](#) RESULT

```
struct CB_NOTIFY_LOCK4res {
    nfsstat4      cnlr_status;
};
```

[20.11.3.](#) DESCRIPTION

The server can use this operation to indicate that a byte-range lock for the given file and lock-owner, previously requested by the client via an unsuccessful LOCK request, might be 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. If the server supports this callback for a given file, it MUST set the OPEN4_RESULT_MAY_NOTIFY_LOCK flag when responding to successful opens for that file. This does not commit the server to the use of CB_NOTIFY_LOCK, but the client may use this as a hint to decide how frequently to poll for locks derived from that open.

If an OPEN operation results in an upgrade, in which the stateid returned has an "other" value matching that of a stateid already allocated, with a new "seqid" indicating a change in the lock being represented, then the value of the OPEN4_RESULT_MAY_NOTIFY_LOCK flag when responding to that new OPEN controls handling from that point going forward. When parallel OPENS are done on the same file and open-owner, the ordering of the "seqid" field of the returned stateid (subject to wraparound) are to be used to select the controlling value of the OPEN4_RESULT_MAY_NOTIFY_LOCK flag.

[20.11.4.](#) 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 LOCK 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 [Section 9.6](#).

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.

[20.12](#). Operation 14: CB_NOTIFY_DEVICEID - Notify Client of Device ID Changes

[20.12.1](#). ARGUMENT

```
/*
 * Device notification types.
 */
enum notify_deviceid_type4 {
    NOTIFY_DEVICEID4_CHANGE = 1,
    NOTIFY_DEVICEID4_DELETE = 2
};

/* For NOTIFY4_DEVICEID4_DELETE */
struct notify_deviceid_delete4 {
    layouttype4    ndd_layouttype;
    deviceid4      ndd_deviceid;
};

/* For NOTIFY4_DEVICEID4_CHANGE */
struct notify_deviceid_change4 {
    layouttype4    ndc_layouttype;
    deviceid4      ndc_deviceid;
    bool           ndc_immediate;
};

struct CB_NOTIFY_DEVICEID4args {
```



```
        notify4 cnda_changes<>;  
};
```

[20.12.2](#). RESULT

```
struct CB_NOTIFY_DEVICEID4res {  
    nfsstat4      cndr_status;  
};
```

[20.12.3](#). DESCRIPTION

The CB_NOTIFY_DEVICEID operation is used by the server to send notifications to clients about changes to pNFS device IDs. The registration of device ID notifications is optional and is done via GETDEVICEINFO. These notifications are sent over the backchannel once the original request has been processed on the server. The server will send an array of notifications, cnda_changes, as a list

of pairs of bitmaps and values. See [Section 3.3.7](#) for a description of how NFSv4.1 bitmaps work.

As with CB_NOTIFY ([Section 20.4.3](#)), it is possible the server has more notifications than can fit in a CB_COMPOUND, thus requiring multiple CB_COMPOUNDS. Unlike CB_NOTIFY, serialization is not an issue because unlike directory entries, device IDs cannot be re-used after being deleted ([Section 12.2.10](#)).

All device ID notifications contain a device ID and a layout type. The layout type is necessary because two different layout types can share the same device ID, and the common device ID can have completely different mappings for each layout type.

The server will send the following notifications:

NOTIFY_DEVICEID4_CHANGE

A previously provided device ID to device address mapping has changed and the client uses GETDEVICEINFO to obtain the updated mapping. The notification is encoded in a value of data type notify_deviceid_change4. This data type also contains a boolean field, ndc_immediate, which if TRUE indicates that the change will

be enforced immediately, and so the client might not be able to complete any pending I/O to the device ID. If `ndc_immediate` is `FALSE`, then for an indefinite time, the client can complete pending I/O. After pending I/O is complete, the client **SHOULD** get the new device ID to device address mappings before issuing new I/O to the device ID.

NOTIFY4_DEVICEID_DELETE

Deletes a device ID from the mappings. This notification **MUST NOT** be sent if the client has a layout that refers to the device ID. In other words if the server is sending a delete device ID notification, one of the following is true for layouts associated with the layout type:

- * The client never had a layout referring to that device ID.
- * The client has returned all layouts referring to that device ID.
- * The server has revoked all layouts referring to that device ID.

The notification is encoded in a value of data type `notify_deviceid_delete4`. After a server deletes a device ID, it **MUST NOT** reuse that device ID for the same layout type until the client ID is deleted.

