NFSv4 D. Noveck, Ed.
Internet-Draft NetApp
Obsoletes: 8881 (if approved) 19 August 2025
Intended status: Standards Track
Expires: 20 February 2026

Network File System (NFS) Version 4 Minor Version 1 Protocol
draft-ietf-nfsv4-rfc8881bis-00

Abstract

This document describes the Network File System (NFS) version 4 minor version 1, including features retained from the base protocol (NFS version 4 minor version 0, which is specified in RFC 7530) and protocol extensions made as part of Minor Version 1. The later minor version has no dependencies on NFS version 4 minor version 0, and was, until recently, documented as a completely separate protocol.

This document is part of a set of documents which collectively obsolete RFCs 8881 and 8434. In addition to many corrections and clarifications, it will rely on NFSv4-wide documents to substantially revise the treatment of protocol extension, internationalization, and security, superseding the descriptions of those aspects of the protocol appearing in RFCs 5661 and 8881.

Status of This Memo

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

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

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

This Internet-Draft will expire on 20 February 2026.

Table of Contents

1. Introduction to this Update

This document is intended to be the basis for a revised and updated specification of NFSv4.1. Unlike [RFC8881], which provided a limited-function update to [RFC5661], this document has a broader mandate and will do the following:

* Revise any text known to be wrong or otherwise inappropriate. Such text will not be retained as it has been merely because it is outside a limited pre-specified change scope.

  This includes changes in some errata reports with the status REJECTED, where there is a Working Group Consensus that change is necessary.

* Correct protocol defects, that, by their nature, can be addressed via a limited use of the extension mechanism described in Section 9 of [RFC8178].

  For more discussion of the handling of existing protocol defects, see Section 1.4. This discussion, which covers protocol defects addressed in NFSv4-wide documents in addition to this document, will focus on how compatibility issues are addressed when defects are corrected. It will also pay attention to the question of when XDR extensions are necessary as part of defect correction.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as specified in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The above differs from the corresponding statement in earlier specification versions ([RFC5661] and [RFC8881]) which only referred to [RFC2119]. For further discussion of this change, see Appendix B.3

In some cases, such keywords will appear in all capitals within quotations, direct or indirect, from earlier documents. In such cases, these terms are to be interpreted just as above, except where it is explicitly noted that such an interpretation is not to be inferred. In some cases, it might be that this document’s approach to the matter would not use those key words for reasons explained in the text. Such a shift might cause compatibility issues, if the previous keyword were actually relied upon but it also possible that it was not relied upon while the implications of that use were ignored for various reasons.

The reader should be aware that, as discussed in Appendix C.1, there are uses of the keywords listed above in RFCs 5661 and 8881 which might not have been appropriate, even though the interpretation specified above was intended when the text was written and submitted for publication. In some cases, the text in this document has been updated to correct the issue but it should be understood that not all such questionable uses have been addressed and that this state of affairs might continue to exist until a later draft of this document is submitted for publication.

1.2. The Changed Role of this Specification

Previous specifications for this minor version ([RFC5661], [RFC8881]) have purported to describe the protocol in its entirety, without reference to features common to all minor versions of NFS Version 4. In contrast, this update relies on a set of base documents describing common aspects of the NFSv4 protocol that applies to all minor versions.

* Rules for extensions and creation of new minor versions appear only in [RFC8178], unlike previously in which they appeared in the NFSv4.1 specification. This eliminates the unfortunate situation in which each minor version was allowed to create its own extension rules.

* Handling of internationalization-related matters (for all minor versions) is now discussed in its own document, which is expected to be an RFC derived from [I-D.ietf-nfsv4-internationalization].

  That document, based in large part on the handling of internationalization for NFSv4 minor version zero outlined in [RFC7530], has been extended to cover all minor versions and enhanced to fully support case-insensitive handling of internationalized file names.

  This corrects the unfortunate situation in which internationalization for minor version one and subsequent minor versions (in [RFC5661] and [RFC8881]) had never been implemented and could never have been implemented by NFS Version four clients and servers.

* Handling of core security-related matters for all NFSv4 minor versions will be consolidated in a set of documents that are expected to be RFCs derived from [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls].

  This shift is made necessary by the following issues, many of which are of long standing and make the continuation of previous approaches to these issues insupportable:

  - The lack of a substantial threat analysis in the Security Considerations section of any existing minor version specification.

  - The unfortunate designation of AUTH_SYS an "OPTIONAL means of authentication" which had the effect of obscuring the severe security problems with its common use. The use of "OPTIONAL" suggested that use of AUTH_SYS had no harmful consequences while the phrase "means of authentication" ignored the fact that no actual authentication took place.

  - The assumption that data confidentiality could be satisfactorily addressed as an occasionally-used optional facility.

  - The neglect of the need for dependable semantic description of the protocol’s authorization semantics.

    Earlier versions of NFS had avoided the need for such descriptions by relying on POSIX semantics. The addition of non-POSIX semantic elements, including named attributes and ACLs, interfered with that approach although the necessity was not recognized as NFSv4 was being formulated.

The availability of new transport-level security features such as those provided by RPC-with-TLS [RFC9289] provides a basis to correct many of the above issues.

In providing that sort of correction, we need to be careful not to declare existing implementations non-compliant post facto, while still providing adequate warning of the security consequences of continuing to use the NFS Version 4 protocol insecurely, as described in previous specifications.

1.3. Possibility of Compatability Issues

Because this document or other documents that are part of this specification update might make changes to matters previously discussed in [RFC8881], the possibility of compatability issues cannot be excluded.

However, the likelihood of such issues arising is expected to be low because:

* In many cases, descrptions within [RFC8881] differed from the existing implementations and change was necessary to a avoid directing implementers to create implementations that could not interoperate with existing implementations.

* There are cases in which attributes were specified to be OPTIONAL even though there were no good reasons to implement a server that did not otherwise provide support for them.

* In some cases, it was necessary to change the definition of OPTIONAL features that had never been implementented, often because problems in the existing specification made the features impossible to implement as defined. The absence of implementations makes it impossible for compatability problems to arise.

* In some cases, defects in the protocol were corrected as described in Section 9 of [RFC8178], by creating OPTIONAL protocol extensions. In such cases, implementations unaware of the extensions will function as they did before, while others have the option of using the extensions when interacting with an implementation supporting the extension.

Specific updates are dealt with in the the following subsections:

* Updates in RFCTBD10 are addressed in Section 1.3.1.

* Updates in new NFSv4-wide documents are discussed un Section 1.3.2.

* Updates in RFCTBD30 are addressed in Section 1.3.3.

1.3.1. Compatibility issues for RFCTBD10

This document introduces a number of potential incompatibilities that either cannot be realized or could only be realized in situations highly unlikely to exist. For discussion of the reasons such inompatibilities are higly unlikely to be problematic , see the rest of this section which also explains the needs for the changes.

One important example concerns the conversion of the attributes mode, owner, and owner_group from OPTIONAL to REQUIRED. Since the vast majority of servers support these attributes and the vast majority of clients use them, any incompatibilities would be very unlikely to exist.

If there did exist servers not supporting any of these attributes and clients capable of interacting with them, this unexpected situation would not result in any incompatibility. The only change would be that the client and the server would become non-compliant when they previously had been considered compliant. This change of status has no consequences for client-server compatibility.

Overall, the need for this change illustrates previous specifications' lack of serious attention to security issues, which it is part of the job of the NFsv4.1 respecification effort to correct.

Another important class of potential compatibility issues in which we had to change the specification derives from situations in which the previous text designate normal, often unavoidable, behavior as non-compliant:

* Previously, clients were required to wait forever for an RPC response while many (e.g. Linux) clients stopped waiting in the event of a control-C. This common behavior, formerly non-compliant, is now allowed but there is no incompatibility resulting from this change even though much existing behavior has gone from non-compliant to compliant.

* Previously the use of RoCE for NFS/RDMA had been disallowed (formally) but is now allowed. Even so, there are clients amd servers that do this and work together doing so and making them compliant raises no compatability issues.

* Previously retries were not allowed ("MUST NOT" was used) unless there was a connection drop. This was changed to a recommendation since there us no way this could be a "fundamental requirement of the specification" given that the reply cache was designed to avoid any negative consequences of retries. In any case there was no possibility of an incompatibility since the effect of the changes is only to make previously non-compliant clients compliant.

There are also other changes made to make specfication text match the actual implementations. Because care was used in making sure that the change reflected implemenation changes documented in erratta reports, we do not expect compatibility issues to arise. Some important examples concern changes to pNFS-related operations that have been documented in errata report and discussed within the group so that existing implemantations all follow the changed approach. These include the following:

* In LAYOUTCOMMIT4args, loca_offset and loca_length are present but not used. Previously, they had been used to effect partial layout commits, whic are no longer provided for.

* In struct notify_deviceid_change4, ndc_immediate is present but not used. Previously, this field had been used to indicate whether or not a device change notification was being done immediately rather than being delayed.

A number of new features that are highly unlikely to have been implemented have been respecified so that they are effectively implementable, Althoough the Working Group’s has limits on its ability to investigate the matter, in each case, it appears veery unlikely that implementations exist. See below for details. In the absence of suc implementations, we can expect new implementations to follow tis specification, avoiding compatibility issues.

* The persistent reply cache feature was added to NFSv4.1 in [RFC5661] but there is good reason to believe it has never been implemented based on that document or on [RFC8881] Although an implementer would have no obligation to inform the Working Group of the existence of any implementation, the nature of the current description makes it very unlikely that any such implementations exist.

  For reasons discussed in Section 8.1, the feature was essentially implementable as described, while it is unlikely that any attempting an approximate implementation would do so without consulting the Working Group.

As a result, there is no real possibility of compatibility issues arrising.

* The direcory delegation was added to NFSv4.1 in [RFC5661] but there has been no discussion of implementatiions even though the performance effects of the feature would be of interest to the Working Group, since this was feature included in NFSv4.1 because of performance concerns.

  As things turned out, there were discussions of NFSv4 performace on metadata-intensive workloads at Working Group meetings with the effect of directory delegations not being commented on leading to a strong belief that no such implementations existed.

  When the reasons for the presumed non-implementation were analyzed, changes were made to the specfication of this feature. While the probability of implemenations existing is non-zero, their possible existence will not give rise to compatibility issues due to the way in which changes were made.

  All of the XDR changes took the form of extensions to coreect defects as discussed in Section 9 of [RFC8178]. In addition the added requirement for delegation recall on certain changes to the directory will not lead to incompatibility since the seerver is free to recall delegation as it chooses. In the case of possible existing servers that fail to recall the delegation in such cases, they will become non-compliant. However, this fact is not troublesome as we will be going from a compliant server that does not work correctly to a non-compliant one, with the non-compliance being an important cue to fix the server.

1.3.2. Compatibility issues for NFSv4-wide Documents

For these three NFSv4-wide douments a major implementation rea was respecified because of major problem in the existing NFSv4.1 descriptions of these area. Because the relaement text was ntended to reflect the actual implementationm, the likelihood of incompatibilities is quite low:

* RFCTBD20 respecified internationalization in a fashion quite different from the way it is specified in [RFC5661] and [RFC8881].

  This major shift did not give rise to compatibility issues because the approach to internationalization presented in [RFC5661] and [RFC8881] had never been implemented. We can be sure of this because any attempt to implement this stringprep-based approach would have generated comments/questions due to the complexity involved in, for exampe, rejecting unassigned code points.

Existing implementations were compatible with the approach discussed in [RFC7530] and the treatment in RFCTBD20 maintained that compatibility.

* RFCTBD21 respecfied security in a fashion a markedly different in tone from the way it had been discussed in previous specifications. This included a different approach to the use of AUTH_SYS, more concern about issues related to the security of data in flight, and the inclusion of a discussion of authorization semantics.

  Despite these major differences, care has been taken to avoid the creation of incompatibilities. The new treatment avoided incompatibilities by avoiding any disallowing of previously allowed behavior, even in cases in which there might be grounds for doing so.

  For example, althoigh the use of AUTH_SYS in the clear has enough associated security issues that saying it MUST NOT or SHOULD NOT be used, this cannot be done for existing minor versions. To deal with this situation, its use is not disallowed or recommended against. However, unlike in previous specifications, implementers are given appropriate notice of the security issues resulting from use of AUTH_SYS in the clear.

  The addition of discussion of RPC-with-TLS is another important difference which cannot cause incompatibility issues. Despite the optionality of the use of this facility, it is important to discuss in connection with NFSv4 security and its handling of the security of data in flight.

* The new treatment of ACLs in RFCTBD22, while extensive, does not give rise to major incompatibilities because of the way we were forced to adapt to the previous approach while using a clearer explanatory framework that made the POSIX-oriented subset the basis of the description and expplicitly making the Windows-ACL-based extentions OPTIONAL.

  For the most part, this approach does not generate incompatibilities in that it moves formerly allowed behaviors (implicitly optional) to be formally OPTIONAL, without changing the set of allowed behaviors. However, with the addition of the OPTIONAL attribute Aclchoice, the client is able to determine what extensions to the base ACL mode have been implemented.

  The one possible formal imcompatibility arises from the three ACE mask btits corresponding to the POSIX R, W, and X bits. support for these are now REQUIRED wheras previously it had een suggested

tat they were effectively opuional along with all the other ACE mask bits. The chance of this being a problem is low since, we have not heard any discussion of server without support for all of these bits and te difficulty of adopting to POSIX demantics, of such implementations existed would certainly have led to Working Group discussion of the issue.

1.3.3. Compatibility issues for RFCTBD30

These changes can be divided into three groups:

* In many cases, the XDR changes consist of changes in the typedef naming structure, with no change to the stucture of messages ddescribed by the XDR. As a result, compatibility issues are not expected to arise.

* Many changes cannot create incompatibilities because the change cosists only to the names of typedefs. Such changes were necessary to clarify how naming interpretaion rules applied to various strings. Because all such strings have types that evaluate to opaque<>;, there can be no incompaibilties generated.

* In some cases, fields defined in various messages have become unused. Space is still reserved for these fields but the on-use is indicated in comments, so there is no XDR change to generate an incomptibility.

* All subsantive XDR changes take the form of XDR extensions as provided for by Section 9 of [RFC8178]. As a result, implementations unaware of the change will function as they dis previously, while those that are aware of extension have the option of adapting to the support or non-support by the peer implementation.

1.4. Addressing Protocol Defects

This section provides an overview of situations in which it was necessary to change the protocol to correct protocol issues that needed to be addressed going forward. Although all these issues can, from today’s perspective, be viewed as mistakes, it is not clear whether that description is appropriate for decisions made so long ago, under very different circumstances. In any case, those questions will not be addressed here.

The correction of protocol defects often gives rise to compatibility issues and their possible presence will be discussed below. In addition, the question of when it is appropriate to address such issues using the protocol extension mechanism described in [RFC8178]

needs to be considered. Section 9 of that document alludes to this possibility but we have to decide when defects are best addressed in that way.

These defects can be divided into two groups based on their origin.

* Defects that originated in minor version zero

  Many of these defects are addressed in the new NFSv4-wide documents ([I-D.dnoveck-nfsv4-security], [I-D.dnoveck-nfsv4-acls], and [I-D.ietf-nfsv4-internationalization]. For these defects , the greater change scope require more attention to compatibility issues. In addition, that greater scope limits the degree to which protocol extension can be used in providing a correction since that extension would need to be propagated to two non-extensible minor versions.

* Defects that originated in minor version one.

  A major source of defects was the result of the addition of a set of OPTIONAL features in v4.1, that have never been implemented, making it important to eliminate issues in the specification that have led to this situation.

  The presumptive non-implementation of these features will limit interoperability concerns. However, since we cannot be sure about the possible existence of implementations under development, we will try to provide for the possibility of interoperating with earlier implementations, even if that interoperation is hypothetical. Only in the case of features whose current specification makes implementation impossible can we ignore the possibility of interoperating with such implementations.

The following defects were addressed as part of this update effort:

1: Internationalization is being thoroughly respecified in the NFSv4-wide document [I-D.ietf-nfsv4-internationalization].

   There were no NFSv4.1 compatibility issues to deal with since the handling of internationalization approach mandated by [RFC8881] had never been implemented.

   Potential NFSv4.0 compatibility issues were very limited since the approach followed in [RFC7530] was continued and that approach had been used by all implementations.

In one particular case, there is a potential compatibility issue arising from the transition of one potential troubling server behavior from being discouraged using SHOULD NOT to being prohibited using MUST NOT. However, in view of the unlikelihood of ongoing use of the discouraged behavior, this has not been considered problematic.

The respecification of the fs_charset_cap attribute raises the possibility of within-NFS4.1 compatibility issues. However the very limited use of this attribute by clients combine with the lack of clarity in the previous definition makes it unlikely that the use of protocol extension to support previous uses would be justified.

2: Because the Persistent reply cache feature could not be implemented as described in [RFC8881], the entire area was respecified in Section 8.

The reasons for this respecification are discussed in Section 8.1.

Because the existing feature specification was unimplementable there were no compatibility issues to deal with.

No protocol extensions were needed since the small set of bits defined for the earlier feature could be coopted.

3: Security for all minor versions is being thoroughly respecified in the NFSv4-wide document [I-D.dnoveck-nfsv4-security]. In this discussion, issues related to authorization semantics and to ACLs are being dealt with separately.

This respecification was made necessary by the lack of threat analyses for all minor versions, the absence of any discussion of the security problems associated with the use of AUTH_SYS, and the half-hearted approach to the security of over-the-wire transmission in which transmission in the clear was the default and the provision of secure transmission was an option requiring per-fs configuration.

4: As part of the new handling of security, a more serious treatment of authorization semantics was necessary. As part of effecting this, the attributes mode, owner, and owner_group became REQUIRED, as it is impossible to effect security without them.

This change was not connected to the shift in terminology in which the attributes incorrectly described as "RECOMMENDED" became "OPTIONAL".

No significant compatibility issue are expected, since the existence of servers not supporting these attributes or of clients interacting with such servers, while possible theoretically, has to be considered extremely unlikely and none are known to the working group.

5: A number of gaps in the description of authorization semantics needed to be addressed. These include the lack of a clear description of authorization for operations on named attribute directories and potential use of the "sticky" bit in controlling authorization of file deletion.

These matters are being addressed within [I-D.dnoveck-nfsv4-security] where they are being tracked as Consensus Items #66 and #6 respectively.

Working group consideration of the security document will involve resolving those two Consensus Items, as well as others.

6: The handling of ACLs for all minor versions is being thoroughly respecified in the NFSv4-wide document [I-D.dnoveck-nfsv4-acls].

Such a respecification was made necessary by the profound underspecification of the ACL feature that arose from a misguided attempt to support two very different approaches to the provision of ACLs. The problems posed by the different semantics of these two were never clearly addressed since it was erroneously assumed that semantic description could be avoided. As a result, potential interoperability was compromised since there was no way for the client to determined what ACL-based facilities were supported by a particular server, given that the specification treated these differences as if they were quality-of-implementation issues.