[20.13.](#) Operation 10044: CB_ILLEGAL - Illegal Callback Operation

[20.13.1.](#) ARGUMENT

void;

[20.13.2.](#) RESULT

```
/*
 * CB_ILLEGAL: Response for illegal operation numbers
 */
struct CB_ILLEGAL4res {
    nfsstat4      status;
};
```

[20.13.3.](#) DESCRIPTION

This operation is a placeholder for encoding a result to handle the case of the server sending an operation code within CB_COMPOUND that is not defined in the NFSv4.1 specification. See [Section 19.2.3](#) for more details.

The status field of CB_ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

[20.13.4.](#) 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 an instance of data type CB_ILLEGAL4res will not be returned.

[21.](#) Security Considerations

Historically the authentication of model of NFS had the entire machine being the NFS client, and the NFS server trusting the NFS client to authenticate the end-user. The NFS server in turn shared its files only to specific clients, as identified by the client's source network address. Given this model, the AUTH_SYS RPC security flavor simply identified the end-user using the client to the NFS server. When processing NFS responses, the client ensured that the responses came from the same network address and port number that the request was sent to. While such a model is easy to implement and simple to deploy and use, it is unsafe. Thus, NFSv4.1 implementations are REQUIRED to support a security model that uses

end to end authentication, where an end-user on a client mutually authenticates (via cryptographic schemes that do not expose passwords or keys in the clear on the network) to a principal on an NFS server. Consideration is also be given to the integrity and privacy of NFS requests and responses. The issues of end to end mutual authentication, integrity, and privacy are discussed [Section 2.2.1.1.1.](#)

Note that being REQUIRED to implement does not mean REQUIRED to use; AUTH_SYS can be used by NFSv4.1 clients and servers. However, AUTH_SYS is merely an OPTIONAL security flavor in NFSv4.1, and so interoperability via AUTH_SYS is not assured.

For reasons of reduced administration overhead, better performance and/or reduction of CPU utilization, users of NFSv4.1 implementations may opt to not use security mechanisms that enable integrity protection on each remote procedure call and response. The use of mechanisms without integrity leaves the user vulnerable to an attacker in the middle of the NFS client and server that modifies the RPC request and/or the response. While implementations are free to provide the option to use weaker security mechanisms, there are three operations in particular that warrant the implementation overriding user choices.

- o The first two such operations are SECINFO and SECINFO_NO_NAME. It is RECOMMENDED that the client send both operations such that they is protected with a security flavor that has integrity protection, such as RPCSEC_GSS with either the rpc_gss_svc_integrity or rpc_gss_svc_privacy service. Without integrity protection encapsulating SECINFO and SECINFO_NO_NAME and their results, an attacker in the middle could modify results such that the client might select a weaker algorithm in the set allowed by server, making the client and/or server vulnerable to further attacks.
- o The third operation that should definitely use integrity protection is any GETATTR for the fs_locations and fs_locations_info attributes. The attack has two steps. First the attacker modifies the unprotected results of some operation to return NFS4ERR_MOVED. Second, when the client follows up with a GETATTR for the fs_locations or fs_locations_info attributes, the attacker modifies the results to cause the client migrate its traffic to a server controlled by the attacker.

Relative to previous NFS versions, NFSv4.1 has additional security considerations for pNFS (see [Section 12.9](#) and [Section 13.12](#)), locking and session state (see [Section 2.10.8.3](#)).

This section uses terms that are defined in [\[43\]](#).

[22.1](#). Named Attribute Definitions

IANA will create a registry called the "NFSv4 Named Attribute Definitions Registry".

The NFSv4.1 protocol supports the association of a file with zero or more named attributes. The name space identifiers for these attributes are defined as string names. The protocol does not define the specific assignment of the name space for these file attributes. An IANA registry will promote interoperability where common interests exist. While application developers are allowed to define and use attributes as needed, they are encouraged to register the attributes with IANA.

Such registered named attributes are presumed to apply to all minor versions of NFSv4, including those defined subsequently to the registration. Where the named attribute is intended to be limited with regard to the minor versions for which they are not be used, the assignment in registry will clearly state the applicable limits.

All assignments to the registry are made on a First Come First Served basis, per section 4.1 of [\[43\]](#). The policy for each assignment is Specification Required, per section 4.1 of [\[43\]](#).