The development of [I-D.dnoveck-nfsv4-acls] has included a respecification of the area in which support for a subset of draft POSIX ACLs, termed UNIX ACLs, was the core and the various additions to that core were considered additional OPTIONAL features. These included the features that motivated the extensions in the NFSv4 ACL model and further accommodations for the semantics of the draft POSIX ACL model.

The development of [I-D.dnoveck-nfsv4-acls] has involved use of protocol extension within NFSv4.1 in addition to necessary structural changes that did not involve XDR changes.

The development of [I-D.dnoveck-nfsv4-acls] to support the rfc8881bis effort will most likely be limited to providing interoperability for those using the facilities within the UNIX ACL core or within the draft POSIX acl model. Interoperability for features beyond that set is likely to be delayed to later ACL bis, while the deletion of unneeded proposed features will have to wait for a later minor version, e.g., NFSv4.3.

7: It has been necessary to define a new read-only per-fs OPTIONAL attribute that will allow clients to determine which of the OPTIONAL extensions to the core UNIX ACL model are supported by the server.

While this was essential to make the NFSv4 extensions usable, it also has a critical role in making POSIX ACL support available within NFSv4, albeit with some client mapping/filtering,

8: The defects described in Appendix C.2.1 needed to be addressed together, in connection with making it clear that the term "Exactly-once Semantics" ignored the fact that there were valid reasons to give up on requests which could leave them unexecuted.

This change did not give rise to compatibility issues since the specification was changed to match existing implementations, and these are expected to remain as they are.

9: The inadvertent prohibition of the use of RoCE in implementing NFSv4.1 using RPC-over-RDMA was removed.

This is another case in which compatibility issue are not expected because the spec has been changed to match existing implementations.

10: The corrections discussed in Appendix C.2.3 had to be made since most of the worries expressed within it were the result of misunderstandings.

Although no compatibility issues are expected we will need to review the changes and reach consensus on them.

11: There were a number of issues in the earlier specification of the directory delegation feature that need to be addressed to enable implementations of this needed feature to be produced. Given the lack of implementation during the long period since they were introduced in [RFC5661] many years ago.

While a significant part of the problems could be ascribed to clarity issues, there were also a set of defects, some of which required protocol extensions, as provided for in Section 9 of [RFC8178].

The defects which contributed substantially to this long-lasting lack of implementation includes the failure to fully address authorization issues for the use of cached directory data, implementability issues regarding the maintenance of cached attribute data, and the assumption that clients could maintain the cached directory contents only in the same format as used by the server. For more discussion of these defects, see Appendix C.2.2.

These issues were addressed by a major rewrite of Section 15.9 in which protocol extension was necessary, including the addition of new values to the enum notify_type4. In addition, there are complementary changes made to Section 23.39 and Section 25.4) and to operations that might result in notifications being sent.

Although no client and server implementations of this feature are known to exist, the possibility of them existing cannot be excluded. As a result, the revised specification takes care to deal appropriately with such hypothetical implementations, and to not prohibit their use unless that is necessary to avoid unacceptable system behavior.

12: Change in the recommendations regarding handling of numeric strings to represent users and groups.

Formerly considered troublesome even in the AUTH_SYS case despite the fact that there is no explanation given as to how to effect mapping between numeric ids and strings. Instead, it is assumed that client and server will somehow agree to do this without the specification making it possible or giving a convincing reason that such mapping is needed.

Of the above, only the items 7, 8, 9, 10, and 14 required protocol extension to resolve. All will to be incorporated in RFCTBD30.

2. Introduction to this Minor Version Specification

2.1. 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 now described in [RFC7530], as modified by [RFC7931] and [RFC8587]. Minor version 1 follows the guidelines for minor versioning presented in [RFC8178].

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, such as the use of the sessions model.

This minor version adds a considerable number of new operations including some that are not OPTIONAL and makes a number of NFSv4.0 operations MANDATORY to NOT implement. As a result, the vast majority of NFSv4.0 requests are not valid in NFSv4.1 and vice versa. While clients and server that support both minor versions are common, such implementations treat the two versions as distinct protocols sharing a substantial common heritage.

2.2. Scope of This Document

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

* describe the NFSv4.0 protocol, except where needed to contrast it with NFSv4.1.

* modify the specification of the NFSv4.0 protocol.

* clarify the NFSv4.0 protocol.

2.3. NFSv4 Goals

The NFSv4 protocol is a further revision of the NFS protocol defined already by NFSv3 [RFC1813]. It retains the essential characteristics of previous versions: easy recovery; independence of transport protocols, operating systems, and file systems; simplicity; and good performance. NFSv4 had the following goals:

* Improved access and good performance on the Internet.

The protocol is designed to transit firewalls easily, perform well where latency is high and bandwidth is low, and scale to very large numbers of clients per server.

* Strong security with facilities for negotiation of security handling built into the protocol.

  The protocol has built 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 has provisions requiring or recommending client and server support for a minimal set of security schemes.

  The protocol now takes advantage of the ability of RPC to make confidentiality available by using TLS-based encryption on connections to be used for NFSv4.1, which may limit the need for negotiation regarding facilities such as privacy.

* Good cross-platform interoperability.

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

* Designed for protocol extensions via minor versioning.

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

  When extensions are OPTIONAL, they can be added to an existing extensible minor version.

2.4. NFSv4.1 Goals

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

* To correct significant structural weaknesses and oversights discovered in the base protocol.

* To add clarity and specificity to areas left unaddressed or not addressed in sufficient detail in the base protocol. However, as stated in Section 2.2, it is not a goal to clarify the NFSv4.0 protocol in the NFSv4.1 specification.

* To add specific features based on experience with the existing protocol and recent industry developments.

* To provide protocol support to take advantage of clustered server deployments including the ability to provide scalable parallel access to sets of files distributed among multiple servers, using a range of data access protocols.

  This parallel access may involve striping of single files among a set of servers within the cluster.

  Alternatively, parallel access may be provided by distributing unstriped files within the cluster allowing each client to contact the server holding each particular file directly.

2.5. General Definitions

The following definitions provide an appropriate context for the reader.

Byte: In this document, a byte is an octet, i.e., a datum exactly 8 bits in length.

Client: The client is the entity that accesses the NFS server's resources. The client may be an application that contains the logic to access the NFS server directly. The client may also be a 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 byte-range locking, the client is also the entity that maintains a set of locks on behalf of one or more applications. This client is responsible for providing crash or failure recovery for those locks it manages, in order to deal with the possibility of server reboot.

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

Client ID: The client ID is a 64-bit quantity used as a unique, short-hand reference to a client-supplied client owner, consisting of a verifier and a client owner id. The server is responsible for supplying the client ID.

Client Owner: The client owner includes a unique string, the client owner id, opaque to the server, that identifies a client. It also includes a verifier to enable successive instances of the same client to be distinguished. 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. See Section 5.5 for more detail.

File System: A file system is the collection of objects accessed using a particular server (as identified by the major identifier of a server owner, which is defined later in this section) that share the same value of the fsid attribute (see Section 11.12.1.9).

Lease: A lease is an interval of time defined by the server for which the client is granted locks. At the end of a lease period, unrecallable locks may be revoked if the lease has not been extended. Such a lock must be revoked if a conflicting lock has been granted after the lease interval. Revocation of unrecallable locks within the lease interval is expected to be an unusual event and clients normally expect such revocations to be rare.

A server grants a client a single lease for all of its associated locking state. Recallable locks such as layouts and delegations can be revoked within the lease period and are generally not affected by the state of the lease.

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.

Secret State Verifier (SSV): The SSV is a unique secret key shared between a client and server. The SSV serves as the secret key for an internal (that is, internal to NFSv4.1) Generic Security Services (GSS) mechanism (the SSV GSS mechanism; see Section 7.9). The SSV GSS mechanism uses the SSV to compute message integrity code (MIC) and Wrap tokens. See Section 7.8.3 for more details on how NFSv4.1 uses the SSV and the SSV GSS mechanism.

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

Stable Storage: Stable storage is storage from which data stored by an NFSv4.1 server can be recovered without data loss from multiple power failures (including cascading power failures, that is, several power failures in quick succession), operating system failures, and/or hardware failure of components other than the storage medium itself (such as disk, nonvolatile RAM, flash memory, etc.).

Some examples of stable storage that are allowable for an NFS server include:

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

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

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

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

Stateid: A stateid is a 128-bit quantity returned by a server that uniquely defines the open and locking states provided by the server for a specific open-owner or lock-owner/open-owner pair for a specific file and type of lock.

Verifier: A verifier is a 64-bit quantity that is changed to indicate that a corresponding change on one of the peers has occurred requiring the other peer to adjust to possibility of change. There are a number of 64-bit quantities identified as verifiers:

* In many cases, a verifier is a 64-bit quantity generated by the server that the client can use to determine if the client has restarted and potentially lost writes that had not been reliably committed to stable storage.

* Another type of verifier is used to indicate whether the server mapping between directory cookies and directory entries has changed, requiring the client to restart directory interrogations that normally may be continued across multiple READDIR requests.

* As described above, client owners include a verifier, allowing the server to determine when a client reboot has occurred

The Secret State Verifier is not a verifier in the sense given in this definition.

2.6. Overview of NFSv4.1 Features

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 who 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 External Data Representation (XDR) and Remote Procedure Call (RPC) protocols as described in [RFC4506] and [RFC5531]. 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 features added in minor version 1 from those present in the base protocol but will treat NFSv4.1 as a unified whole. See Section 4 for a summary of the differences between NFSv4.0 and NFSv4.1.

2.7. 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 [RFC4506] and [RFC5531], as extended by [RFC9289]) to provide TLS-based encryption and client-host authentication. NFSv4.1 security. A description of the basics of NFv4.1 security will appear in an NFSv4-wide security document, to be derived from [I-D.dnoveck-nfsv4-security].

NFSv4.1 introduces parallel access (see Section 2.8.2), through the use of pNFS. The security framework described above is significantly modified by the introduction of pNFS (see Section 17.9), because of the addition of additional actors and because data access sometimes does not rely on RPC for principal identification/authentication. The appropriate handling depends on the data access protocol used (see Section 17.2.5) which depends in turn on the layout type (see Section 17.2.7.) The sections 17.9.1 through 17.9.3 discuss the security implications of using different sorts of data access protocols.

2.8. Protocol Structure

2.8.1. Core Protocol

Unlike NFSv3, which relied on a series of ancillary protocols (e.g., NLM, NSM (Network Status Monitor), MOUNT), within all minor versions of NFSv4 a single RPC protocol is available to make requests to the server. Facilities that had been separate protocols, such as locking, are now integrated within a single unified protocol, although, to implement pNFS, different data access protocols may also be used.

2.8.2. Parallel Access

Minor version 1 supports high-performance data access to a clustered server implementation by enabling a separation of metadata access and data access, with the latter able to be 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.

Because the protocols used for parallel data access are not necessarily RPC-based, the RPC-based security model (Section 2.7) is impacted (see Section 17.9). The degree of impact varies with the protocol (see Section 17.2.5) used for data access, and can be as low as zero for some RPC-based data access protocols (see Section 18.13).

2.9. File System Model

The general file system model used for the NFSv4.1 protocol is the same as for previous minor versions of NFSv4. The server file system is hierarchical with the regular files contained within being treated as opaque byte streams. File names MAY be restricted to UTF-8-encoded strings of Unicode characters or treated as opaque. In addition, for some file systems, name handling MAY reflect the UTF-8 canonical equivalence relation, and in some cases, case-based equivalence relations as well.

The NFSv4.1 protocol does not rely on 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 is responsible for providing any necessary pseudo file systems to bridge any gaps that arise due to unexported portions of the server-local name space that are between exported file systems.

2.9.1. 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 and bever to be reused after that. In addition, it provides support to allow servers to provide filehandles with more limited validity guarantees, referred to as volatile filehandles.

2.9.2. Numbered File Attributes

The NFSv4.1 protocol has a rich and extensible file object attribute structure in which each attribute is assigned an attribute number. The set of such attributes can be usefully divided in a number of ways, in order to provide helpful context for server implementers choosing to implement or not implement particular attributes which are not REQUIRED and for client implementers deciding how to deal with non-support of particular attributes which are not REQUIRED.

* Attributes differ as to their scope, with only a subset applicable to a particular file object, while others apply to an entire file system.

  As a practical matter, attributes applicable to a single file object, often require support within the file system proper. While this functionality is most often provided by a file system created initially for local access and only later adapted to remote use through an NFSv4.1 server, there are also file systems purpose-built for remote access.

Attributes applicable to an entire file system do not typically require support within the file system proper. One possible exception is when such attributes can be set to indicate the client's desire for some particular feature that inherently require file system support.

Note that, in all these cases, applications are most likely to be adapted to features that can be accessed using existing file access facilities. As a result, implementers are unlikely to devote efforts to implementation of OPTIONAL features and attributes which require interactions with applications while being more open to attributes usable by the client and server to communicate to optimize data flows without requiring application involvement.

* Attributes differ as to their mutability characteristics, including whether the attribute is question can be modified explicitly by the client and whether the attribute modification happens as a result of performing other operations, such as modifying a file or directory

* In this update of the NFSv4.1 specification, the details for handling authorization-related attributes are the responsibility of the NFSv4-wide security documents expected to be derived from [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls].

  Although the detailed categorization of such attributes will be the responsibility of the security documents, this document will, in Section 11.3, provide a brief summary and make clear that some of these are more necessary than others, and that they all cannot be reasonably treated as having the being of the same class regarding the need for server support.

Attributes are divided into a number of classes based on the protocol's requirements/recommendations for server implementation and the client's expected response to a server's non-support of those attributes. This categorization differs from that appearing previously for a number of reasons with the specific differences explained in Section 11.2.

* A significant number of attributes are described as REQUIRED so that servers MUST provide support for them.

These include the same set of attributes described in this way in RFCs [RFC7530] and [RFC8881]. In addition, there are a set of authorization-related attributes that need to be included in this group for reasons explained in [I-D.dnoveck-nfsv4-security]. The inclusion of all these attributes is discussed in more detail in Section 11.4.

* Many attributes are truly OPTIONAL, even though such attributes have been erroneously categorized as "RECOMMENDED" in the past. These attributes are discussed in more detail in Section 11.5.

  For a more detailed explanation of these shifts in terminology, see Section 11.2.

* It appears necessary to designate certain authorization-related attributes as Experimental.

  These attributes are discussed in more detail in Sections 11.3 and 11.6.

Descriptions of each specific attribute appears in the following places:

* Those which are authorization-related are described in [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls].

* Others are described sections 11.14 through 11.17.5

These descriptions can be found using the three sources described below:

* The list of Authorization-related attributes appears in Section 11.18

* The list of REQUIRED Attributes appears in Table 4 within Section 11.10.

* The list of attributes which are not REQUIRED) appears in Table 5 within Section 11.11.

"Named attributes", despite this designation, which will be retained, differ substantially from file attributes per se and are explained in Section 2.9.3. All such attributes are OPTIONAL as is the entire named attribute feature and such attributes are not part of the categorization above.

2.9.3. Named Attributes

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 were intended to be used by client applications as a method to associate application-specific data with a regular file or directory. Servers providing support for the named attribute feature, which is OPTIONAL, allow a number of opaque byte stream to be associated with a directory or file. This feature allows applications to define extended attributes which could be opened, read and written just as files are.

For further information about the use of named attributes, see Section 11.7

2.9.4. 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 nondisruptive transfer of support for individual file systems between servers. They are all based upon attributes that allow one file system to specify alternate, additional, and new location information that specifies how the client may access that file system.

These attributes can be used to provide for individual active file systems:

* Alternate network addresses to access the current file system instance.

* The locations of alternate file system instances or replicas to be used in the event that the current file system instance becomes unavailable.

These file system location attributes may be used together with the concept of absent file systems, in which a position in the server namespace is associated with locations on other servers without there being any corresponding file system instance on the current server. For example,

* These 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.

* These attributes may be provided when a previously present file system becomes absent. This allows nondisruptive migration of file systems to alternate servers.

3. Locking Facilities

As mentioned previously, NFSv4.1 is a single protocol that 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:

* Share reservations as established by OPEN operations.

* Byte-range locks.

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

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

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

All non-recallable 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 the client’s lease is not promptly renewed, the client’s 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.

Recallable locks are subject to revocation irrespective of lease state. Servers often need to revoke such locks when recalling them does not result in their prompt return.

4. Differences from NFSv4.0

The following summarizes the major differences between minor version 1 and the base protocol:

* Implementation of the sessions model (Section 7).

* Parallel access to data (Section 17).

* Addition of the RECLAIM_COMPLETE operation to better structure the lock reclamation process (Section 23.51).

* Enhanced delegation support as follows.

  - Delegations on directories and other file types in addition to regular files (Section 23.39, Section 23.49).

  - Operations to optimize acquisition of recalled or denied delegations (Section 23.49, Section 25.5, Section 25.7).

  - Notifications of changes to files and directories (Section 23.39, Section 25.4).

  - A method to allow a server to indicate that 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 25.6).

* Attributes can be set atomically during exclusive file create via the OPEN operation (see the new EXCLUSIVE4_1 creation method in Section 23.16).

* Open files can be preserved if removed and the hard link count ("hard link" is defined in an Open Group [hardlink] standard) goes to zero, thus obviating the need for clients to rename deleted files to partially hidden names, a technique colloquially called "silly rename" (see the new OPEN4_RESULT_PRESERVE_UNLINKED reply flag in Section 23.16).

* Improved compatibility with Microsoft Windows for Access Control Lists (sacl and dacl attributes, acl inheritance).

* Data retention (Section 11.17).

* Identification of the implementation of the NFS client and server (Section 23.35).

* Support for notification of the availability of byte-range locks (see the new OPEN4_RESULT_MAY_NOTIFY_LOCK reply flag in Section 23.16 and see Section 25.11).

* In NFSv4.1, LIPKEY and SPKM-3 are not required security mechanisms [RFC2847].

5. Core Infrastructure

5.1. Introduction

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

5.2. 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 [RFC5531] and [RFC4506]. The transport-level encryption and client-host authentication facilities described in [RFC9289] can also be used.

5.3. RPC-Based Security

In addition to the above, as discussed in Section 5.3.1, 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, can be implemented as well, although the security implications of doing so need to be carefully considered, particularly when the client host is not itself authenticated. In particular, it is RECOMMENDED by rpc-tls [RFC9289] that AUTH_SYS not be used when client host authentication is not in effect.

5.3.1. RPCSEC_GSS and Security Services

RPCSEC_GSS [RFC2203] uses the functionality of GSS-API [RFC2743]. This allows for the use of various security mechanisms by the RPC layer without the additional implementation overhead of adding RPC security flavors.

5.3.1.1. 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, the following form is RECOMMENDED:

```
nfs/hostname
```

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

NFSv4.1 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 replay protection for non-idempotent requests.