Under the NFSv4.1 specification, the name of a named attribute can in theory be up to $2^{32} - 1$ bytes in length, but in practice NFSv4.1 clients and servers will be unable to handle a string that long. IANA should reject any assignment request with a named attribute that exceeds 128 UTF-8 characters. To give IESG the flexibility to set up bases of assignment of Experimental Use and Standards Action, the prefixes of "EXPE" and "STDS" are Reserved. The zero length named attribute name is Reserved.

The prefix "PRIV" is allocated for Private Use. A site that wants to make use of unregistered named attributes without risk of conflicting with an assignment in IANA's registry should use the prefix "PRIV" in all of its named attributes.

Because some NFSv4.1 clients and servers have case insensitive semantics, the fifteen additional lower case and mixed case permutations of each of "EXPE", "PRIV", and "STDS", are Reserved (e.g. "expe", "expE", "exPe", etc. are Reserved). Similarly, IANA must not allow two assignments that would conflict if both named attributes were converted to a common case.

The registry of named attributes is a list of assignments, each containing three fields for each assignment.

1. A US-ASCII string name that is the actual name of the attribute. This name must be unique. This string name can be 1 to 128 UTF-8 characters long.
2. A reference to the specification of the named attribute. The reference can consume up to 256 bytes (or more if IANA permits).
3. The point of contact of the registrant. The point of contact can consume up to 256 bytes (or more if IANA permits).

[22.1.1.](#) Initial Registry

There is no initial registry.

[22.1.2.](#) Updating Registrations

The registrant is always permitted to update the point of contact field. To make any other change will require Expert Review or IESG Approval.

[22.2.](#) Device ID Notifications

IANA will create a registry called the "NFSv4.1 Device ID Notifications Registry".

The potential exists for new notification types to be added to the CB_NOTIFY_DEVICEID operation [Section 20.12](#). This can be done via changes to the operations that register notifications, or by adding new operations to NFSv4. This requires a new minor version of NFSv4, and requires a standards track document from IETF. Another way to add a notification is to specify a new layout type (see [Section 22.4](#)).

Hence all assignments to the registry are made on a Standards Action basis per section 4.1 of [\[43\]](#), with Expert Review required.

The registry is a list of assignments, each containing five fields per assignment.

1. The name of the notification type. This name must have the prefix: "NOTIFY_DEVICEID4_". This name must be unique.
2. The value of the notification. IANA will assign this number, and

the request from the registrant will use TBD1 instead of an actual value. IANA MUST use a whole number which can be no

higher than 2^32-1, and should be the next available value. The value assigned must be unique. A Designated Expert must be used to ensure that when the name of the notification type and its value are added to the NFSv4.1 notify_deviceid_type4 enumerated data type in the NFSv4.1 XDR description ([12]), the result continues to be a valid XDR description.

3. The Standards Track RFC(s) that describe the notification. If the RFC(s) have not yet been published, the registrant will use RFCTBD2, RFCTBD3, etc. instead of an actual RFC number.
4. How the RFC introduces the notification. This is indicated by a single US-ASCII value. If the value is N, it means a minor revision to the NFSv4 protocol. If the value is L, it means a new pNFS layout type. Other values can be used with IESG Approval.
5. The minor versions of NFSv4 that are allowed to the use the notification. While these are numeric values, IANA will not allocate and assign them; the author of the relevant RFCs with IESG Approval assigns these numbers. Each time there is new minor version of NFSv4 approved, a Designated Expert should review the registry to make recommended updates as needed.

22.2.1. Initial Registry

The initial registry is in Table 15. Note that next available value is zero.

Notification Name	Value	RFC	How	Minor Versions
NOTIFY_DEVICEID4_CHANGE	1	RFCTBD10	N	1
NOTIFY_DEVICEID4_DELETE	2	RFCTBD10	N	1

Table 15: Initial Device ID Notification Assignments

22.2.2. Updating Registrations

The update of a registration will require IESG Approval on the advice of a Designated Expert.

[22.3](#). Object Recall Types

IANA will create a registry called the "NFSv4.1 Recallable Object Types Registry".

Shepler, et al.

Expires March 9, 2009

[Page 590]

Internet-Draft

NFSv4.1

September 2008

The potential exists for new object types to be added to the CB_RECALL_ANY operation (see [Section 20.6](#)). This can be done via changes to the operations that add recallable types, or by adding new operations to NFSv4. This requires a new minor version of NFSv4, and requires a standards track document from IETF. Another way to add a new recallable object is to specify a new layout type (see [Section 22.4](#)).

All assignments to the registry are made on a Standards Action basis per section 4.1 of [\[43\]](#), with Expert Review required.

Recallable object types are 32 bit unsigned numbers. There are no Reserved values. Values in the range 12 through 15, inclusive, are for Private Use.

The registry is a list of assignments, each containing five fields per assignment.

1. The name of the recallable object type. This name must have the prefix: "RCA4_TYPE_MASK_". The name must be unique.
2. The value of the recallable object type. IANA will assign this number, and the request from the registrant will use TBD1 instead of an actual value. IANA MUST use a whole number which can be no higher than $2^{32}-1$, and should be the next available value. The value must be unique. A Designated Expert must be used to ensure that when the name of the recallable type and its value are added to the NFSv4 XDR description [\[12\]](#), the result continues to be a valid XDR description.
3. The Standards Track RFC(s) that describe the recallable object type. If the RFC(s) have not yet been published, the registrant

will use RFCTBD2, RFCTBD3, etc. instead of an actual RFC number.

4. How the RFC introduces the recallable object type. This is indicated by a single US-ASCII value. If the value is N, it means a minor revision to the NFSv4 protocol. If the value is L, it means a new pNFS layout type. Other values can be used with IESG Approval.
5. The minor versions of NFSv4 that are allowed to the use the recallable object type. While these are numeric values, IANA will not allocate and assign them; the author of the relevant RFCs with IESG Approval assigns these numbers. Each time there is new minor version of NFSv4 approved, a Designated Expert should review the registry to make recommended updates as needed.

[22.3.1.](#) Initial Registry

The initial registry is in Table 16. Note that next available value is five.

Recallable Object Type Name	Value	RFC	How	Minor Versions
RCA4_TYPE_MASK_RDATA_DLG	0	RFCTBD10	N	1
RCA4_TYPE_MASK_WDATA_DLG	1	RFCTBD10	N	1
RCA4_TYPE_MASK_DIR_DLG	2	RFCTBD10	N	1
RCA4_TYPE_MASK_FILE_LAYOUT	3	RFCTBD10	N	1
RCA4_TYPE_MASK_BLK_LAYOUT	4	RFCTBD20	L	1
RCA4_TYPE_MASK_OBJ_LAYOUT_MIN	8	RFCTBD30	L	1
RCA4_TYPE_MASK_OBJ_LAYOUT_MAX	9	RFCTBD30	L	1

Table 16: Initial Recallable Object Type Assignments

[22.3.2.](#) Updating Registrations

The update of a registration will require IESG Approval on the advice of a Designated Expert.

[22.4.](#) Layout Types

IANA will create a registry called the "pNFS Layout Types Registry".

All assignments to the registry are made on a Standards Action basis, with Expert Review required.

Layout types are 32 bit numbers. The value zero is Reserved. Values in the range 0x80000000 to 0xFFFFFFFF inclusive are for Private Use. IANA will assign numbers from the range 0x00000001 to 0x7FFFFFFF inclusive.

The registry is a list of assignments, each containing five fields.

1. The name of the layout type. This name must have the prefix: "LAYOUT4_". The name must be unique.
2. The value of the layout type. IANA will assign this number, and the request from the registrant will use TBD1 instead of an actual value. The value assigned must be unique. A Designated Expert must be used to ensure that when the name of the layout type and its value are added to the NFSv4.1 layouttype4 enumerated data type in the NFSv4.1 XDR description ([\[12\]](#)), the

result continues to be a valid XDR description.

3. The Standards Track RFC(s) that describe the notification. If the RFC(s) have not yet been published, the registrant will use RFCTBD2, RFCTBD3, etc. instead of an actual RFC number. Collectively, the RFC(s) must adhere to the guidelines listed in [Section 22.4.3](#).
4. How the RFC introduces the notification. This is indicated by a single US-ASCII value. If the value is N, it means a minor revision to the NFSv4 protocol. If the value is L, it means a new pNFS layout type. Other values can be used with IESG Approval.
5. The minor versions of NFSv4 that are allowed to use the notification. While these are numeric values, IANA will not allocate and assign them; the author of the relevant RFCs with IESG Approval assigns these numbers. Each time there is new

minor version of NFSv4 approved, a Designated Expert should review the registry to make recommended updates as needed.

22.4.1. Initial Registry

The initial registry is in Table 17.