5.5. Client Identifiers and Client Owners

For each operation that obtains or depends on locking state, the specific client needs to 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 7) 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 byte-range lock, share reservation, layout state. Also, where the server supports neither the CLAIM_DELEGATE_PREV nor the CLAIM_DELEG_PREV_FH claim types, all delegation state associated with the same client is released as well. For discussion of delegation state recovery, see Section 15.2.1. For discussion of layout state recovery, see Section 17.7.1.

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 13.3) 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, allowing the server to distinguish successive incarnations (e.g., reboots) of the same client. 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, contains the client owner id. This is a variable-length string that uniquely defines the client so that subsequent instances of the same client bear the same co_ownerid with a different verifier.

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

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

* 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 implementer 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.

* 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 7.5. (Note: the precise opposite was advised in the NFSv4.0 specification [RFC3530].)

* 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:

* If applicable, the client's statically assigned network address.

* Additional information that tends to be unique, such as one or more of:

  - The client machine's serial number (for privacy reasons, it is best to perform some one-way function on the serial number).

  - A Media Access Control (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.

* 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 8), 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 needs to 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 needs to 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 needs to 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 13.4.2).

See the descriptions of EXCHANGE_ID (Section 23.35) and CREATE_SESSION (Section 23.36) for a complete specification of these operations.

5.5.1. Upgrade from NFSv4.0 to NFSv4.1

To facilitate upgrade from NFSv4.0 to NFSv4.1, a server may compare a value of data type client_owner4 in an EXCHANGE_ID with a value of data type nfs_client_id4 that was 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 that the value of data type client_owner4 be constructed the same way as the value of data type nfs_client_id4. If the latter's contents included the server's network address (per the recommendations of the NFSv4.0 specification [RFC3530]), 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.

5.5.2. Server Release of Client ID

NFSv4.1 introduces a new operation called DESTROY_CLIENTID (Section 23.50), which the client uses 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 (associated state includes unrevoked 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 that 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 7.13.1.4 for discussion on releasing inactive sessions.

5.5.3. 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 is true:

* The client ID was created without explicit state protection (i.e. SP4_NONE was used) and without client host authentication while the current EXCHANGE_ID shares those characteristics and the principals used for client Id creation and the current EXCHANGE_ID match as well.

* The client ID was created without explicit state protection (i.e. SP4_NONE was used) and with client host authentication while the current EXCHANGE_ID shares those characteristics with the EXCHANGE_ID used to create the client ID while the authenticated client hosts match as well.

* The principal that created the client ID for the client owner is the same as the principal that is sending the EXCHANGE_ID operation. Note that if the client ID was created with SP4_MACH_CRED state protection (Section 23.35), 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.

* The client ID was established with SP4_SSV protection (Section 23.35, Section 7.8.3) and the client sends the EXCHANGE_ID with the security flavor set to RPCSEC_GSS using the GSS SSV mechanism (Section 7.9).

* 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 either is updating properties of the client ID (Section 23.35) or is attempting trunking (Section 7.5), the server MUST NOT delete state.

5.6. Server Owners

The server owner is similar to a client owner (Section 5.5), 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 that each EXCHANGE_ID were sent over can be assumed to address the same server (as defined in Section 2.5). 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 Sections 7.5 and 23.35 in order to avoid acting on falsely matching server owner values.

The considerations for generating an so_major_id are similar to that for generating a co_ownerid string (see Section 5.5). 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 7.5).

5.7. Transport Layers

5.7.1. REQUIRED and RECOMMENDED Properties of Transports

NFSv4.1 works over Remote Direct Memory Access (RDMA) and non-RDMA-based transports with the following attributes:

* The transport supports reliable delivery of data, which NFSv4.1 requires. However the possibility of connections breaking is addressed in NFSv4.1 by a session-based replay cache to prevent the spurious re-execution of non-idempotent requests or modifying idempotent requests.

* 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 [RFC5531].

Because efficient handling is required when sending large amounts of data, congestion control facilities are a significant concern.

* When NFSv4.1 is used over an IP-based network protocol, it is REQUIRED that the transport provide congestion control.

* When NFSv4.1 is used over a non-IP network protocol, it is RECOMMENDED that the transport provide congestion control.

To enhance the possibilities for interoperability, it is strongly recommended that NFSv4.1 client and server implementations support operation over the TCP transport protocol.

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. As a consequence, UDP by itself MUST NOT be used as an NFSv4.1 transport, although transports to be used for NFSv4.1 may be layered on UDP. 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.

5.7.2. Client and Server Transport Behavior

[Author Aside]: Section substantially revised to address unjustified use of RFC2119-defined keywords regarding retries and replace that with appropriate implementation advice.

When a connection-oriented transport (e.g., TCP) is used, the client and server are normally expected to maintain the use of connections already established for a considerable length of time. This is for a number reasons:

* This will prevent the weakening of the transport's congestion control mechanisms by the confusion resulting from dispersing a single burst of network traffic into multiple connections.

* This will improve performance for the WAN environment by eliminating the need for connection setup handshakes. This is particularly so given that each new connection requires the re-establishment of a connection id, and setting up new connections.

* The NFSv4.1 callback model requires the client and server to maintain a client-created backchannel (see Section 7.3.1) for the server to use so that any dropping of the connection will interfere with the use of the backchannel.

Although it is not fatal for a requester to retry without a disconnect between the request and retry, there are good reasons to avoid this practice. 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.

There is no situation in which a replier is allowed to silently drop a request, whether the request is a retry or not. (The silent drop behavior of RPCSEC_GSS [RFC2203] is not relevant here since this behavior happens at the RPCSEC_GSS layer, which is at a lower layer in the request processing.) While the replier MAY disconnect the connection, if it does not do so, it is obligated to execute the request or return an appropriate error based on the contents of the reply cache (see Section 7.6.1).

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) from which the requester sent the request. If using a connection-oriented transport, replies MUST be sent on the connection from which the request was received.

If a connection is dropped after the replier receives the request but before the replier sends the reply, the replier might have a 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 requester retries the request. The reason for this prohibition is that the requester MAY retry a request over a different connection (provided that connection is associated with the original request's session).

When using RDMA transports, there are other reasons for avoiding retries over the same connection:

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

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

In addition, as described in Section 7.6.2, while a session is active, an NFSv4.1 requester that ceases to wait for an outstanding reply MUST take appropriate care to avoid that situation vitiating guarantees needed to maintain the exactly-once semantics needed for the successful operation of the session-based reply cache.

5.7.3. Ports

Historically, NFSv3 servers have listened over TCP port 2049. The registered port 2049 [RFC3232] for the NFS protocol should be the default configuration for NFSv4.1, although the port 20049 is used for NFSv4.1 layered on RPC-over-RDMA.

The use of a reserved port has been common for NFS implementations and it is expected that this will apply to NFSv4.1 as well. While the use of RPC binding protocols as described in [RFC1833] is a possibility, there is no requirement that servers provide support for such use.

In light of this, a client should avoid this sort of use unless it has good reason to expect such support to be present on the server, while accessing NFS services at the appropriate well-known port depending on the transport to be used.

6. Security-related Infrastructure

NFSv4.1 relies on the Infrastructure described by the NFSv4-wide security-related documents, currently [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls]. This infrastructure includes:

* The RPC-based facilities to provide authentication, privacy and integrity, including facilities provided by the various authentication flavors and those provided at the transport layer.

* The security negotiation facilities described in section 16 of the security document, as they have been enhanced to support selection of transport-layer facilities, as well as authentication flavors and associated services.

* The authorization facilities described in Sections 5.3, 5.4, and 7 of the security document.

* The audit and alarm facilities described in Section 13 of the security document.

There are, however, a number of places where the NFSv4-wide treatment needs to be supplemented to deal with NFSv4.1-specific features, requirements, and recommendations as discussed below:

* The parallel NFS feature is described in Section 17, with security for it dealt with in Section 17.9. The case of the file layout type is described in Section 17 with security for it dealt with in Section 18.13.

  The security for parallel NFS is dealt with in this specification even though it relies on the security infrastructure describe in the NFSv4-wide security document. As a result, it will be discussed in a restructured Section 26.

* The handling of SECINFO and SECINFO_NO_NAME is complicated because both share the extensions to the negotiation process described in Section 12 of the NFSv4-wide security document, currently [I-D.dnoveck-nfsv4-security], while the former operation in present in all minor versions while the latter is specific to NFSv4.1.

* The requirements and recommendations regarding associated security services are discussed in Section 6.1. The discussion had been modified to include the possibility that encryption at the RPC transport layer might obviate the need for these services, although the existing requirements and recommendations still stand.

  [Author Aside]: Further work in this area is likely and there should be working group discussion of possible changes. Of particular concern is the use of "SHOULD" in connection with support for privacy, as it is not clear what might be valid reasons not to support this. The provision of confidentiality using transport-based encryption further complicates the matter, although it needs to be clear that the need that confidentiality be available in some form is strongly recommended.

6.1. NFSv4.1-specific Recommendations and Requirements Regarding Security Services

[Author Aside]: Significant revisions have been made to address the hole created by the fact that the discussion of client support of data privacy uses the word "SHOULD".

Via the GSS-API, RPCSEC_GSS can be used to identify and authenticate users on clients to servers, and servers to users. Authentication of the client itself is not provided but can be provided by RPC independently of the use of RPCSEC_GSS.

GSS-API can also perform integrity checking on the entire RPC message, including the RPC header, and on the arguments or results. Finally, privacy/confidentiality, usually via encryption, is a service available with RPCSEC_GSS. Privacy is provided for 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 [RFC2203] 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. Given that it is has never been made clear, as required by the definition of "SHOULD in [RFC2119], it has to be assumed that this statement, appearing in previous specifications has been treated as providing permission for clients not to support RPCSEC_GSS privacy. In light of this situation, it needs to be understood that, with regard to the use of "SHOULD" above, valid reasons to bypass this recommendation are limited to the reliance of implementors on those previous specifications and the difficulty of changing them now.

The following consequences need to be kept in mind by those not providing such support:

* Any data accessed on connections for which rpc-tls support is not provided will be available, in the clear, to those with the ability to monitor network traffic on a network segment used to effect the access.

* The existence of clients without privacy support would make it difficult or impossible to enforce privacy constraints that would otherwise be straightforward. The effect is to undercut the process of security negotiation, which is the only possible way to provide confidentiality when rpc-tls encryption is not in effect.

The reader is directed to Section 18.3.1 of [I-D.dnoveck-nfsv4-security] for a more complete discussion of security issues regarding data in flight.

6.2. NFSv4.1-specific Details of Security Negotiation

Unlike NFSv4.0, which only has the SECINFO operation, NFSv4.1 has the SECINFO_NO_NAME operation as well. As a result, many of the details of performing security negotiation will different from those in other minor versions and need to be discussed in this document, in the sections below.

6.2.1. 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.

6.2.1.1. 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 or not the put filehandle operation returns NFS4ERR_WRONGSEC, 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 6.2.1 apply.

6.2.1.2. Two or More Put Filehandle Operations

For a series of N put filehandle operations, the server MUST NOT return NFS4ERR_WRONGSEC to the first N-1 put filehandle operations. The Nth put filehandle operation is handled as if it is the first in a subseries of operations. For example, if the server received a COMPOUND request with this series of operations -- PUTFH, PUTROOTFH, LOOKUP -- then the PUTFH operation is ignored for NFS4ERR_WRONGSEC purposes, and the PUTROOTFH, LOOKUP subseries is processed as according to Section 6.2.1.3.

6.2.1.3. 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 its parent directory.

(b) sec_policy_child ^ sec_policy_parent != {} (^ for intersection, {} for the empty set). This means that the set of 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 security 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.

6.2.1.4. 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 6.2.1.3, 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.

6.2.1.5. 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 either the 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 the 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.

6.2.1.6. Put Filehandle Operation + Nothing

The NFSv4.1 server MUST NOT return NFS4ERR_WRONGSEC.

6.2.1.7. 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 of 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).

6.2.1.8. Operations after SECINFO and SECINFO_NO_NAME

Suppose a client sends a COMPOUND procedure containing the series SEQUENCE, PUTFH, SECINFO_NO_NAME, READ, and suppose the security tuple used does not match that required for the target file. By rule (see Section 6.2.1.5), neither PUTFH nor SECINFO_NO_NAME can return NFS4ERR_WRONGSEC. By rule (see Section 6.2.1.7), 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.

6.2.2. LINK and RENAME

The LINK and RENAME operations use both the current and saved filehandles. If the security policy of the saved filehandle rejects the security flavor used in the COMPOUND request's credentials, the server MAY return NFS4ERR_WRONGSEC from LINK or RENAME. When the server does so, if there is no intersection between the security policies of saved and current filehandles, this implies that it will be impossible for the 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 or SECINFO, 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 from LINK or RENAME in that situation. Instead the server MUST do one of two things:

* The server can return NFS4ERR_XDEV.

* The server can allow the security policy of the current filehandle to override that of the saved filehandle, and so return NFS4_OK.

7. Session

NFSv4.1 clients and servers MUST support and MUST use the session feature as described in this section.

7.1. Motivation and Overview

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

* Lack of support for Exactly Once Semantics (EOS). This includes lack of support for EOS through server failure and recovery.

* Limited callback support, including no support for sending callbacks through firewalls, and races between replies to normal requests and callbacks.

* Limited trunking over multiple network paths.

* Requiring machine credentials for fully secure operation.

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

* EOS is enabled by a reply cache with a bounded size, making it feasible to keep the cache in persistent storage and enable EOS through server failure and recovery. One reason than previous revisions of NFS did not support EOS was because some EOS approaches often limited parallelism. As will be explained in Section 7.6, NFSv4.1 supports EOS without unduly limiting parallelism.

* The NFSv4.1 client (defined in Section 2.5) creates transport connections and provides them to the server to use for sending callback requests, thus solving the firewall issue (Section 23.34). Races between responses from client requests and callbacks caused by the requests are detected via the session’s sequencing properties that are a consequence of EOS (Section 7.6.3).

* The NFSv4.1 client can associate an arbitrary number of connections with the session, and thus provide trunking (Section 7.5).

* The NFSv4.1 client and server produce 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 7.8.3).

* 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 7.8.2).

A session is a dynamically created, long-lived server object created by a client and 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 or not the connection exists. 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.

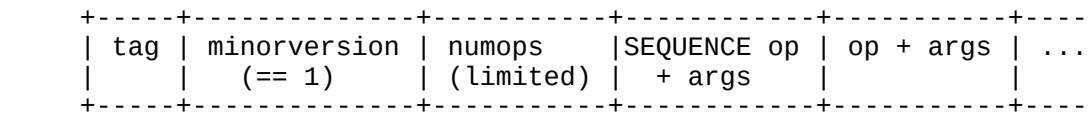
7.2. NFSv4 Integration

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 Open Network Computing (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 23.37).

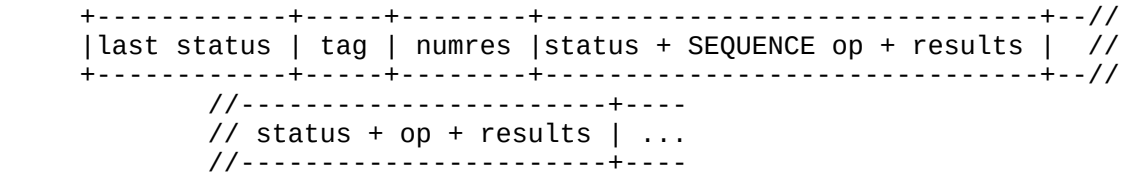
7.2.1. SEQUENCE and CB_SEQUENCE

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 7.6). Session-enabled NFSv4.1 COMPOUND

requests thus have the form:



and the replies have the form:



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 7.6.3).

7.2.2. Client ID and Session Association

Each client ID (Section 5.5) 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 (which can consist of share reservations, locks, delegations, and layouts (Section 3)) 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 that of the request 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. Sections 7.8.1 and 7.8.2 discuss the security considerations around callbacks.

7.3. Channels

A channel is not a connection. A channel represents a single direction in which ONC RPC requests are sent as part of a session..

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 is used for callback requests from server to client, and carries CB_COMPOUND requests and responses. Whether or not there is a backchannel is decided 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 7.6.1). Note that even the backchannel requires a reply cache (or, at least, a slot table in order to detect retries) because some callback operations are non-idempotent.

7.3.1. 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 23.36) and the BIND_CONN_TO_SESSION (Section 23.34) 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 23.35) 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 TCP and RDMA connections can be associated with the fore channel. In the event an RDMA and non-RDMA connection are associated with the same channel, it is desirable for the maximum number slots to be at least one more than the total number of RDMA credits (Section 7.6.1). 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.

7.4. 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 has arranged to use distinct values of opaque identifiers so that the two servers never assign the same value to two distinct objects. Thus, the identifiers generated by two servers within that set can be assumed compatible so that, in certain important cases, identifiers generated by one server in that set may be presented to another server of the 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 inadvertently accept a value generated by another server. 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 following identifiers:

* Filehandle values. A filehandle value accepted by two servers of the same server scope denotes the same object. A WRITE operation sent to one server is reflected immediately in a READ sent to the other.

* Server owner values. When the server scope values are the same, server owner value may be validly compared. In cases where the server scope values are different, server owner values are treated as different even if they contain identical strings of bytes.

The coordination 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:

* When server owner values received in response to EXCHANGE_ID operations sent to multiple network addresses are compared for the purpose of determining the validity of various forms of trunking, as described in Section 16.5.2.

* 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 13.4.2.1.

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.

* If both EXCHANGE_ID requests were sent with RPCSEC_GSS ([RFC2203], [RFC5403], [RFC7861]) 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 and server_owner major_id also share the same principal name. In some cases, this simplifies the client's task of validating server scope.

* The client may accept the appearance of the second server in the 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).

7.5. 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.

In the context of a single server network address, it can be assumed that all connections are accessing the same server, and NFSv4.1 servers MUST support both forms of trunking. When multiple connections use a set of network addresses to access the same server, the server MUST support both forms of trunking. 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 5.6) and server scope (Section 7.4), 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.

Distinguishing when the client is allowed to use session and client ID trunking requires understanding how the results of the EXCHANGE_ID (Section 23.35) operation identify a server. Suppose a client sends EXCHANGE_IDs 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 Section 7.5.1 below), 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, and 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 of 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 decline 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 and the client to be able to associate the same locking state with multiple sessions.

It is always possible that, as a result of various sorts of reconfiguration events, eir_server_scope and eir_server_owner values may be different on subsequent EXCHANGE_ID requests made to the same network address.

In most cases, such reconfiguration events will be disruptive and indicate that an IP address formerly connected to one server is now connected to an entirely different one.