Layout Type Name	Value	RFC	How	Minor Versions
LAYOUT4_NFSV4_1_FILES	0x1	RFCTBD10	N	1
LAYOUT4_OSD2_OBJECTS	0x2	RFCTBD30	L	1
LAYOUT4_BLOCK_VOLUME	0x3	RFCTBD20	L	1

Table 17: Initial Layout Type Assignments

22.4.2. Updating Registrations

The update of a registration will require IESG Approval on the advice of a Designated Expert.

22.4.3. Guidelines for Writing Layout Type Specifications

The author of a new pNFS layout specification must follow these steps to obtain acceptance of the layout type as a Standards Track RFC:

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 loh_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 storage devices.
 - * Describe whether revocation of layouts is supported.
 - * At a minimum, describe the methods of recovery from:
 1. Failure and restart for client, server, storage device.
 2. Lease expiration from perspective of the active client, server, storage device.
 3. Loss of layout state resulting in fencing of client access to storage devices (for an example, see [Section 12.7.3](#)).
 - * Include an IANA considerations section, will in turn include:
 - + A request to IANA for a new layout type per [Section 22.4](#).
 - + A list of requests to IANA for any new recallable object types for `CB_RECALL_ANY`; each entry is to presented in the form described in [Section 22.3](#).
 - + A list of requests to IANA for any new notification values for `CB_NOTIFY_DEVICEID`; each entry is to presented in the form described in [Section 22.2](#).
 - * 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.

[22.5.](#) Path Variable Definitions

This section deals with the IANA considerations associated with the variable substitution feature for location names as described in [Section 11.10.3](#). As described there, variables subject to substitution consist of a domain name and a specific name within that domain, with two separated by a colon. There are two sets of IANA considerations here:

1. The list of variable names.
2. For each variable name, the list of possible values.

Thus, there will be one registry for the list of variable names, and possibly one registry for listing the values of each variable name.

[22.5.1.](#) Path Variables Registry

IANA will create a registry called the "NFSv4 Path Variables Registry".

[22.5.1.1.](#) Path Variable Values

Variable names are of the form "\${", followed by a domain name, followed by a colon (":"), followed by a domain-specific portion of the variable name, followed by "}". When the domain name is "ietf.org" all variables names must be registered with IANA on a Standards Action basis, with Expert Review required. Path variables with registered domain names neither part of nor equal to ietf.org are assigned on a Hierarchical Allocation basis (delegating to the domain owner) and thus of no concern to IANA, unless the domain owner chooses to register a variable name from his domain. If the domain owner chooses to do so, IANA will do so on a First Come First Serve basis. To accommodate registrants who do not have their own domain, IANA will accept requests to register variables with the prefix "\${FCFS.ietf.org:" on a First Come First Served basis. Assignments on a First Come First Basis do not require Expert Review, unless the registrant also wants IANA to establish a registry for the values of the registered variable.

The registry is a list of assignments, each containing three fields.

1. The name of the variable. The name of this variable must start with a "\${" followed by a registered domain name, followed by ":", or it must start with "\${FCFS.ietf.org". The name must be no more than 64 UTF-8 characters long. The name must be unique.
2. For assignments made on Standards Action basis, the Standards Track RFC(s) that describe the variable. If the RFC(s) have not yet been published, the registrant will use RFCTBD1, RFCTBD2, etc. instead of an actual RFC number. Note that the RFCs do not have to be a part of a NFS minor version. For assignments made on a First Come First Serve basis, an explanation (consuming no more than 1024 bytes, or more if IANA permits) of the purpose of the variable. A reference to the explanation can be substituted.
3. The point of contact, including an email address. The point of contact can consume up to 256 bytes (or more if IANA permits). For assignments made on a Standards Action basis, the point of contact is always IESG.

[22.5.1.1.1](#). Initial Registry

The initial registry is in Table 18.

Variable Name	RFC	Point of Contact
\${ietf.org:CPU_ARCH}	RFCTBD10	IESG
\${ietf.org:OS_TYPE}	RFCTBD10	IESG
\${ietf.org:OS_VERSION}	RFCTBD10	IESG

Table 18: Initial List of Path Variables

IANA will need to create registries for the values of the variable names \${ietf.org:CPU_ARCH} and \${ietf.org:OS_TYPE}. See [Section 22.5.2](#) and [Section 22.5.3](#).

For the values of the variable \${ietf.org:OS_VERSION}, no registry is needed as the specifics of the values of the variable will vary with the value of \${ietf.org:OS_TYPE}. Thus values for \${ietf.org:OS_VERSION} are on a Hierarchical Allocation basis and are of no concern to IANA.

Internet-Draft

NFSv4.1

September 2008

[22.5.1.1.2.](#) Updating Registrations

The update of an assignment made on a Standards Action basis will require IESG Approval on the advice of a Designated Expert.

The registrant can always updated the point of contact of an assignment made on a First Come First Serve basis. Any other update will require Expert Review.

[22.5.2.](#) Values for the `${ietf.org:CPU_ARCH}` Variable

IANA will create a registry called the "NFSv4 `${ietf.org:CPU_ARCH}` Value Registry".

Assignments to the registry are made on a First Come First Serve basis. The zero length value of `${ietf.org:CPU_ARCH}` is Reserved. Values with a prefix of "PRIV" are Reserved for Private Use.

The registry is a list of assignments, each containing three fields.

1. A value of the `${ietf.org:CPU_ARCH}` variable. The value must be 1 to 32 UTF-8 characters long. The value must be unique.
2. An explanation (consuming no more than 1024 bytes, or more if IANA permits) of what CPU architecture the value denotes. A reference to the explanation can be substituted.
3. The point of contact, including an email address. The point of contact can consume up to 256 bytes (or more if IANA permits).

[22.5.2.1.](#) Initial Registry

There is no initial registry.

[22.5.2.2.](#) Updating Registrations

The registrant is free to update the assignment, i.e. change the explanation and/or point of contact fields.

[22.5.3.](#) Values for the `${ietf.org:OS_TYPE}` Variable

IANA will create a registry called the "NFSv4 `${ietf.org:OS_TYPE}` Value Registry".

Assignments to the registry are made on a First Come First Serve basis. The zero length value of `${ietf.org:OS_TYPE}` is Reserved. Values with a prefix of "PRIV" are Reserved for Private Use.

Shepler, et al.

Expires March 9, 2009

[Page 597]

Internet-Draft

NFSv4.1

September 2008

The registry is a list of assignments, each containing three fields.

1. A value of the `${ietf.org:OS_TYPE}` variable. The value must be 1 to 32 UTF-8 characters long. The value must be unique.
2. An explanation (consuming no more than 1024 bytes, or more if IANA permits) of what CPU architecture the value denotes. A reference to the explanation can be substituted.
3. The point of contact, including an email address. The point of contact can consume up to 256 bytes (or more if IANA permits).

[22.5.3.1.](#) Initial Registry

There is no initial registry.

[22.5.3.2.](#) Updating Registrations

The registrant is free to update the assignment, i.e. change the explanation and/or point of contact fields.

[23.](#) References

[23.1.](#) Normative References

- [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.
- [2] Eisler, M., "XDR: External Data Representation Standard", STD 67, [RFC 4506](#), May 2006.
- [3] Srinivasan, R., "RPC: Remote Procedure Call Protocol

Specification Version 2", [RFC 1831](#), August 1995.

- [4] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", [RFC 2203](#), September 1997.
- [5] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism Version 2", [RFC 4121](#), July 2005.
- [6] Eisler, M., "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM", [RFC 2847](#), June 2000.
- [7] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", [RFC 2743](#), January 2000.

Shepler, et al.

Expires March 9, 2009

[Page 598]

Internet-Draft

NFSv4.1

September 2008

- [8] Talpey, T. and B. Callaghan, "Remote Direct Memory Access Transport for Remote Procedure Call", [draft-ietf-nfsv4-rpcrdma-08](#) (work in progress), April 2008.
- [9] Talpey, T., Callaghan, B., and I. Property, "NFS Direct Data Placement", [draft-ietf-nfsv4-nfsdirect-08](#) (work in progress), April 2008.
- [10] Recio, P., Metzler, B., Culley, P., Hilland, J., and D. Garcia, "A Remote Direct Memory Access Protocol Specification", [RFC 5040](#), October 2007.
- [11] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", [RFC 2104](#), February 1997.
- [12] Shepler, S., Eisler, M., and D. Noveck, "NFSv4 Minor Version 1 XDR Description", [draft-ietf-nfsv4-minorversion1-dot-x-08](#) (work in progress), Aug 2008.
- [13] Eisler, M., "IANA Considerations for RPC Net Identifiers and Universal Address Formats", [draft-ietf-nfsv4-rpc-netid-03](#) (work in progress), Aug 2008.
- [14] International Organization for Standardization, "Information Technology - Universal Multiple-octet coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane",

ISO Standard 10646-1, May 1993.