Some guidelines on client handling of such situations follow:

* When eir_server_scope changes, the client has no assurance that any IDs that it obtained previously (e.g., filehandles) can be validly used on the new server, and, even if the new server accepts them, there is no assurance that this is not due to accident. Thus, it is best to treat all such state as lost or stale, although a client may assume that the probability of inadvertent acceptance is low and treat this situation as within the next case.

* When eir_server_scope remains the same and eir_server_owner.so_major_id changes, the client can use the filehandles it has, consider its locking state lost, and attempt to reclaim or otherwise re-obtain its locks. It might find that its filehandle is now stale. However, if NFS4ERR_STALE is not returned, it can proceed to reclaim or otherwise re-obtain its open locking state.

* When eir_server_scope and eir_server_owner.so_major_id remain the same, the client has to use the now-current values of eir_server_owner.so_minor_id in deciding on appropriate forms of trunking. This may result in connections being dropped or new sessions being created.

7.5.1. Verifying Claims of Matching Server Identity

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

* For session trunking, clients 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, unless client host authentication is in effect, specify that BIND_CONN_TO_SESSION is

to be verified according to the SP4_SSV or SP4_MACH_CRED (Section 23.35) state protection options. For SP4_SSV, reliable verification depends on a shared secret (the SSV) that is established via the SET_SSV (see Section 23.47) operation.

When a new connection is associated with the session (via the BIND_CONN_TO_SESSION operation, see Section 23.34), 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 (see Section 7.9).

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.

* 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 sending each EXCHANGE_ID operation. 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 that 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 sending a CREATE_SESSION operation 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 cooperating for the purposes of specifying server scope and client ID trunking.

7.6. Exactly Once Semantics

[Author Aside]: This section, including some subsections, has been substantially modified from the corresponding section appearing in previous specifications [RFC5661] [RFC8881] and earlier drafts of this document. Change has been driven primarily by the incorrect use of RFC2119-defined keywords, most importantly in the case in which RPC requests need to be aborted, leading to some related changes to clarify the appropriate level of checking for the possibility of false retry. As part of this revised description, it is explained that, given the possibility of requests being aborted, the term "Exactly-once semantics" describes an aspiration and that what is really provided would better be called "at-most-once semantics. Also, the description of retry has been revised to properly use RFC2119 keywords. For more detailed information regarding changes which have been made, see Appendix C.2.1.

Via the session, NFSv4.1 offers what is termed "exactly once semantics" (EOS) for requests sent over a channel. EOS is supported on both the fore channel and backchannel.

Although this term is well-established and will not be changed, it should be noted that what is actually provided is at-most-once semantics to accommodate the possibility that the client will need to abort RPC requests, remaining unsure about whether the requested actions have been performed one time or not at all.

Each COMPOUND or CB_COMPOUND request that is sent with a leading SEQUENCE or CB_SEQUENCE operation needs to be executed by the receiver at most once. This requirement holds regardless of whether the request is sent with reply caching specified (see Section 7.6.1.3). The requirement also holds in the case in which NFSv4.1 is a pNFS data access protocol and the requester is sending the request over a session created between a pNFS data client and pNFS data server. To help understand the need for this requirement, we divide the requests sent to be executed into three categories:

* Non-idempotent requests.

* Idempotent modifying requests.

* Idempotent non-modifying requests.

An example of a non-idempotent request is RENAME. Obviously, 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 non-idempotent 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, which could result from executing the same write request multiple times including some executions that occur after the completion of the first is noted by the requester.

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 byte-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 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 or not they are idempotent or modifying.

Note that fully complete EOS is not possible unless the server persists the reply cache in stable storage, and 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 8 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 relative to previous versions of NFS because the earlier duplicate request/reply caches were based on

the ONC RPC transaction identifier (XID). Section 7.6.1 explains the shortcomings of the XID as a basis for a reply cache and describes how NFSv4.1 sessions improve upon the XID.

7.6.1. 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 look up (often via a hash that includes XID and source address). NFSv4.1 requests include a non-opaque slot ID, which can be used as 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. In practice, previous versions of NFS have chosen to store a fixed number of replies in the cache, and to 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 23.36), 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 that 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 (Section 23.36). 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 mod 2^32).

The sequence ID accompanies the slot ID in each request. It is for the critical check at the replier: 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 requester to assert that it is retransmitting to implement this, because for any given request the requester cannot know whether the replier has seen it unless the replier actually replies. Of course, if the requester has seen the reply, the requester 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:

* 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.

* 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 7.6.2 for direction on how the replier deals with retries of requests that are still in progress.

* 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).

* A misordered new request, in which the sequence ID is two or more than (accounting for sequence wraparound) that previously seen in the slot. Note that because the sequence ID MUST wrap around to zero 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.

As a result, the slot ID, sequence ID, and session ID 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 needs to be formulated in NFSv4.1 requests as in any other ONC RPC request. The reasons include:

* The RPC layer retains its existing semantics and implementation.

* 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.

* 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.

* 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 raises the question: why do SEQUENCE and CB_SEQUENCE replies have a session ID, slot ID, and sequence ID? Having the session ID in the reply means that the requester does not have to use the XID to look up 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 that the requester does not have to use the XID to look up 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 (See [rpc_xid_issues]); 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:

* 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 might have active requests in flight reflecting prior values; therefore, the replier cannot immediately require the requester to comply.

* 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.

  A requester can be intransigent with respect to lowering its highest_slotid argument to a Sequence operation, i.e. the requester continues to ignore the target highest_slotid in the response to a Sequence operation, and continues to set its highest_slotid argument to be higher than the target highest_slotid. This can be considered particularly egregious behavior when the replier knows there are no outstanding requests with slot IDs higher than its target highest_slotid. When faced with such intransigence, the replier is free to take more forceful action, and 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_HIGH_SLOT, 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.

The requester can also be intransigent with respect to sending non-retry requests that have a slot ID that exceeds the replier's highest_slotid. Once the replier has forcibly lowered the enforced highest_slotid, the requester is only allowed to send retries on slots that exceed the replier's highest_slotid. If a request is received with a slot ID 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 by using the next sequence ID, the requester is indicating it has received the previous reply for the slot.

* The requester is better off using the lowest available slot when sending 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 the requester has 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.

7.6.1.1. 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 or not a replier is re-computing highest slot ID, target slot ID, and status on replies to retries, the requester cannot assume that 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 replier 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 of the reply 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 that might be out of date.

7.6.1.2. Errors from SEQUENCE and CB_SEQUENCE

Any time SEQUENCE or CB_SEQUENCE returns 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.

7.6.1.3. 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 [Chet]. Caching the reply may offer little benefit. If the reply is too large (see Section 7.6.4), it may not be cacheable anyway. Even if the reply to an idempotent request is small enough to cache, unnecessarily caching the reply slows down the server and increases RPC latency.

Whether or not the requester requests the reply to be cached has no effect on the slot processing. If the result of SEQUENCE or CB_SEQUENCE is 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:

* The replier can cache the entire original reply. Even though sa_cachethis or csa_cachethis is FALSE, the replier is always free to cache. It may choose this approach in order to simplify implementation.

* 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. If a replier receives a retried Sequence operation where the reply to the COMPOUND or CB_COMPOUND was not cached, then the replier,

  - MAY return NFS4ERR_RETRY_UNCACHED_REP in reply to a Sequence operation if the Sequence operation is not the first operation (granted, a requester that does so is in violation of the NFSv4.1 protocol).

  - MUST NOT return NFS4ERR_RETRY_UNCACHED_REP in reply to a Sequence operation if the Sequence operation is the first operation.

* If the second operation is an illegal operation, or an operation that was legal in a previous minor version of NFSv4 and MUST NOT be supported in the current minor version (e.g., SETCLIENTID), the replier MUST NOT ever return NFS4ERR_RETRY_UNCACHED_REP. Instead the replier MUST return NFS4ERR_OP_ILLEGAL or NFS4ERR_BADXDR or NFS4ERR_NOTSUPP as appropriate.

* If the second operation can result in another error status, the replier MAY return a status other than NFS4ERR_RETRY_UNCACHED_REP, provided the operation is not executed in such a way that the state of the replier is changed. Examples of such error statuses include NFS4ERR_SEQUENCE_POS,NFS4ERR_REQ_TOO_BIG, and NFS4ERR_NOTSUPP returned for an operation that is legal but not REQUIRED in the current minor version and but is not supported by the replier or its file system.

The discussion above assumes that the retried request matches the original one. Section 7.6.1.3.1 discusses what the replier might do, and MUST do when it is aware that original and retried requests do not match. Since the replier might only cache a small amount of the information that would be required to determine whether this is a case of a false retry, the replier may send to the client any of the following responses:

* The cached reply to the original request. This done if users of the original request and retry match, and there is no evidence that there is in fact a mismatch between the original request and retry.

This can occur if the server caches the entire request and compares it to the retry but also in situations in which only a limited comparison or no comparison is possible. For details see Section 7.6.1.3.1

* A reply that consists only of the Sequence operation with the error NFS4ERR_SEQ_FALSE_RETRY.

  This is done if the users of the original request and putative retry do not match, or if there is the server has sufficient data to indicate that that supposed retry does not match the original request.

* A reply consisting of the response to Sequence with the status NFS4_OK, together with the second operation as it appeared in the retried request with an error of NFS4ERR_RETRY_UNCACHED_REP or other error as described above.

* A reply that consists of the response to Sequence with the status NFS4_OK, together with the second operation as it appeared in the original request with an error of NFS4ERR_RETRY_UNCACHED_REP or other error as described above.

7.6.1.3.1. False Retry

[Author Aside]: Section substantially revised to explain why false retries can occur, even though EOS is designed to avoid them. This is used as a basis for explaining the potential need for false retry detection while avoiding a level of checking that would be a performance issue.

The mechanisms described in Section 7.6 are designed to ensure that if a Sequence operation is sent and matches a request in the reply cache with the same slot ID and sequence ID then, it is a retry of that original request. However, it is possible, although quite unlikely, that servers will encounter requests where this is not the case, in which case the request is considered a "false retry".

* False reties can occur if the client does not implement request sequencing as described in Section 7.6.

* They can also occur as a result of situations in which large number of requests are aborted and considered complete, even though no response has been received by the requester. However, for this situation to result in a false retry there would have to be a sequence of over four billion such requests being processed using the same slot ID with that sequence followed by a long-delayed transmission of an abandoned request.

If a requester sent a Sequence operation with a slot ID and sequence ID that are in the reply cache but the replier detects that the retried request is not the same as the original request, including a retry that has different operations or different arguments in the operations from the original and a retry that uses a different principal in the RPC request's credential field that translates to a different user, then this is a false retry.

Given the low expected frequency of such false retries, the replier is not obligated to check for their existence although it is prudent to do so with requesters whose implementation of EOD is any way suspect or where the requests are transmitted over a network capable of delivering a request a very long time after it was sent. When the replier does detect a false retry, it is permitted (but not always obligated) to return NFS4ERR_SEQ_FALSE_RETRY in response to the Sequence operation when it detects a false retry.

Translations of particularly privileged user values to other users due to the lack of appropriately secure credentials, as configured on the replier, should be applied before determining whether the users are the same or different. If the replier determines the users are different between the original request and a retry, then the replier MUST return NFS4ERR_SEQ_FALSE_RETRY.

Regardless of whether such user mismatches do occur, the occurrence of false retries is an indication that the EOS logic is faulty, has not been implemented correctly, or that there is an extraordinary frequency of aborted requests. In light of this fact, there are practical limits to the information that might be saved in order to determine whether a particular request is a false retry. In the case of large requests recording the entire request might not be practical while a recording a compact form in the form of a checksum might unacceptably limit performance.

In the case of requests for which the reply is cached, comparing the operations in the cached response to those in the putative retry can serve to detect interactions with clients not properly implementing EOS or aborting requests inappropriately. In other cases, recording the operation count and the identity of the first non-SEQUENCE operation can make a simple check for false retry feasible.

If an operation of the retry is an illegal operation, or an operation that was legal in a previous minor version of NFSv4 and MUST NOT be supported in the current minor version (e.g., SETCLIENTID), the replier MAY return NFS4ERR_SEQ_FALSE_RETRY (and MUST do so if the users of the original request and retry differ). Otherwise, the replier MAY return NFS4ERR_OP_ILLEGAL or NFS4ERR_BADXDR or NFS4ERR_NOTSUPP as appropriate. Note that the handling is in

contrast for how the replier deals with retries requests with no cached reply. The difference is due to NFS4ERR_SEQ_FALSE_RETRY being a valid error for only Sequence operations, whereas NFS4ERR_RETRY_UNCACHED_REP is a valid error for all operations except illegal operations and operations that MUST NOT be supported in the current minor version of NFSv4.

7.6.2. Retry and Replay of Reply

Because NFSv4.1 is used on transports providing reliable delivery, retrying requests within an existing connection is unlikely to be helpful. Requesters will not normally 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, to deal with the possibility that the original connection is no longer functioning appropriately.

If the requester is a server wanting to re-send a callback operation over the backchannel of a 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 is forced to 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 then has no option but to associate a new connection with the session (or destroy the session).

Note that it is not, in general, fatal for a requester to retry without a disconnect between the request and retry. However, in order to prevent false retries (see Section 7.6.1.3.1), the requester MUST NOT retry a request once the slot used to send that request has been used to send a new request.

Nevertheless, 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 will normally wait for a reply to a request before using the slot for another request and MUST do so unless events such as termination of the issuing process makes it impossible to do so. If no such situation were to arise, then the protocol design would ensure no false retry situation could occur (see Section 7.6.1.3.1 for details. When it does not wait for a reply, the requester cannot

be sure that using the next sequence ID for the slot chosen, as it normally does, will always be accepted. 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).

In light of the above, clients that do not wait for a reply before reusing the slot need to be aware of the possibility of receiving NFS4ERRR_SEQ_MISORDERED as a result and infer the probable existence of a request not received by the server. The client will then adjust the current sequence id sent, using successful execution as an indication that seqids on that slot are again correctly aligned.

RDMA fabrics do not guarantee that the memory handles (Steering Tags) within each RPC/RDMA "chunk" [RFC8166] 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 that 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. In this case, 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 that the requester specified the reply to be cached).

7.6.3. 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 the 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 that might result in some sort of server callback, the server SHOULD keep track of the { session ID, slot ID, sequence ID } triple of the client request until the slot ID 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 that 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 to 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 17.5.5.2.

7.6.4. 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 23.36), 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 the first operation (SEQUENCE or CB_SEQUENCE) in the request (which means that no operations in the request executed and that 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 that at least one operation did execute and that the state of the slot in the 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 the 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 is TRUE, then the replier MUST cache a reply except if an error is returned by the SEQUENCE or CB_SEQUENCE operation (see Section 7.6.1.2). If the reply exceeds ca_maxresponsesize_cached (and sa_cachethis or csa_cachethis is 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 an operation other than the first operation (SEQUENCE or CB_SEQUENCE), then the reply MUST be cached if sa_cachethis or csa_cachethis is 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 the 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), it does 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 23.16), retry is not always available as an option. The following guidelines for the handling of filehandle-changing operations are advised:

* 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.

* A server MAY return NFS4ERR_REP_TOO_BIG or NFS4ERR_REP_TOO_BIG_TO_CACHE (if sa_cachethis is TRUE) on a filehandle-changing operation if the reply would be too large on the next operation.

* A server SHOULD return NFS4ERR_REP_TOO_BIG or NFS4ERR_REP_TOO_BIG_TO_CACHE (if sa_cachethis is TRUE) on a filehandle-changing, non-idempotent operation if the reply would be too large on the next operation, especially if the operation is OPEN.

* A server MAY return NFS4ERR_UNSAFE_COMPOUND 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.

7.7. RDMA Considerations

A complete discussion of the operation of RPC-based protocols over RDMA transports is in [RFC8166]. A discussion of the operation of NFSv4, including NFSv4.1, over RDMA is in [RFC8267]. 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.

7.7.1. 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 that would cause a fatal error on the RDMA connection.

NFSv4.1 manages slots as resources on a per-session basis (see Section 7), 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 of slots in the session is Y, then the maximum number of outstanding requests is the lesser of X and Y.

7.7.2. 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 23.36). These limits then provide the maxima within which each connection associated with the session's channel(s) must remain. RDMA connections are managed within these limits as described in Section 3.3 of [RFC8166]; 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 the total number of RDMA credits across all connections associated with the replier's channel does exceed the channel's maximum number of outstanding requests.

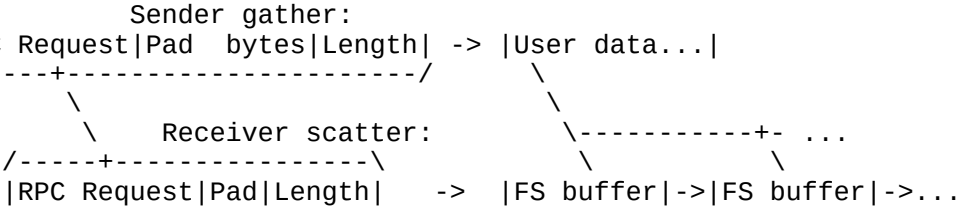
The limits may also be modified dynamically at the replier's choosing by manipulating certain parameters present in each NFSv4.1 reply. In addition, the CB_RECALL_SLOT callback operation (see Section 25.8) can be sent by a server to a client to return RDMA credits to the server, thereby lowering the maximum number of requests a client can have outstanding to the server.

7.7.3. Padding

Header padding is requested by each peer at session initiation (see the ca_headerpadsize argument to CREATE_SESSION in Section 23.36), and subsequently used by the RPC RDMA layer, as described in [RFC8166]. 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 WRITE operation is aligned. Most servers can make good use of such padding, which allows them to chain receive buffers in such a way that any data carried by client requests will be placed into appropriate buffers at the server, ready for file system processing. The receiver's RPC layer encounters no overhead from skipping over pad bytes, and the RDMA layer's high performance makes the insertion and transmission of padding on the sender a significant optimization. In this way, the need for servers to perform RDMA Read to satisfy all but the largest client writes is obviated. An added benefit is the reduction of message round trips on the network -- a potentially good trade, where latency is present.

The value to choose for padding is subject to a number of criteria. A primary source of variable-length data in the RPC header is the authentication information, the form of which is client-determined, possibly in response to server specification. The contents of COMPOUNDs, sizes of strings such as those passed to RENAME, etc. all go into the determination of a maximal NFSv4.1 request size and therefore minimal buffer size. The client must select its offered value carefully, so as to avoid overburdening the server, and vice versa. The benefit of an appropriate padding value is higher performance.



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 that 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.

7.7.4. Dual RDMA and Non-RDMA Transports

Some RDMA transports (e.g. [RFC5040]) 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 Sections 23.36 and 23.34).

7.8. Session Security

7.8.1. 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 23.34) and subject to the same firewall and routing checks as the fore channel. At the client’s option (see Section 23.35), connection association is fully authenticated before being activated (see Section 23.34). Traffic from the server over the backchannel is authenticated exactly as the client specifies (see Section 7.8.2).

7.8.2. 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 combinations not offered. This way, the client need not provide a target GSS principal for the backchannel as it did with NFSv4.0, nor does the server have to implement an RPCSEC_GSS initiator as it did with NFSv4.0 [RFC3530].

The CREATE_SESSION (Section 23.36) and BACKCHANNEL_CTL (Section 23.33) operations allow the client to specify flavor/ principal combinations.

Also note that the SP4_SSV state protection mode (see Sections 23.35 and 7.8.3) has the side benefit of providing SSV-derived RPCSEC_GSS contexts (Section 7.9).

7.8.3. 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 that 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 23.35).

The first (SP4_NONE) is to simply waive state protection, except for that provided by client host authentication.

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

* 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.

* 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.

* 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 safeguarding the per-machine credential. Assuming a proper safeguard 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. 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 needs to be allocated at least one session each, so the approach suffers from lack of economy.

The SP4_SSV protection option uses the SSV (Section 2.5), via RPCSEC_GSS and the SSV GSS mechanism (Section 7.9), to protect state from attack. The SP4_SSV protection option is intended for the situation comprised of a client that has multiple active users and a system administrator who wants to avoid the burden of installing a permanent machine credential on each client. The SSV is established and updated on the server via SET_SSV (see Section 23.47). To prevent eavesdropping, a client SHOULD send SET_SSV via RPCSEC_GSS with the privacy service or use tls encryption on the connection making the request. 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:

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

* Because the initial value of the SSV is zero, therefore known, the client that opts for 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 7.9). A client SHOULD send SET_SSV as soon as a session is created.

* A SET_SSV request 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:

* 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 the 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.

* 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 that 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 which the server has, and now the client knows the session was hijacked. The legitimate client establishes a new client ID.

* 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 or TLS 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 23.35) is in force, this will thwart unauthorized attempts to change locking state.

7.9. The Secret State Verifier (SSV) GSS Mechanism

The SSV provides the secret key for a GSS mechanism internal to NFSv4.1 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 (1.3.6.1.4.1.28882.1.1). 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 6.2).

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 [RFC2104] 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 value for that subkey of data type ssv_subkey4. If the length of the output of the HMAC algorithm exceeds the length of key of the encryption algorithm (which is also negotiated by EXCHANGE_ID), then the subkey MUST be truncated from the HMAC output, i.e., if the subkey is of N bytes long, then the first N bytes of the HMAC output MUST be used for the subkey. The specification of EXCHANGE_ID states that the length of the output of the HMAC algorithm MUST NOT be less than the length of subkey needed for the encryption algorithm (see Section 23.35).

```
/* 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_MIC_T2I 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 [RFC2743]). 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. Prepended 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 one after SET_SSV (Section 23.47) is called the first time on a client ID. Thereafter, the SSV sequence number 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 23.35, 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. Prepended 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 the 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 data 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 the 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 [RFC2743]). 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 of whether or not the caller of GSS_Wrap() requests confidentiality, 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. 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, which is done 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 has 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 "Context Creation Requests", Section 5.2.2 of [RFC2203]). However, Section 7.10 discusses special considerations for the SSV mechanism when used with RPCSEC_GSS.

7.10. Security Considerations for RPCSEC_GSS When Using the SSV Mechanism

When a client ID is created with SP4_SSV state protection (see Section 23.35), the client is permitted to associate multiple RPCSEC_GSS handles with the single SSV GSS context (see Section 7.9). Because of the way RPCSEC_GSS (both version 1 and version 2, see [RFC2203] and [RFC5403]) calculate the verifier of the reply, special care must be taken by the implementation of the NFSv4.1 client to prevent attacks by a man-in-the-middle. The verifier of an RPCSEC_GSS reply is the output of GSS_GetMIC() applied to the input value of the seq_num field of the RPCSEC_GSS credential (data type rpc_gss_cred_ver_1_t) (see Section 5.3.3.2 of [RFC2203]). If multiple RPCSEC_GSS handles share the same GSS context, then if one handle is used to send a request with the same seq_num value as another handle, an attacker could block the reply, and replace it with the verifier used for the other handle.

There are multiple ways to prevent the attack on the SSV RPCSEC_GSS verifier in the reply. The simplest is believed to be as follows.

* Each time one or more new SSV RPCSEC_GSS handles are created via EXCHANGE_ID, the client SHOULD send a SET_SSV operation to modify the SSV. By changing the SSV, the new handles will not result in the re-use of an SSV RPCSEC_GSS verifier in a reply.

* When a requester decides to use N SSV RPCSEC_GSS handles, it SHOULD assign a unique and non-overlapping range of seq_nums to each SSV RPCSEC_GSS handle. The size of each range SHOULD be equal to MAXSEQ / N (see Section 5 of [RFC2203] for the definition of MAXSEQ). When an SSV RPCSEC_GSS handle reaches its maximum, it SHOULD force the replier to destroy the handle by sending a NULL RPC request with seq_num set to MAXSEQ + 1 (see Section 5.3.3.3 of [RFC2203]).

* When the requester wants to increase or decrease N, it SHOULD force the replier to destroy all N handles by sending a NULL RPC request on each handle with seq_num set to MAXSEQ + 1. If the requester is the client, it SHOULD send a SET_SSV operation before using new handles. If the requester is the server, then the client SHOULD send a SET_SSV operation when it detects that the server has forced it to destroy a backchannel's SSV RPCSEC_GSS handle. By sending a SET_SSV operation, the SSV will change, and so the attacker will be unavailable to successfully replay a previous verifier in a reply to the requester.

Note that if the replier carefully creates the SSV RPCSEC_GSS handles, the related risk of a man-in-the-middle splicing a forged SSV RPCSEC_GSS credential with a verifier for another handle does not exist. This is because the verifier in an RPCSEC_GSS request is computed from input that includes both the RPCSEC_GSS handle and seq_num (see Section 5.3.1 of [RFC2203]). Provided the replier takes care to avoid re-using the value of an RPCSEC_GSS handle that it creates, such as by including a generation number in the handle, the man-in-the-middle will not be able to successfully replay a previous verifier in the request to a replier.

7.11. Session Mechanics - Steady State

7.11.1. 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 7.13.2.

7.11.2. Obligations of the Client

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

* Keep a necessary session from going idle on the server. A client that requires a session but nonetheless is not sending operations risks having the session be destroyed by the server. This is because sessions consume resources, and resource limitations may force the server to cull 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 7.12) has expired.

* 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.

* Maintain GSS contexts and RPCSEC_GSS handles 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 RPCSEC_GSS handles and/or their GSS contexts that are handed to the server via BACKCHANNEL_CTL or CREATE_SESSION are unexpired.

* 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 to 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.

7.11.3. 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 it 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 or not the server undertakes to persist the session reply cache in which a server restarts, 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 needs to 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 needs to 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 needs to 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.

7.12. 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 11.12.1.11). As with lease renewal (Section 13.3), 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.

7.13. Session Mechanics - Recovery

7.13.1. Events Requiring Client Action

The following events require client action to recover.

7.13.1.1. RPCSEC_GSS Context Loss by Callback Path

If all RPCSEC_GSS handles granted by the client to the server for callback use have expired, the client MUST establish a new handle via BACKCHANNEL_CTL. The sr_status_flags field of the SEQUENCE results indicates when callback handles are nearly expired, or fully expired (see Section 23.46.3).

7.13.1.2. Connection Loss

If the client loses the last connection of the session and wants to retain the session, then it needs to 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 of connection loss, then if the client wants to continue to use the session, it MUST retry the request, as described in Section 7.6.2. 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 prevent revocation of recallable state, the client needs to 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.

7.13.1.3. 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 needs to use BACKCHANNEL_CTL to assign new contexts.

7.13.1.4. Loss of Session

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

* Replier failure and restart.

* A catastrophe that causes the reply cache to be corrupted or lost on the media on which it was stored. This applies even if the replier indicated in the CREATE_SESSION results that it would persist the cache.

* The server purges the session of a client that has been inactive for a very extended period of time.

* 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 that 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 often leads 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.

Although loss of session is often associated with loss of the associated clientid and corresponding locking state, this is not always the case. A session can be lost without loss of the corresponding clientid-based locking state in the event of clientid trunking, or when locking state is stored persistently but the reply cache is not. See Section 8 for details.

In the event of server restart, in the absence of clientid trunking, the following situations can arise: can arise:

* If neither the reply cache nor locking state is being stored persistently both the session and clientid are lost and new ones need to be established to continue operation.

* If the reply cache is persistent, it is possible that existing locking state is available so the existing session id and clientid can be tried going forward to determine if operation can be continued with existing locking state or a new clientid needs to be established and locks reclaimed.

* If the reply cache is not persistent, and the locking state is available in persistent storage the session is lost and a new session can be created for the existing clientid.

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 that 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 that 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_id value 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 for which 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 that results in the same network address 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 either that 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 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. In this example, one of the following occur: the NFSv4 server returns NFS4ERR_MINOR_VERS_MISMATCH, the NFS server returns a PROG_MISMATCH error, the RPC listener on 2049 returns PROG_UNVAIL, or attempts to reconnect 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 needs to 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 byte-range 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 byte-range 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, byte-range lock, open, delegation, and layout state. See Section 13.4.2. A session can survive a server restart, but lock recovery may still be needed.

It is possible that 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.

7.13.2. Events Requiring Server Action

The following events require server action to recover.

7.13.2.1. Client Crash and Restart

As described in Section 23.35, a restarted client sends EXCHANGE_ID in such a way that it causes the server to delete any sessions it had.

7.13.2.2. 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.

7.13.2.3. Extended Network Partition

To the server, the extended network partition may be no different from a client crash with no restart (see Section 7.13.2.2). 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.

7.13.2.4. Backchannel Connection Loss

If there were callback requests outstanding at the time of a connection loss, then the server MUST retry the requests, as described in Section 7.6.2. 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 re-established. There are two situations, each of which uses different status flags: no connectivity for the session's backchannel and no connectivity for any session backchannel of the client. See Section 23.46 for a description of the appropriate flags in sr_status_flags.

7.13.2.5. GSS Context Loss

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

7.14. 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 18.1 for pNFS sessions considerations.

8. Persistence

[Author Aside]: This is a new top-level section which is based on the Persistence section previously within the discussion of Exactly-once Semantics. Essentially, it deletes the feature described in [RFC8881] which could never be implemented in that form and addresses the need with a new feature having the same goals.

While file data and metadata are typically stored persistently and are not affected by server restart, with the exception of certain optimizations for writing data, there are two sorts of data not normally stored persistently, that often are affected by server restart. Since [RFC8881] did not address either of these in a way that could be implemented, the entire area has been respecified for reasons discussed in Section 8.1.

For each of these type of data, the protocol provides an OPTIONAL feature whereby the server can provide persistent storage to eliminate functional problems when the data is lost or to simplify the process of reconstructing the data based on the client's knowledge.

* Reply caches may be stored persistently, as described in Section 8.2, allowing the same at-most-once semantics (often called "EOS") provided by the session-based reply cache to be maintained across server restarts.

  As discussed below, the server may provide a persistent reply cache allowing EOS across server restarts or fully persistent sessions that allow the use of existing sessions to be continued across server restart.

* Per-client locking state may be stored persistently, as described in Section 8.3, allowing clients to continue after server restart without the delay caused by interposing a grace period during which all new lock requests are to be rejected.

  If per-client locking state is not stored persistently, a grace period is provided to allow clients time to reclaim their locks. When this period is needed, requests to obtain new locks (e.g. when opening a file) are delayed until all clients have had a chance to reclaim their locks.

Although the incremental cost of supporting lock persistence is generally low enough that servers providing persistent sessions would provide persistent locking state as well, these two features are independent and the client cannot always assume lock persistence is available when an associated session is persistent and successfully recovered. For a discussion of how the client would be able to determine what state has been stored persistently and continue operation without unnecessary disruption, see Section 8.4

8.1. Need for Feature Respecification

The original material has been modified substantially and extended in order address the three items listed below. As a result, the focus of the section has shifted to include all elements relevant to persistence across server failure, rather than dealing only with reply cache issues.

* Eliminate elements of the description that made the feature essentially unimplementable. These include overbroad requirements for atomicity and the assumption that all requests needed to be continued across server restart.

* Appropriately discuss lock persistence and its relation to reply cache persistence and session persistence.

* Provide new material describing the process by which the client finds out about the presence of persistence- related features in the event of server restart.

8.2. Persistence of Reply Cache

Since the reply cache is bounded, it is possible for the reply cache to be maintained in persistent storage so that it can be made available across server restarts. When the server undertakes to provide this support when the session is created (see Section 23.36 for details), it is uncertain whether what will provided is either:

* Persistence of the reply cache only.

* Persistent of the session including its membership within the clientid of which it is a part.

The replier needs to persist the following information if it agreed to provide persistence for the session (when the session was created;

* The session ID.

* The slot table

  This need to include the sequence ID and cached reply for each slot.

* Information about the connection(s) used by the server with is sufficient to determine whether a client attempting to connect after a server Restart.

This sort of information can be used to provide either of the two distinct sorts of session-based persistence. The server provides no specific commitment to provide either of these, although, as described in Section 8.4, the client will be able to determine which form, if any, has actually been provided, and respond appropriately

In describing persistence-related semantics it will be helpful to define the following two terms:

* An operation is said "reply-caching relevant" if it is either non-idempotent, modifying, or is the final operation (including the case of request termination because of an error) of a request that is specifically requested to be cached (i.e., has a SEQUENCE operation with sa_cachethis set to true).

* A request is said "reply-caching relevant" if it contains one or more operations which are non-idempotent or modifying or it is specifically requested to be cached (i.e., has a SEQUENCE operation with sa_cachethis set to true).

Whichever form of session-based persistence is provided by the server, any requests the client retries after the server restarts will return the results that are cached in the reply cache, However, these two forms differ with regard to the handling of new requests and the possible use of clientid-based persistence facilities:

* If only reply cache persistence is provided, any new requests will fail with NFS4ERR_DEADSESSION being returned as the result of the initial SEQUENCE operation.

  Because there is no need to use the sequence id to order future request the server does not need to update persistent storage, if two successive requests using the same slot are both not reply-caching relevant, although it does if one or both of the request is reply-cache relevant.

* If session persistence is provided, the existing session can be used after connection re-establishment to support the execution of new requests so that the client will be able to continue just as it would have if no session restart had occurred.

A persistent reply cache places certain demands on the server. Although it is not it is not necessary to execute successive operations within a COMPOUND atomically, the transfer of the results of a set of operations and their installation in the persistent cache must be immediate following the execution of any reply-cache relevant operation so that it is impossible for operations to be executed or have other visible effects while not appearing in persistent reply cache.

If a client were to retry a sequence of operations that was issued to the server, the only acceptable outcomes are:

* an indication that the request is still being processed.

* a cached reply reflecting the completion of the request,

* a cached reply reflecting the interruption of the request due to server failure.

* an indication that the 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).

The possibility exists of situations in which 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. If the server allows COMPOUND procedures to be continued after server failure, it creates significantly greater challenges for the execution of such requests and the atomic placement of results in the reply cache.

When a server providing a persistent reply cache does not continue a COMPOUND procedure that was interrupted by a server failure, the error NFS4ERR_DEADSESSION is returned on the last operation which was executed.

8.3. Persistence of Locking State

Servers may make locking state available across a server restart in a number of ways including the following:

* Data related to the existence of locks and their corresponding characteristics can be stored in persistent RAM and then used after restart if the address of that storage can be reliably obtained after restart.

* The storage of locking-related state can be integrated with the file system by treating locking state in the same fashion used for other metadata.

* Locking state information may be periodically logged to block-based low-latency persistent storage with logging of individual updates.

Although the details will vary with the means of providing persistence that is adopted, it is important that locking state made available across the server restart be consistent with locking state reflected in the results of requests made by clients.

The simplest part of this is to ensure that all locking state changes are effectively made available persistently before returning to the requester. In addition, when lock state additions or deletions are reflected in the processing of other operations, the state changes must be available persistently before allowing or denying some operation done by another client. For example, when opens denying write prevent file removal, granting such opens or doing corresponding closes need to be reflected persistently before denying or allowing corresponding file removal. Similar consideration apply to doing IO when mandatory byte-range locks are supported

The following items need to be kept in mind:

* There is no commitment by the server to provide this persistence and it may be dropped if for a particular client if unusual situations make it advisable.

  This decision is made separately for each client so that it is possible there will be server restarts where some, but not all, clients have persistent locking state available.

* While the fact that a reclaim on a reclaimable lock is part of the locking state which is to be persistent, the client’s state of awareness of that need not be.

  There is thus no need for the reclaiming client to inform the server that it has completed specific individual reclaims after receiving the response.

8.4. Client Handling of Server Failure When Persistence Can be Used

When server failure occurs, the connection to the client will be disconnected and the client can then find out, as described below, whether server failure has occurred and what steps are necessary to continue use of the client with minimal disruption to those using the client.

This process includes the potential use of a persistent reply cache, as described in Section 8.5. The same process is followed depending on whether the server provided only a persistent reply cache or full session persistence.

If the server did not promise any session persistence, the client instead immediately does an EXCHANGE_ID followed by a CREATE_SESSION. On the other hand, if there was a possible use of a persistent reply cache, the use of EXCHANGE_ID/CREATE_SESSION is conditional and only happens if a new request has been completed with the error NFS4ERR_STALECLIENTID.

In either case, the next step depends on whether the clientid is the same as the one before the disconnection. If it is, then recovery is complete and new requests can be issued. This could happen if there were no server restart but also could if a combination of session-based and clientid-based persistence allowed the server failure to be dealt with essentially transparently.

In the case in which the clientid is different, the client need to reclaim its locks, as described in Section 8.6.

Even in the case in which lock persistence is available for a client, it is still possible that attempts to obtain new locks will fail with NFS4ERR_GRACE if other clients do not have their locks made available persistently.

8.5. Client Use of Session-based Persistence

After the connection to the server is re-established, the server will try to re-establish the connection, as the connection breakage occurred at a lower layer, without server restart. Although it is theoretically possible for an intermediary to hide such a disconnection, it would cause problems if it were to do so and the client had no knowledge of the server failure The discussion here assumes that no such disconnection-hiding implementation is in effect

After re-establishing the connection to the server, the client would initially attempt to continue use of the session, since it has no knowledge of whether the disconnection was the result of a server

restart. If persistence not was requested when creating the session or the server indicated it was not present, then the client can legitimately conclude that EOS semantics was not available across server restart and needs to operate in that environment.

The continued use of the existing session could include both retries of requests issued before the disconnection and issuing new requests. As a result, the discussion below will deal with both type of requests. Given that context, one needs to note the following:

* Whether a given request is a retry or a new one may be judged differently by the client and the server.

  While it is virtually certain that a new request issued by the client will be perceived as such by the server, the reverse is not the case. Retries issued by the client might be perceived as new requests, if the original requests was lost before it was executed or its existence was noted in persistent storage.

* Although it might be desirable for a client to obtain information about existing requests before issuing new one, the discussion will not assume that clients take steps to prevent new requests from being issued.

  Since retries, as perceived by the client, may be considered as new requests by the server, the prevention of new requests by the client does not ensure that the server will not see and respond to such requests.

After re-establishing the connection, the client will be able to issue requests, including retries of requests already issued before the disconnection occurred. These retries need to be issued since there is no way the results of these requests could be communicated back to the client in the absence of a retry since the connection on which it was received no longer exists.

When responses to these requests are received, what is to be done depends primarily about the error, if any, associated with the response:

* In all the cases except the two special error codes noted in the bulleted items below, including receiving no error, the client can conclude that the request was executed to completion as reflected in the response. By design, the client is not aware of whether the execution occurred before or after the serve restart, or whether a server restart, in fact, occurred. However, if persistence was requested when the session was created and the server indicated it was present, the client can assume that the request was executed exactly once with the result reflected in the response.

  When this is the result that is returned for new requests, it can be because the server has provided full session persistence or because no server restart has occurred. In the former case, it must be true that the server has provided persistent storge of locking state for the d associated clientid since, if it had not, the error NFS4ERR_STALECLIENTID would have been returned.

* In the case that NFS4ERR_DEADSESSION is returned on the SEQUENCE operation, the most likely cause is that the request was, from the server's point of view, a new request and that session persistence was not provided by the server. In this case, the current request should be deferred until the results of all retried requests known to the client have been resolved. Others that are considered new by the server also need to be deferred until are reply cache information is obtained.

  In the case that NFS4ERR_DEADSESSION is returned on another operation, the request is one that was discontinued as a result of server restart. It is most likely that the request was one that contained more than one non-idempotent or modifying operations, with the server failing after one had been completed but before later operations were started. In this case the client has been informed of a partially complete request and needs to issue a new request to include the operations that were not performed as part of the initial request.

* In the case that the error NFS4ERR_STALECLIENTID is returned, the server has recognized a new request but was unable to continue its execution because the locking information it would use has been destroyed as part of the server restart. This can occur if no persistence was provided for the session, if the persistence was limited to the reply cache or if there was session persistence and client locking state was not maintained persistently.

In this case lock recovery will be required but it will need to be delayed until all requests that were issued before the disconnection have been marked completed using the persisted reply cache.

Once the existing pending requests are disposed of, the client can proceed to doing new requests, although it might have to do lock recovery first. This can occur after a persistent reply cache is used to provide EOS or after it is found that there is no session persistence provided by the server.

8.6. Client Use of Clientid-based Persistence

At this point, lock recovery needs to begin if a new request is processed and completes returning the error NFS4ERR_STALECLIENTID. If no new requests have been issued at this point, the client can issue a request consisting only of a SEQUENCE operation to provide a test. If NFS4ERR_STALECLIENTID is not returned then the client will assume either that there has been no server restart or thar server restart as been accompanied with the recovery of locking state for the current clientid. Otherwise, lock recovery can be done as part of a server-provided grace period. The following three steps need to be taken:

When lock recovery is necessary, the client need to inform the new server of the existence of its locks before using stateids it obtained before the server restart. This process is referred to as reclaiming the client’s locks, which is accomplished using the method listed below, depending on the type of lock to be reclaimed.

* Opens can generally be reclaimed by doing an OPEN with the claim type CLAIM_PREVIOUS.

  This includes the case of opens associated with delegation. For details, see Section 15.2.1,

  There is no specific way to reclaim delegations that have no associated open. In such cases, the client can open the file asking for an associated delegation, and return it immediately

* To reclaim byte-range locks, a LOCK operation with the reclaim parameter set to true is used.

  The associated open will need to be reclaimed first.

* There is no provision regarding reclaiming of layouts and thus no way to obtain them during a grace period.

As a result, in case in which locking state is not made available by the server across a server failure, use of the data server is not immediately available and the client is best off doing IO through the MDS until obtaining needed layouts once the rest of lock reclamation is complete.

Once all reclaimable locks have been reclaimed, the client needs to do a global RECLAIM_COMPLETE to indicate that process is complete. The is necessary to allow new locks to be obtained. However, even after this done, such requests might still be rejected with NFS4ERR_GRACE if other clients have not completed their lock reclamations.

9. 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]) and RPC ([RFC5531]) 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 RFCTBD30.

9.1. 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          = 0xffffffffffffffff;
const NFS4_INT32_MAX           = 0x7fffffff;
const NFS4_UINT32_MAX          = 0xffffffff;

const NFS4_MAXFILELEN          = 0xffffffffffffffff;
const NFS4_MAXFILEOFF          = 0xfffffffffffffffe;
```