- [15] Alvestrand, H., "IETF Policy on Character Sets and Languages", [BCP 18](#), [RFC 2277](#), January 1998.
- [16] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", [RFC 3454](#), December 2002.
- [17] Hoffman, P. and M. Blanchet, "Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)", [RFC 3491](#), March 2003.
- [18] Schaad, J., Kaliski, B., and R. Housley, "Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", [RFC 4055](#), June 2005.
- [19] National Institute of Standards and Technology, "Cryptographic Algorithm Object Registration", December 2005.

Shepler, et al.

Expires March 9, 2009

[Page 599]

Internet-Draft

NFSv4.1

September 2008

[23.2.](#) Informative References

- [20] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", [RFC 3530](#), April 2003.
- [21] Callaghan, B., Pawłowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", [RFC 1813](#), June 1995.
- [22] Eisler, M., "NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5", [RFC 2623](#), June 1999.
- [23] Juszczak, C., "Improving the Performance and Correctness of an NFS Server", USENIX Conference Proceedings , June 1990.
- [24] Reynolds, J., "Assigned Numbers: [RFC 1700](#) is Replaced by an On-line Database", [RFC 3232](#), January 2002.

- [25] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", [RFC 1833](#), August 1995.
- [26] Werme, R., "RPC XID Issues", USENIX Conference Proceedings , February 1996.
- [27] Nowicki, B., "NFS: Network File System Protocol specification", [RFC 1094](#), March 1989.
- [28] Bhide, A., Elnozahy, E., and S. Morgan, "A Highly Available Network Server", USENIX Conference Proceedings , January 1991.
- [29] Halevy, B., Welch, B., and J. Zelenka, "Object-based pNFS Operations", [draft-ietf-nfsv4-pnfs-obj-09](#) (work in progress), June 2008.
- [30] Black, D., Fridella, S., and J. Glasgow, "pNFS Block/Volume Layout", [draft-ietf-nfsv4-pnfs-block-09](#) (work in progress), June 2008.
- [31] Callaghan, B., "WebNFS Client Specification", [RFC 2054](#), October 1996.
- [32] Callaghan, B., "WebNFS Server Specification", [RFC 2055](#), October 1996.
- [33] Shepler, S., "NFS Version 4 Design Considerations", [RFC 2624](#), June 1999.

Shepler, et al.

Expires March 9, 2009

[Page 600]

Internet-Draft

NFSv4.1

September 2008

- [34] Simonsen, K., "Character Mnemonics and Character Sets", [RFC 1345](#), June 1992.
- [35] The Open Group, "Protocols for Interworking: XNFS, Version 3W, ISBN 1-85912-184-5", February 1998.
- [36] Floyd, S. and V. Jacobson, "The Synchronization of Periodic Routing Messages", IEEE/ACM Transactions on Networking 2(2), pp. 122-136, April 1994.
- [37] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M., and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)",

[RFC 3720](#), April 2004.

- [38] Snively, R., "Fibre Channel Protocol for SCSI, 2nd Version (FCP-2)", ANSI/INCITS 350-2003, Oct 2003.
- [39] Weber, R., "Object-Based Storage Device Commands (OSD)", ANSI/INCITS 400-2004, July 2004,
<<http://www.t10.org/ftp/t10/drafts/osd/osd-r10.pdf>>.
- [40] The Open Group, "The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition", 2004.
- [41] Callaghan, B., "NFS URL Scheme", [RFC 2224](#), October 1997.
- [42] Chiu, A., Eisler, M., and B. Callaghan, "Security Negotiation for WebNFS", [RFC 2755](#), January 2000.
- [43] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", [BCP 26](#), [RFC 5226](#), May 2008.

[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.

Initial drafts relating to multi-server namespace features, including the concept of referrals, were contributed by Dave Noveck, Carl Burnett, and Charles Fan with contributions from Ted Anderson, Neil

Shepler, et al.

Expires March 9, 2009

[Page 601]

Internet-Draft

NFSv4.1

September 2008

Brown, and Jon Haswell.

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 pNFS work was inspired by the NASD and OSD work done by Garth Gibson. Gary Grider has also been a champion of high-performance parallel I/O. Garth Gibson and Peter Corbett started the pNFS effort with a problem statement document for IETF that formed the basis for the pNFS work in NFSv4.1.