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

* NFS4_FHSIZE is the maximum size of a filehandle.

* NFS4_VERIFIER_SIZE is the fixed size of a verifier.

* NFS4_OPAQUE_LIMIT is the maximum size of certain opaque information.

* NFS4_SESSIONID_SIZE is the fixed size of a session identifier.

* NFS4_INT64_MAX is the maximum value of a signed 64-bit integer.

* NFS4_UINT64_MAX is the maximum value of an unsigned 64-bit integer.

* NFS4_INT32_MAX is the maximum value of a signed 32-bit integer.

* NFS4_UINT32_MAX is the maximum value of an unsigned 32-bit integer.

* NFS4_MAXFILELEN is the maximum length of a regular file.

* NFS4_MAXFILEOFF is the maximum offset into a regular file.

9.2. 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;

	The length of a byte-range within a file.
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 [RFC2743] 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 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<>; UTF-8 encoding for strings.
utf8str_cis	typedef utf8string utf8str_cis; Case-insensitive UTF-8 string.
utf8str_cs	typedef utf8string utf8str_cs; Case-sensitive UTF-8 string.
utf8str_mixed	typedef utf8string utf8str_mixed; UTF-8 strings with a domain or host prefix and an server or file name suffix. Domains can be internationalized as described in [I-D.ietf-nfsv4-internationalization].
utf8pref	typedef opaque utf8pref<>; String for which UTF-8 encoding is preferred, although other encodings can be used,
component4	typedef utf8pref component4; Represents pathname components, which may be either encoded using UTF-8 or nor, with use of UTF-8 needed to support normalization and case-insensitivity.
linktext4	typedef opaque linktext4<> Symbolic link contents ("symbolic link" is defined in an Open Group [symlink] standard).
pathname4	typedef component4 pathname4<>;

	Represents pathname 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.

Table 1

End of Base Data Types

9.3. Structured Data Types

9.3.1. nfstime4

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

The nfstime4 data type gives the number of seconds and nanoseconds since midnight or zero hour January 1, 1970 Coordinated Universal Time (UTC). Values greater than zero for the seconds field denote dates after the zero hour January 1, 1970. Values less than zero for the seconds field denote dates before the zero 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 zero hour January 1, 1970, the seconds field would have a value of negative one (-1) and the nseconds field 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 skew between client and server times.

9.3.2. time_how4

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

9.3.3. 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.

9.3.4. 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.

9.3.5. fsid4

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

9.3.6. change_policy4

```
   struct change_policy4 {
           uint64_t        cp_major;
           uint64_t        cp_minor;
   };
```

The change_policy4 data type is used for the change_policy OPTIONAL 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.

9.3.7. fattr4

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

The fattr4 data type is used to represent sets of protocol-defined attributes.

The bitmap is a counted array of 32-bit integers used to contain bit values. The position of the integer in the array that contains bit n can be computed from the expression (n / 32), and its bit within that integer is (n mod 32).

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

9.3.8. 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.

9.3.9. 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 na_r_netid and na_r_addr fields respectively contain a netid and uaddr. The netid and uaddr concepts are defined in [RFC5665]. The netid and uaddr formats for TCP over IPv4 and TCP over IPv6 are defined in [RFC5665], specifically Tables 2 and 3 and in Sections 5.2.3.3 and 5.2.3.4.

9.3.10. 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 9.3.10.1) and lock_owner4 (Section 9.3.10.2).

9.3.10.1. open_owner4

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

9.3.10.2. lock_owner4

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

9.3.11. 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.

9.3.12. 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 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.

9.3.13. 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 11.16.1). A client asks for layouts of a particular type in LAYOUTGET, and processes those layouts using 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 27.5; 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 18, is to be used. The LAYOUT4_OSD2_OBJECTS enumeration specifies that the object layout, as defined in [RFC5664], is to be used. Similarly, the LAYOUT4_BLOCK_VOLUME enumeration specifies that the block/volume layout, as defined in [RFC5663], is to be used.

9.3.14. 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 17.2.10 for more details.

9.3.15. 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 is interpreted based on the specified da_layout_type field.

This document defines the device address for the NFSv4.1 file layout (see Section 18.3), 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-based storage devices and block storage devices (e.g., Small Computer System Interface (SCSI) volume labels) are defined by their respective layout specifications.

9.3.16. layout_content4

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

The loc_body field is interpreted based on the layout type (loc_type). This document defines the loc_body for the NFSv4.1 file layout type; see Section 18.3 for its definition.

9.3.17. 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.

9.3.18. 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 23.42) operation. This data type provides a channel to pass layout type specific information (in field lou_body) back to the metadata server. For example, 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.

9.3.19. 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 11.16.4. 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 18.3.

9.3.20. 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.

9.3.21. 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 with which the implementer is associated. 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.

9.3.22. 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	If a file's length is less than the value of threshold4_read_size, then it is RECOMMENDED that the client read from the file via the MDS and not a storage device.
threshold4_write_size	1	length4	If a file's length is less than the value of threshold4_write_size, then it is RECOMMENDED that the client write to the file via the MDS and not a storage device.
threshold4_read_iosize	2	length4	For read I/O sizes below this threshold, it is RECOMMENDED that the client read data using the MDS.
threshold4_write_iosize	3	length4	For write I/O sizes below this threshold, it is RECOMMENDED that the client write data using the MDS.

Table 2

9.3.23. 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.

10. 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.

10.1. 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]), 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 pathname to a filehandle, which can then be used by the NFS protocols.

The MOUNT protocol has deficiencies in the area of security and use via firewalls. This is one reason that the use of the public filehandle was introduced in [RFC2054] and [RFC2055]. With the use of the public filehandle in combination with the LOOKUP operation in the 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 pathnames to a filehandle. Two special filehandles will be used as starting points for the NFS client.

10.1.1. Root Filehandle

The first of the special filehandles is the ROOT filehandle. The ROOT filehandle is the "conceptual" root of the file system namespace 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 namespace is in Section 12.

10.1.2. 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.

10.2. 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 that 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 that may be used by the client to determine the filehandle types being returned by the server.

10.2.1. 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 that might cause incorrect behavior. Further discussion of filehandle and attribute comparison in the context of data caching is presented in Section 15.3.4.

As an example, in the case that two different pathnames 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 (see [hardlink]) is used to create two file names that 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 pathnames' traversals.

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

10.2.3. 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 cases in which the continuity of filehandle use is not specified by handle class information within the fs_locations_info attribute. When this bit is set, clients without access to fs_locations_info information should assume that 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 that provide volatile filehandles that can 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 is not set, or if a non-read-only 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 12.5 for a discussion of this.

10.3. 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]

* slot is an index in the server volatile filehandle table

* 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 the volatile bit is 0, then it is a persistent filehandle with a different structure following it.

10.4. 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 namespace that is still available or by starting at the root of the server’s file system namespace.

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 pathname based on the processing of the rename request. The client can then regenerate the new filehandle based on the new pathname. The client could also use the COMPOUND procedure to construct a series 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.

11. File Attributes

To meet the requirements of extensibility and increased interoperability with non-UNIX platforms, attributes are being handled in a more flexible manner than NFSv3. The NFSv3 fattr3 structure consists of a fixed list of attributes some of which that might not all be supported by some potential servers and includes some attributes that not all clients have an interest in. The fattr3 structure and similar fixed structures cannot be extended as new needs arise and provide no way to indicate non-support of particular attributes. Within the NFSv4.1 protocol, the client is able to query what attributes the server supports and construct requests that deal only with those supported attributes (or a subset thereof). This raises the issues, discussed in Sections 11.1 through 11.3 and 11.5 through 11.6, of determining how the non-support of particular attributes is to be dealt with.

11.1. Categorization of File Attributes

In order to clarify the requirements for server support of particular attributes, and to provide guidance for clients dealing with non-support of particular attributes, all NFSv4.1 attributes are divided into the groups listed below:

All of these attributes are accommodated in the NFSv4.1 protocol by a specific, well-defined encoding and are identified by a number. They are interrogated by setting a bit in the bit vector sent in a GETATTR, request. The server response includes a bit vector to indicate which attributes were returned in the response.

The following attribute categories are defined:

* REQUIRED attributes, as discussed in Section 11.4.

* OPTIONAL attributes, as discussed in Section 11.5.

* Experimental attributes, as discussed in Section 11.6.

New attributes of any of these categories may be added to the NFSv4 protocol as part of a new minor version by publishing a Standards Track RFC that allocates a new attribute number value and defines the encoding for the attribute. In addition, new minor versions can move attributes between categories or make formerly OPTIONAL and Experimental attributes MANDATORY to NOT implement. Similarly, OPTIONAL attributes may be added to an existing extensible version by publishing a Standards Track RFC that allocates a new attribute number value and defines the encoding for the attribute. See [RFC8178] for further details

11.2. Changes in the Categorization of File Attributes

The categorization of file attributes appearing in this specification differs from that previously published for a number of reasons:

* The description of the attributes for which support is not REQUIRED no longer uses the RFC2119 keyword "RECOMMENDED" as this is not in accord with the definition of that term in [RFC2119].

  We now describe such attributes as OPTIONAL, leaving it to server to decide which are worthy of support and to clients to decide whether they wish to use servers on which they are not supported.

* The categorization of requirements/recommendation as to support for authorization-related attributes is now the responsibility of the NFSv4-wide security documents, to be derived from [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls].

  Currently, given the likely lack of agreement on the semantics of ACLs, it is likely that acl would best be described as an Experimental attribute. See Section 11.3 for further discussion.

As one illustration of the new approach to these matters, and its differences from older approaches, let us consider the following statement from Section 5.1 of [RFC8881]. Referring to the REQUIRED attributes, it states:

> The client is expected to 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. In the case of servers not supporting the owner, mode, or acl-related attributes, there would be no ability to provide substantial security-related functionality.

This expectation was not a reasonable one when first formulated and as the NFSv4 protocols have been developed, there have never been any cases of it being realized. There is no reason to implement a server without the minimal authorization-related attributes derived from NFSv3 and no point in working to develop clients capable of interoperating with it. There is no motivation for the working group to devote any time to defining how such a combination is to operate or for implementers to experiment to try to implement remote file access without any meaningful authorization process.

Further, the above also seems to conflict with the following, appearing in Section 5.2 of [RFC8881]:

> It is expected that servers will support all attributes they comfortably can and only fail to support attributes that are difficult to support in their operating environments.

Together, these imply that there are operating environments in which it difficult to support all of mode, owner, group, and acl attributes. It is hard to believe that any such environments exist or that there would be any point in implementing an NFSv4.1 server using then, if they did exist.

11.3. Categorization of Authorization-related Attributes

This section provides an overview of the issues in involved in appropriately categorizing the authorization-related attributes, although the final categorization of these will appear in NFSv4-wide security documents, expected to be based on [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls].

Authorization-related attributes that are part of NFSv4.1 can be divided into those connected to he POSIX-based authorization model used in NFSv3 and those related to the use of ACLs to provide a more flexible authorization model. Within the context of NFSv4.1, the following should be noted:

* The attributes mode, owner, and owner_group need to be considered REQUIRED, as they are in [I-D.dnoveck-nfsv4-security].

This is despite the fact that previous specifications have considered these attributes as OPTIONAL, although the word "RECOMMENDED" was sometime used. In any case, the new categorization in [I-D.dnoveck-nfsv4-acls] has to be considered dispositive both with regard to NFSv4.1 an other minor versions.

* The attributes acl, sacl, and dacl, although designated as OPTIONAL, have never been documented in a manner allowing effective client-server interoperability, suggesting that they would more appropriately be designated as "Experimental".

  While it is possible that tightening of the specifications being done in [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls] as part of the rfc8881bis effort might allow this to change, that is not yet assured

  In any case, efforts to provide a path to interoperability will continue and might affect this categorization in later minor versions, even if NFSv4.1 is not affected. See [I-D.dnoveck-nfsv4-acls] for details,

* The attribute aclsupport is appropriately designated as OPTIONAL, as it in [I-D.dnoveck-nfsv4-security].

11.4. 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 when requested. A client may ask for the value of any of these attributes to be returned by setting a bit in the GETATTR request, and the server MUST return their value.

The client is expected to be able to function with an attribute set limited to these attributes. With just the REQUIRED attributes some client functionality may be unavailable or functionally limited .

11.5. OPTIONAL Attributes

These attributes are understood well enough to warrant support in the NFSv4.1 protocol. However, they might not be supported on all servers or used by all clients. A client may ask for any of these attributes to be returned by setting a bit in the GETATTR request but need to handle the case where the server does not return them. A client MAY ask for the set of attributes the server supports within a given file system and has no reason to request attributes the server does not support. A server is REQUIRED to be deal with requests for unsupported attributes by not returning values for them rather than by considering the request an error.

Previous versions of the NFSv4.1 specification [RFC5661] [RFC8881] have described these attributes as "RECOMMENDED" even though that description is not accord with [RFC2119]. The NFSv4.0 specification [RFC7530] still uses "RECOMMENDED" although explicitly disclaiming the assumption that the RFC2119 definition applies in this case. The description of these attribute as OPTIONAL connects them appropriately to provisions for protocol extension and minor versioning in which attributes are to be treated as OPTIONAL.

11.6. Experimental Attributes

While the vast majority of attributes are, as described in Section 11.5, "understood well enough to warrant support in the NFSv4.1 protocol", it appears to be the case that, for several attributes, that understanding was never properly recorded in existing NFSv4.1 specification documents. While it might be possibly to rectify that issue before eventual publication of this document, the likely existence of multiple incompatible implementations of such attributes make that unlikely

Although the existence of such attributes has never been acknowledged before as part of the categorization of NFSv4 attributes. Nevertheless, such attributes have existed in all NFSv4 minor versions and the necessary clarification, if it occurs, is not likely to be complete for some time.

While the intention has always been that attribute not be included in Proposed Standards unless they are described adequately to allow interoperable implementation to be developed. Despite that intention, such attributes have been included in multiple minor versions. Given the need to correct that situation, we need to be clear about the issues that have led to these unfortunate situations, so that we can, over time, address them.

11.7. Named Attributes

These attributes are not supported by direct encoding in the NFSv4 protocol but are accessed using string names rather than numbers and each corresponds to an uninterpreted stream of bytes that is stored in its own file system object. The namespace for these attributes may be accessed by using the OPENATTR operation as described below. The OPENATTR operation returns a filehandle for a "named attribute directory", and further perusal and modification of the namespace may be done using operations that work on more typical directories, subject to restrictions discussed below. 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 can be accomplished using an OPEN specifying file creation.