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, Peter Honeyman, and Stephen Fridella.

Fredric Isaman found several errors in draft versions of the ONC RPC XDR description of the NFSv4.1 protocol.

Audrey Van Belleghem provided, in numerous ways, essential coordination and management of the process of editing the specification drafts.

Richard Jernigan gave feedback on the file layout's striping pattern design.

Several formal inspection teams were formed to review various areas of the protocol. All the inspections found significant errors and room for improvement. NFSv4.1's inspection teams were:

- o ACLs, with the following inspectors: Sam Falkner, Bruce Fields, Rahul Iyer, Saadia Khan, Dave Noveck, Lisa Week, Mario Wurzl, and Alan Yoder.
- o Sessions, with the following inspectors: William Brown, Tom Doeppner, Robert Gordon, Benny Halevy, Fredric Isaman, Rick Macklem, Trond Myklebust, Dave Noveck, Karen Rochford, John Scott, and Peter Shah.
- o Initial pNFS inspection, with the following inspectors: Andy Adamson, David Black, Mike Eisler, Marc Eshel, Sam Falkner, Garth Goodson, Benny Halevy, Rahul Iyer, Trond Myklebust, Spencer

Shepler, and Lisa Week.

- o Global namespace, with the following inspectors: Mike Eisler, Dan Ellard, Craig Everhart, Fred Isaman, Trond Myklebust, Dave Noveck, Theresa Raj, Spencer Shepler, Renu Tewari, and Robert Thurlow.
- o NFSv4.1 file layout type, with the following inspectors: Andy Adamson, Marc Eshel, Sam Falkner, Garth Goodson, Rahul Iyer, Trond Myklebust, and Lisa Week.
- o NFSv4.1 locking and directory delegations, with the following inspectors: Mike Eisler, Pranoop Erasani, Robert Gordon, Saadia Khan, Eric Kustarz, Dave Noveck, Spencer Shepler, and Amy Weaver.
- o EXCHANGE_ID and DESTROY_CLIENTID, with the following inspectors: Mike Eisler, Pranoop Erasani, Robert Gordon, Benny Halevy, Fred Isaman, Saadia Khan, Rick Macklem, Spencer Shepler, and Brent Welch.
- o Final pNFS inspection, with the following inspectors: Andy Adamson, Mike Eisler, Mark Eshel, Sam Falkner, Jason Glasgow, Garth Goodson, Robert Gordon, Benny Halevy, Dean Hildebrand, Rahul Iyer, Suchit Kaura, Trond Myklebust, Anatoly Pinchuk, Spencer Shepler, Renu Tewari, Lisa Week, and Brent Welch.

A review team worked together to generate the tables of assignments of error sets to operations and make sure that each such assignment had two or more people validating it. Participating in the process were: Andy Adamson, Mike Eisler, Sam Falkner, Garth Goodson, Robert Gordon, Trond Myklebust, Dave Noveck, Spencer Shepler, Tom Talpey, Amy Weaver, and Lisa Week.

Lars Eggert provided valuable review and guidance.

Others who provided comments include: Jason Goldschmidt, James Lentini, Archana Ramani, Jim Rees, and Mahesh Siddheshwar.

[Appendix B](#). RFC Editor Notes

[RFC Editor: please remove this section prior to publishing this document as an RFC]

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCTBD10 with RFCxxxx where xxxx is the RFC number of this document]

[RFC Editor: prior to publishing this document as an RFC, please

Internet-Draft

NFSv4.1

September 2008

replace all occurrences of RFCTBD20 with RFCyyyy where yyyy is the RFC number of the document referenced in [30]]

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCTBD30 with RFCzzzz where zzzz is the RFC number of the document referenced in [29]]

Authors' Addresses

Spencer Shepler
Storspeed, Inc.
7808 Moonflower Drive
Austin, TX 78750
USA

Phone: +1-512-402-5811 ext 8530
Email: shepler@storspeed.com

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

Phone: +1-719-599-9026
Email: mike@eisler.com
URI: <http://www.eisler.com>

David Noveck
NetApp
1601 Trapelo Road, Suite 16
Waltham, MA 02454
USA

Phone: +1-781-768-5347
Email: dnoveck@netapp.com

Internet-Draft

NFSv4.1

September 2008

Full Copyright Statement

Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.