OPENATTR takes a filehandle for the object and returns the filehandle for the attribute directory. The filehandle for the named attributes designates a directory object accessible by LOOKUP or READDIR and contains files whose names identify the named attributes and whose data bytes are the value of those attributes. 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

Table 3

Named attributes are intended for data needed by applications rather than by NFS client implementations. NFS implementers who wish to define new attributes need to specify them as OPTIONAL attributes using the protocol extension facilities specified in [RFC8178].

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

Named attributes may have their own (non-named) attributes. Each of these objects MUST have all of the REQUIRED attributes and may have additional attributes which are not REQUIRED. However, the sets of supported attributes for named attributes need not be, and typically will not be, as large as that for other objects in that file system. Nevertheless, the value of the supported_attrs attribute should reflect the supported attributes for the file system and will not reflect the restricted attribute sets for these special objects.

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

Support for named attributes is OPTIONAL and clients need to be prepared to deal with servers that do not support them. However, clients are entitled to assume that if OPENATTR is supported, there will be support for arbitrarily named attributes, rather than support

for a few specific names known to the server. If a server does support named attributes, a client that is also able to handle them should be able to copy a file's data and metadata with complete transparency from one location to another since names allowed for regular directory entries are expected to be 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, non-hierarchal structure for named attributes. In such a mixed environment, clients or applications might come to depend on non-portable extensions. The restrictions are:

* 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 a hierarchical structure of named attributes for a single object is not allowed.

* If OPENATTR is done on a named attribute directory or on a named attribute, the server MUST return NFS4ERR_WRONG_TYPE.

* 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.

* 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, beyond what is necessary to regulate the names of files within directories to handle internationalization and case-insensitivity. See Section 27.2 for further discussion.

11.8. Classification of Attributes

Each of the protocol-defined attributes can be classified in one of three categories: per server (i.e., the value of the attribute will be the same for all file objects that share the same server owner; see Section 5.6 for a definition of server owner), per file system (i.e., the value of the attribute will be the same for some or all file objects that share the same fsid attribute (Section 11.12.1.9) and server owner), or per file system object. Note that it is possible that some per file system attributes may vary within the

file system, depending on the value of the "homogeneous" (Section 11.12.2.16) attribute. Note that the attributes time_access_set and time_modify_set are not listed in this section because they are write-only attributes corresponding to time_access and time_modify, and are used in a special instance of SETATTR.

* The per-server attribute is:

  lease_time

* The per-file system attributes are:

  supported_attrs, suppattr_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

* The per-file system object attributes are:

  type, change, size, named_attr, fsid, rdattr_error, filehandle, acl, archive, fileid, hidden, maxlink, mimetype, mode, numlinks, owner, owner_group, rawdev, space_used, system, time_access, time_backup, time_create, time_metadata, time_modify, mounted_on_fileid, dir_notif_delay, dirent_notif_delay, dacl, sacl, layout_type, layout_hint, layout_blksize, layout_alignment, mdsthreshold, retention_get, retention_set, retentevt_get, retentevt_set, retention_hold, mode_set_masked

For quota_avail_hard, quota_avail_soft, and quota_used, see their definitions below for the appropriate classification.

11.9. Set-Only and Get-Only Attributes

Some of the protocol-defined attributes are set-only; i.e., they can be set via SETATTR but not retrieved via GETATTR. Similarly, some protocol-defined attributes are get-only; i.e., they can be retrieved via GETATTR but not set via SETATTR. If a client attempts to set a get-only attribute or get a set-only attributes, the server MUST return NFS4ERR_INVAL.

11.10. REQUIRED Attributes - List and Definition References

The list of REQUIRED attributes appears in Table 4. The meaning of the columns of the table are:

Name: The name of the attribute.

Id: The number assigned to the attribute. In the event of conflicts between the number assigned here and in RFCTBD30, the latter is likely authoritative, but should be resolved with Errata to this document and/or RFCTBD30. See [errata] for the Errata process.

Data Type: The XDR data type of the attribute.

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).

Defined in: The section of this specification that describes the attribute.

Name	Id	Data Type	Acc	Defined in:
supported_attrs	0	bitmap4	R	Section 11.12.1.1
type	1	nfs_ftype4	R	Section 11.12.1.2
fh_expire_type	2	uint32_t	R	Section 11.12.1.3
change	3	uint64_t	R	Section 11.12.1.4
size	4	uint64_t	R W	Section 11.12.1.5
link_support	5	bool	R	Section 11.12.1.6
symlink_support	6	bool	R	Section 11.12.1.7
named_attr	7	bool	R	Section 11.12.1.8

fsid	8	fsid4	R	Section 11.12.1.9
unique_handles	9	bool	R	Section 11.12.1.10
lease_time	10	nfs_lease4	R	Section 11.12.1.11
rdattr_error	11	enum	R	Section 11.12.1.12
filehandle	19	nfs_fh4	R	Section 11.12.1.13
mode	33	mode4	R W	Section 11.18
owner	36	utf8str_mixed	R W	Section 11.18
owner_group	37	utf8str_mixed	R W	Section 11.18
suppattr_exclcreat	75	bitmap4	R	Section 11.12.1.14

Table 4

11.11. OPTIONAL Attributes - List and Definition References

The OPTIONAL attributes are defined in Table 5. The meanings of the column headers are the same as Table 4; see Section 11.10 for the meanings.

Name	Id	Data Type	Acc	Defined in:
acl	12	nfsace4<>	R W	Section 11.18
aclsupport	13	uint32_t	R	Section 11.18
archive	14	bool	R W	Section 11.12.2.1

cansettime	15	bool	R	Section 11.12.2.2
case_insensitive	16	bool	R	Section 11.12.2.3
case_preserving	17	bool	R	Section 11.12.2.4
change_policy	60	chg_policy4	R	Section 11.12.2.5
chown_restricted	18	bool	R	Section 11.12.2.6
dacl	58	nfsacl41	R W	Section 11.18
dir_notif_delay	56	nfstime4	R	Section 11.15.1
dirent_notif_delay	57	nfstime4	R	Section 11.15.2
fileid	20	uint64_t	R	Section 11.12.2.7
files_avail	21	uint64_t	R	Section 11.12.2.8
files_free	22	uint64_t	R	Section 11.12.2.9
files_total	23	uint64_t	R	Section 11.12.2.10
fs_charset_cap	76	uint32_t	R	Section 11.12.2.11
fs_layout_type	62	layouttype4<>	R	Section 11.16.1
fs_locations	24	fs_locations	R	Section 11.12.2.12
fs_locations_info	67	fs_locations_info4	R	Section 11.12.2.13

fs_status	61	fs4_status	R	Section 11.12.2.14
hidden	25	bool	R W	Section 11.12.2.15
homogeneous	26	bool	R	Section 11.12.2.16
layout_alignment	66	uint32_t	R	Section 11.16.2
layout_blksize	65	uint32_t	R	Section 11.16.3
layout_hint	63	layouthint4	W	Section 11.16.4
layout_type	64	layouttype4<>	R	Section 11.16.5
maxfilesize	27	uint64_t	R	Section 11.12.2.17
maxlink	28	uint32_t	R	Section 11.12.2.18
maxname	29	uint32_t	R	Section 11.12.2.19
maxread	30	uint64_t	R	Section 11.12.2.20
maxwrite	31	uint64_t	R	Section 11.12.2.21
mdsthreshold	68	mdsthreshold4	R	Section 11.16.6
mimetype	32	utf8str_cs	R W	Section 11.12.2.22
mode	33	mode4	R W	Section 11.18
mode_set_masked	74	mode_masked4	W	Section 11.18

mounted_on_fileid	55	uint64_t	R	Section 11.12.2.23
no_trunc	34	bool	R	Section 11.12.2.24
numlinks	35	uint32_t	R	Section 11.12.2.25
quota_avail_hard	38	uint64_t	R	Section 11.12.2.26
quota_avail_soft	39	uint64_t	R	Section 11.12.2.27
quota_used	40	uint64_t	R	Section 11.12.2.28
rawdev	41	specdata4	R	Section 11.12.2.29
retentevt_get	71	retention_get4	R	Section 11.17.3
retentevt_set	72	retention_set4	W	Section 11.17.4
retention_get	69	retention_get4	R	Section 11.17.1
retention_hold	73	uint64_t	R W	Section 11.17.5
retention_set	70	retention_set4	W	Section 11.17.2
sacl	59	nfsacl41	R W	Section 11.18
space_avail	42	uint64_t	R	Section 11.12.2.30
space_free	43	uint64_t	R	Section 11.12.2.31
space_total	44	uint64_t	R	Section 11.12.2.32

space_used	45	uint64_t	R	Section 11.12.2.33
system	46	bool	R W	Section 11.12.2.34
time_access	47	nfstime4	R	Section 11.12.2.35
time_access_set	48	settime4	W	Section 11.12.2.36
time_backup	49	nfstime4	R W	Section 11.12.2.37
time_create	50	nfstime4	R W	Section 11.12.2.38
time_delta	51	nfstime4	R	Section 11.12.2.39
time_metadata	52	nfstime4	R	Section 11.12.2.40
time_modify	53	nfstime4	R	Section 11.12.2.41
time_modify_set	54	settime4	W	Section 11.12.2.42

Table 5

11.12. Attribute Definitions

11.12.1. Definitions of REQUIRED Attributes

11.12.1.1. Attribute 0: supported_attrs

The bit vector that would retrieve all protocol-defined attributes that are supported for this object. The scope of this attribute applies to all objects with a matching fsid.

11.12.1.2. Attribute 1: type

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

* NF4REG designates a regular file.

* NF4DIR designates a directory.

* NF4BLK designates a block device special file.

* NF4CHR designates a character device special file.

* NF4LNK designates a symbolic link.

* NF4SOCK designates a named socket special file.

* NF4FIFO designates a fifo special file.

* NF4ATTRDIR designates a named attribute directory.

* NF4NAMEDATTR designates a named attribute.

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

* The phrase "is a directory" means that the object's type attribute is NF4DIR or NF4ATTRDIR.

* The phrase "is a special file" means that the object's type attribute is NF4BLK, NF4CHR, NF4SOCK, or NF4FIFO.

* The phrases "is an ordinary file" and "is a regular file" mean that the object's type attribute is NF4REG or NF4NAMEDATTR.

11.12.1.3. Attribute 2: fh_expire_type

Server uses this to specify filehandle expiration behavior to the client. See Section 10 for additional description.

11.12.1.4. Attribute 3: change

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

11.12.1.5. Attribute 4: size

The size of the object in bytes.

11.12.1.6. Attribute 5: link_support

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

11.12.1.7. Attribute 6: symlink_support

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

11.12.1.8. Attribute 7: named_attr

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

11.12.1.9. Attribute 8: fsid

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

11.12.1.10. Attribute 9: unique_handles

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

11.12.1.11. Attribute 10: lease_time

Duration of the lease at server in seconds.

11.12.1.12. Attribute 11: rdattr_error

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

11.12.1.13. Attribute 19: filehandle

The filehandle of this object (primarily for use by READDIR requests).

11.12.1.14. Attribute 75: suppattr_exclcreat

The bit vector that would set all protocol-defined 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.

11.12.2. Definitions of Uncategorized OPTIONAL Attributes

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

11.12.2.1. Attribute 14: archive

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

11.12.2.2. Attribute 15: cansettime

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

11.12.2.3. Attribute 16: case_insensitive

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

11.12.2.4. Attribute 17: case_preserving

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

11.12.2.5. 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 9.3.6 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.

11.12.2.6. 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).

11.12.2.7. Attribute 20: fileid

A number uniquely identifying the file within the file system.

11.12.2.8. Attribute 21: files_avail

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

11.12.2.9. Attribute 22: files_free

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

11.12.2.10. Attribute 23: files_total

Total file slots on the file system containing this object.

11.12.2.11. Attribute 76: fs_charset_cap

Character set capabilities for this file system. See Section 19.1.

11.12.2.12. 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 16.16 for more details.

11.12.2.13. Attribute 67: fs_locations_info

Full function file system location. See Section 16.17.2 for more details.

11.12.2.14. Attribute 61: fs_status

Generic file system type information. See Section 16.18 for more details.

11.12.2.15. Attribute 25: hidden

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

11.12.2.16. Attribute 26: homogeneous

TRUE, if this object's file system is homogeneous; i.e., all objects in the file system (all objects on the server with the same fsid) have common values for all per-file-system attributes.

11.12.2.17. Attribute 27: maxfilesize

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

11.12.2.18. Attribute 28: maxlink

Maximum number of links for this object.

11.12.2.19. Attribute 29: maxname

Maximum file name size supported for this object.

11.12.2.20. Attribute 30: maxread

Maximum amount of data the READ operation will return for this object.

11.12.2.21. Attribute 31: maxwrite

Maximum amount of data the WRITE operation will accept for this object. This attribute SHOULD be supported if the file is writable. Lack of this attribute can lead to the client either wasting bandwidth or not receiving the best performance.

11.12.2.22. Attribute 32: mimetype

MIME body type/subtype of this object.

11.12.2.23. 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 that 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 OPTIONAL, and the server should provide it if possible. For a UNIX-based server, this is straightforward. Usually, mounted_on_fileid will be requested as part of 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.

11.12.2.24. 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.

11.12.2.25. Attribute 35: numlinks

Number of hard links to this object.

11.12.2.26. Attribute 38: quota_avail_hard

The value in bytes that represents the amount of additional disk space beyond the current allocation that can be allocated to this file or directory before further allocations will be refused. It is understood that this space may be consumed by allocations to other files or directories.

11.12.2.27. Attribute 39: quota_avail_soft

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

11.12.2.28. Attribute 40: quota_used

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

Note that there may be a number of distinct but overlapping sets of files or directories for which a quota_used value is maintained, e.g., "all files with a given owner", "all files with a given group owner", etc. The server is at liberty to choose any of those sets when providing the content of the quota_used attribute, but should do so in a repeatable way. The rule may be configured per file system or may be "choose the set with the smallest quota".

11.12.2.29. Attribute 41: rawdev

Raw device number of file of type NF4BLK or NF4CHR. The device number is split into major and minor numbers. If the file's type attribute is not NF4BLK or NF4CHR, the value returned SHOULD NOT be considered useful.

11.12.2.30. 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.

11.12.2.31. Attribute 43: space_free

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

11.12.2.32. Attribute 44: space_total

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

11.12.2.33. Attribute 45: space_used

Number of file system bytes allocated to this object.

11.12.2.34. Attribute 46: system

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

11.12.2.35. Attribute 47: time_access

The time_access attribute represents the time of last access to the object by a READ operation sent to the server. The notion of what is an "access" depends on the server's operating environment and/or the server's file system semantics. For example, for servers obeying Portable Operating System Interface (POSIX) semantics, time_access would be updated only by the READ and READDIR operations and not any of the operations that modify the content of the object [read_atime], [readdir_atime], [write_atime]. Of course, setting the corresponding time_access_set attribute is another way to modify the time_access attribute.

Whenever the file object resides on a writable file system, the server should make its best efforts to record time_access into stable storage. However, to mitigate the performance effects of doing so, and most especially whenever the server is satisfying the read of the object's content from its cache, the server MAY cache access time

updates and lazily write them to stable storage. It is also acceptable to give administrators of the server the option to disable time_access updates.

11.12.2.36. Attribute 48: time_access_set

Sets the time of last access to the object. SETATTR use only.

11.12.2.37. Attribute 49: time_backup

The time of last backup of the object.

11.12.2.38. 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".

11.12.2.39. Attribute 51: time_delta

Smallest useful server time granularity.

11.12.2.40. Attribute 52: time_metadata

The time of last metadata modification of the object.

11.12.2.41. Attribute 53: time_modify

The time of last modification to the object.

11.12.2.42. Attribute 54: time_modify_set

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

11.13. Interpreting owner and owner_group

The attributes "owner" and "owner_group" (and also users and groups within the "acl" attribute) are transferred in the form of a UTF-8 string. This string can be used to identify users and groups in several ways:

* A string of the form "name@domain" can be used to give a user or group name together with a domain in which those are defined.

  This form provides greater degree of extensibility than was possible in NFSv3 which limited these identifiers to 32-bit unsigned integers whose values are all centrally administered as members within a common domain.

* Numeric ids converted to string form.

  Using this format maintains the strengths and weaknesses of the NFSv3 approach.

The following issues are relevant in selected the form to use.

* The use of the form "name@domain" provides greater flexibility, both with regard to the number of users that can be accommodated and to the management of multiple sets of users in separate domains.

  Taking advantage of this flexibility often requires extensive work because of limitations of the API's used to reference users and groups.

* The use of the form "name@domain" allows clients and servers to work together even if they have different internal formats for user and groups.

  In many cases, there is no need for such mapping.

  Providing this mapping requires extra implementation and raises potential security issues.

For detailed discussions regarding which of the forms clients and server are to use for these values, see Section 5.1 of [I-D.dnoveck-nfsv4-security].

11.14. Character Case Attributes

With respect to the case_insensitive and case_preserving attributes, each UCS-4 character (which UTF-8 encodes) can be mapped to an equivalent character of different case or compared in a case-insensitive manner. The details vary based on the Unicode version implemented by the server for the current file system. Details of the process and how the client can best deal with uncertainty about the process will be discussed in the NFSv4-wide internationalization document (See [I-D.ietf-nfsv4-internationalization] for the latest version)

11.15. Directory Notification Attributes

As described in Section 23.39, 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 sending a GETATTR operation 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.

11.15.1. 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.

11.15.2. 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.

11.16. pNFS Attribute Definitions

11.16.1. Attribute 62: fs_layout_type

The fs_layout_type attribute (see Section 9.3.13) 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.

11.16.2. 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.

11.16.3. 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.

11.16.4. Attribute 63: layout_hint

The layout_hint attribute (see Section 9.3.19) 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 subset of the layout structure returned by LAYOUTGET. For example, instead of specifying particular devices, this would be used to suggest the stripe width of a file. The server implementation determines which fields within the layout will be used.

11.16.5. 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.

11.16.6. 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 sending its read or write requests to the data server. Alternatively, if only one of the specified thresholds is reached or exceeded, the I/O requests are sent to the metadata server.

For each threshold type, a value of zero indicates no READ or WRITE should be sent to the metadata server, while a value of all ones indicates that 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.

11.17. 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, OPTIONAL, 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.

11.17.1. 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 11.9). 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.

11.17.2. 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 11.9). 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 or the modification of the retention_hold attribute.

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

* As long as retention is not enabled, the client is permitted to decrease the duration.

* 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 11.17.1 or Section 11.17.3) 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.

* While retention is enabled, attempts to disable retention or decrease the retention's duration MUST fail with the error NFS4ERR_INVAL.

* 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.

11.17.3. Attribute 71: retentevt_get

Gets 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.

11.17.4. Attribute 72: retentevt_set

Sets the event-based retention duration, and optionally enables event-based retention on the file object. This attribute corresponds to retentevt_get and 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 or the modification of the retention_hold attribute.

11.17.5. Attribute 73: retention_hold

Gets or sets 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.

11.18. Access Control Attributes

The use of the access control attributes are fully described in various sections of the NFSv4-wide security documents [I-D.dnoveck-nfsv4-security] [I-D.dnoveck-nfsv4-acls].

* The mode, mode_set_masked, owner, and owner_group attributes are described in Sections 5.3.1 though 5.3.4 of [I-D.dnoveck-nfsv4-security].

* The acl, aclsupport, sacl, and dacl attributes are described in Sections 3.4, 3.5, 3.6, and 3.8 of [I-D.dnoveck-nfsv4-acls].

12. Single-Server Namespace

This section 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 16.

12.1. 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 the server’s exports.

12.2. 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 12.3) 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.

12.3. 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 the 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.

12.4. 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.

12.5. 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.

12.6. Exported Root

If the server's root file system is exported, one might conclude that a pseudo file system is unneeded. This is not necessarily so. Assume the following file systems on a server:

```
/       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.

12.7. Mount Point Crossing

The server file system environment may be constructed in such a way that one file system contains a directory that 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.

12.8. 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_NO_NAME, the server SHOULD NOT present a different view of the namespace based on the security mechanism being used by a client. Instead, it should present a consistent view and return NFS4ERR_WRONGSEC if an attempt is made to access data with an inappropriate security mechanism.

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

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

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

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

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

13. 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 (Network Lock Manager) [xnfs]. These features include expanded locking facilities, which provide some measure of inter-client exclusion, but the state also offers features not readily providable using a stateless model. There are three components to making this state manageable:

* clear division between client and server

* ability to reliably detect inconsistency in state between client and server

* simple and robust recovery mechanisms

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

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

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

13.1. Client and Session ID

A client must establish a client ID (see Section 5.5) and then one or more sessionids (see Section 7) before performing any operations to open, byte-range 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.

13.2. 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 that represents the claim of those stateids and the objects they represent to be maintained by the server. See Section 13.3 for a discussion of the lease.

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.

13.2.1. Stateid Types

With the exception of special stateids (see Section 13.2.3), 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.

* 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.

* Stateids may represent sets of byte-range locks.

  All locks held on a particular file by a particular owner and 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.

* Stateids may represent file delegations, which are recallable guarantees by the server to the client that other clients will not reference or 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.

* 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.

* 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 of byte-ranges change, via layout stateid changing operations such as LAYOUTGET and LAYOUTRETURN.

13.2.2. 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 13.2.3), 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 that modify the set of locks, the server is required to increment the "seqid" field by one 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 17.5.3), 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 cancelled before the client even received an indication that an upgrade had happened.

When a stateid is sent by the server to the 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 latter, despite the fact that it is numerically greater.

13.2.3. 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:

* 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 18.6).

* 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 operations. This stateid MUST NOT be used on operations to data servers.

* 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 that 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.

* 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 that has all zeros 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 the current filehandle does not match that associated with the current stateid, the operation to which the stateid is passed will return NFS4ERR_BAD_STATEID.

13.2.4. 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, as long as the client ID is valid, 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 frees the stateids via the FREE_STATEID operation.

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.

* an index into a table of locking-state structures.

* a generation number that is incremented on each allocation of a table entry for a particular use.

And then store in each table entry,

* the client ID with which the stateid is associated.

* the current generation number for the (at most one) valid stateid sharing this index value.

* the filehandle of the file on which the locks are taken.

* an indication of the type of stateid (open, byte-range lock, file delegation, directory delegation, layout).

* the last "seqid" value returned corresponding to the current "other" value.

* 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 13.2.3 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 that 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:

* If the "other" and "seqid" fields do not match a defined combination associated with a special stateid, the error NFS4ERR_BAD_STATEID is returned.

* 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 are performed.

* 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.

* 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 nor 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.

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

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

* If the selected table entry does not match the current filehandle, return NFS4ERR_BAD_STATEID.

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

* If the stateid represents revoked state, then return NFS4ERR_EXPIRED, NFS4ERR_ADMIN_REVOKED, or NFS4ERR_DELEG_REVOKED, as appropriate.

* 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 that doesn’t represent byte-range locks is passed to the non-from_open case of LOCK or to LOCKU, or when a stateid that does not represent an open is passed to CLOSE or OPEN_DOWNGRADE. In such cases, the server MUST return NFS4ERR_BAD_STATEID.

* If the "seqid" field is not zero and it is greater than the current sequence value corresponding to the current "other" field, return NFS4ERR_BAD_STATEID.

* If the "seqid" field is not zero and it is less than the current sequence value corresponding to the current "other" field, return NFS4ERR_OLD_STATEID.

* 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, e.g., open-owner and lock-owner information, as well as information on the specific locks, e.g., open modes and byte-ranges.

13.2.5. 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 sending the I/O requests. SETATTR operations that change the file size are treated like I/O operations in this regard.

The following rules, applied in order of decreasing priority, govern the selection of the appropriate stateid. In following these rules, the client will only consider locks of which it has actually received notification by an appropriate operation response or callback. Note that the rules are slightly different in the case of I/O to data servers when file layouts are being used (see Section 18.10.1).

* If the client holds a delegation for the file in question, the delegation stateid SHOULD be used.

* Otherwise, if the entity corresponding to the lock-owner (e.g., a process) sending the I/O has a byte-range lock stateid for the associated open file, then the byte-range lock stateid for that lock-owner and open file SHOULD be used.

* If there is no byte-range lock stateid, then the OPEN stateid for the open file in question SHOULD be used.

* Finally, if none of the above apply, then a special stateid SHOULD be used.

Ignoring these rules may result in situations in which the server does not have information necessary to properly process the request. For example, when mandatory byte-range locks are in effect, if the stateid does not indicate the proper lock-owner, via a lock stateid, a request might be avoidably rejected.

The server however should not try to enforce these ordering rules and should use whatever information is available to properly process I/O requests. In particular, when a client has a delegation for a given file, it SHOULD take note of this fact in processing a request, even if it is sent with a special stateid.

13.2.6. 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 that set the file size, the client may send either a special stateid or, when a delegation is held for the file in question, a delegation stateid. While the server SHOULD validate the stateid and may use the stateid to optimize the determination as to whether a delegation is held, it SHOULD note the presence of a delegation even when a special stateid is sent, and MUST accept a valid delegation stateid when sent.

13.3. 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:

* When trunking is in effect, the client should consider sending 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.

* 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 13.4.2.1. 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 backoff, such that the maximum retransmission timeout exceeds 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, 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 reconnect.

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 23.46.3 for details).

* The status bits SEQ4_STATUS_CB_PATH_DOWN and SEQ4_STATUS_CB_PATH_DOWN_SESSION indicate problems with the backchannel that the client may need to address in order to receive callback requests.

* The status bits SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRING and SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRED indicate problems with GSS contexts or RPCSEC_GSS handles for the backchannel that the client might have to address in order to allow callback requests to be sent.

* 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.

* The status bit SEQ4_STATUS_LEASE_MOVE indicates that responsibility for lease renewal has been transferred to one or more new servers.

* The status bit SEQ4_STATUS_RESTART_RECLAIM_NEEDED indicates that due to server restart the client must reclaim locking state.

* The status bit SEQ4_STATUS_BACKCHANNEL_FAULT indicates that 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).

13.4. 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:

* 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.

* Subsequent recovery of locks may make execution of the operation inappropriate (NFS4ERR_GRACE).

13.4.1. 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 13.3, 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 23.36.3 for a description of 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 that are associated with the old client ID that 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.

13.4.2. 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, if mandatory locks are a possibility, 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 that 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 13.1) and re-establish its lock state with the new client ID, after the CREATE_SESSION operation succeeds (see Section 13.4.2.1).

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 13.1) and re-establish its lock state after the CREATE_SESSION, with the new client ID, succeeds (Section 13.4.2.1).

3. When an 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 13.1) and re-establish its lock state (Section 13.4.2.1).

13.4.2.1. 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 "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 operations with reclaim set to TRUE and OPEN operations with a claim type of CLAIM_PREVIOUS; see Section 14.11) 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 might get an NFS4ERR_GRACE status result from each such operation until the period of special handling is over. See Section 16.11.9 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 any 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. In addition, READ and WRITE requests that are not associated with a reclaimed OPEN need to be rejected as well.

The grace period may last until all clients that are known to have possibly had locks have done a global RECLAIM_COMPLETE operation, indicating that they have finished reclaiming the locks they held before the server restart. This means that a client that 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 clients that 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 without all expected RECLAIM_COPLETEs 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 sending requests on associated sessions with a frequency governed by the lease time. Note that when a client does not send such requests (or they are sent 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 16.11.9.

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 not subsumed within reclaimed OPENs 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 locking operation, or a READ or WRITE outside a reclaimed OPEN 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 can 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 operations 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 operations may be validly processed during the grace period because of the fact that 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 [Floyd]. 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:

* 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.

* 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 7.5) and not to control lock reclaim.

13.4.2.1.1. Security Issues for State Reclaim

During the grace period, a client can reclaim state that it believes or asserts it had before the server restarted. Unless the server has maintained a complete record of all the state the client had, the server has little choice but to trust the client's requests. (Of course, if the server maintained a complete record, then there would be no need to force the client to reclaim state after server restart.) While the server has to trust the client to tell the truth, the negative consequences for security are limited to enabling denial-of-service attacks in situations in which AUTH_SYS, particularly AUTH_SYS in the clear, is supported. The fundamental rule for the server when processing reclaim requests is that it MUST NOT grant the reclaim if an equivalent non-reclaim request would not be granted during steady state due to access control or access conflict issues. For example, an OPEN request during a reclaim will be refused with NFS4ERR_ACCESS if the principal making the request does not have sufficient access to open the file according to the acl, dacl, or mode attributes of the file.

Nonetheless, it is possible that a client operating in error or maliciously could, during reclaim, prevent another client from reclaiming access to state. For example, an attacker could send an OPEN reclaim operation with a deny mode that prevents another client from reclaiming the OPEN state it had before the server restarted. The attacker could perform the same denial of service during steady state prior to server restart, as long as the attacker had permissions. Given that the attack vectors are equivalent, the grace period does not offer any additional opportunity for denial of service, and any concerns about this attack vector, whether during grace or steady state, are addressed in the same way, by using RPCSEC_GSS for authentication and limiting access to the file only to principals that the owner of the file trusts.

Note that if prior to restart the server had client IDs with the EXCHGID4_FLAG_BIND_PRINC_STATEID (Section 23.35) capability set, then the server SHOULD record in stable storage the client owner id and the principal that established the client ID via EXCHANGE_ID. If the server does not do so, then there is a risk a client will be unable to reclaim state if it does not have a credential for a principal that was originally authorized to establish the state.

13.4.3. 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 to avoid interfering 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 client's 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 as 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, the client will attempt to reclaim locks. Normally, the server will not allow the client to reclaim locks, because the server will not be in its recovery grace period.

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 a 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 either that the server always assumes after it restarts that some edge condition occurs, and thus returns 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 some minimal information. For example, a server implementation could, for each client, save in stable storage a record containing:

* the co_ownerid field from the client_owner4 presented in the EXCHANGE_ID operation.

* a boolean that indicates if the client's lease expired or if there was administrative intervention (see Section 13.5) to revoke a byte-range lock, share reservation, or delegation and there has been no acknowledgment, via FREE_STATEID, of such revocation.

* a boolean that indicates whether the client may have locks that it believes to be reclaimable in situations in 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 that it is not known if there are other edge conditions.

   In the event that, after a server restart, the server determines there is unrecoverable damage or corruption to the information in stable storage, then for all clients and/or locks that 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 for which the client is trying to reclaim state, and use that to determine whether to re-establish the state via normal OPEN or LOCK operations. This is acceptable provided that the client's operating environment allows it. In other words, the client implementer is advised to document for his users the behavior. The client could also inform the application that its byte-range lock or share reservations (whether or not they were delegated) have been lost, such as via a UNIX signal, a Graphical User Interface (GUI) pop-up window, etc. See Section 15.5 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 13.5.

13.5. 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 13.4.2.1.

The second occasion of lock revocation is the inability to renew the lease before expiration, as discussed in Section 13.4.3. 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 23.46.3) 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 occurs, 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.

13.6. Short and Long Leases

When determining the time period for the server lease, the usual lease trade-offs apply. A short lease is 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). A long lease is 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 a long lease 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, the longer period for a lease to expire will force conflicting requests to wait longer.

A long lease is practical if the server can store lease state in stable storage. Upon recovery, the server can reconstruct the lease state from its stable storage and continue operation with its clients.

13.7. 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.

13.8. 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 that provide 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.

* SETCLIENTID since its function has been replaced by EXCHANGE_ID.

* SETCLIENTID_CONFIRM since client ID confirmation now happens by means of CREATE_SESSION.

* OPEN_CONFIRM because state-owner-based seqids have been replaced by the sequence ID in the SEQUENCE operation.

* 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.

* 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 1. They were used in minor version 0 to perform functions now provided in a different fashion.

* Sequence ids used to sequence requests for a given state-owner and to provide retry protection, now provided via sessions.

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

Such vestigial fields in existing operations 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.

14. File Locking and Share Reservations

To support Win32 share reservations, it is necessary to provide operations that 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 that is capable of atomically looking up, creating, and locking a file on the server.

14.1. 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 LOCK operation contains the heavyweight information required to establish a byte-range lock and uniquely define the owner of the lock.

14.1.1. State-Owner Definition

When opening a file or requesting a byte-range lock, the client must specify an identifier that 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 that, when concatenated with the current client ID, uniquely defines the owner of a 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).

14.1.2. Use of the Stateid and Locking

All READ, WRITE, and SETATTR operations contain a stateid. For the purposes of this section, SETATTR operations that change the size attribute of a file are treated as if they are writing the area between the old and new sizes (i.e., the byte-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 operation 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 a byte-range lock or a share reservation, a special stateid for anonymous state (zero as the value for "other" and "seqid") is used. (See Section 13.2.3 for a description of ’special’ stateids in general.) Regardless of 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 for which it knows it has the proper share reservation, it will need to

send a LOCK operation on the byte-range of the file that includes the byte-range the I/O was to be performed on, with an appropriate locktype field of the LOCK operation's arguments (i.e., READ*_LT for a READ operation, WRITE*_LT for a WRITE operation).

Note that for UNIX environments that support mandatory byte-range locking, the distinction between advisory and mandatory locking is subtle. In fact, advisory and mandatory byte-range locks are exactly the same as 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_LT) or exclusive (WRITE_LT) byte-range lock on the byte-range it wishes to READ from 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 operation 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 LOCK operation does not need to distinguish between advisory and mandatory byte-range locks. It is the server's processing of the READ and WRITE operations that introduces the distinction.

Every stateid that is validly passed to READ, WRITE, or SETATTR, with the exception of special stateid values, defines an access mode for the file (i.e., OPEN4_SHARE_ACCESS_READ, OPEN4_SHARE_ACCESS_WRITE, or OPEN4_SHARE_ACCESS_BOTH).

* For stateids associated with opens, this is the mode defined by the original OPEN that caused the allocation of the OPEN stateid and as modified by subsequent OPENs and OPEN_DOWNGRADEs for the same open-owner/file pair.

* For stateids returned by byte-range LOCK operations, the appropriate mode is the access mode for the OPEN stateid associated with the lock set represented by the stateid.

* For delegation stateids, the access mode is based on the type of delegation.

When a READ, WRITE, or SETATTR (that specifies the size attribute) operation 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 that 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 OPEN4_SHARE_ACCESS_WRITE, 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 specified OPEN4_SHARE_DENY_READ or OPEN4_SHARE_DENY_BOTH). Note that a server that does enforce the access mode check on READs need not explicitly check for conflicting share reservations since the existence of OPEN for OPEN4_SHARE_ACCESS_READ guarantees that no conflicting share reservation can exist.

The READ bypass special stateid (all bits of "other" and "seqid" set 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:

* A mandatory byte-range lock is requested with a byte-range that conflicts with the byte-range of the READ or WRITE operation. For the purposes of this paragraph, a conflict occurs when a shared lock is requested and a WRITE operation is being performed, or an exclusive lock is requested and either a READ or a WRITE operation is being performed.

* A share reservation is requested that denies reading and/or writing and the corresponding operation is being performed.

* A delegation is to be granted and the delegation type would prevent the I/O operation, i.e., READ and WRITE conflict with an OPEN_DELEGATE_WRITE delegation and WRITE conflicts with an OPEN_DELEGATE_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, 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.

14.2. 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 byte-range that overlaps -- fully or partially -- either with 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 for reasons related to the recovery of byte-range locking state in the event of server failure. As discussed in Section 13.4.2, 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.

14.3. Upgrading and Downgrading Locks

If a client has a WRITE_LT lock on a byte-range, it can request an atomic downgrade of the lock to a READ_LT lock via the LOCK operation, 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_LT lock on a byte-range, it can request an atomic upgrade of the lock to a WRITE_LT lock via the LOCK operation 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 operation 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.

14.4. Stateid Seqid Values and Byte-Range Locks

When a LOCK or LOCKU operation is performed, the stateid returned has the same "other" value as the argument's stateid, and a "seqid" value that is incremented (relative to the argument's stateid) to reflect the occurrence of the LOCK or LOCKU operation. 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 the 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_LT to being locked for WRITE_LT or the reverse.

When there is no such change, as, for example, when a range already locked for WRITE_LT is locked again for WRITE_LT, the server MAY increment the "seqid" value.

14.5. 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 sent 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_LT or WRITE_LT type 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 sent.

When a byte is locked by the lock-owner in question, the open-owner to which that byte-range 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.

14.6. 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_LT and WRITEW_LT, 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 needless 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 operation, 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 operations. 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 a server indicates, via the flag OPEN4_RESULT_MAY_NOTIFY_LOCK, that CB_NOTIFY_LOCK callbacks might 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 that the server is under no obligation to reserve the lock for that particular client.

14.7. 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 (OPEN4_SHARE_DENY_NONE, OPEN4_SHARE_DENY_READ, OPEN4_SHARE_DENY_WRITE, or OPEN4_SHARE_DENY_BOTH). If the OPEN fails, the client will fail the application's open request.

Pseudo-code definition of the semantics:

```
        if (request.access == 0) {
          return (NFS4ERR_INVAL)
        } else {
          if ((request.access & file_state.deny)) ||
             (request.deny & file_state.access)) {
            return (NFS4ERR_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;
```

14.8. 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. Clients that do not have a deny mode built into their programming interfaces for opening a file should request a deny mode of OPEN4_SHARE_DENY_NONE.

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

The CLOSE operation removes all share reservations held by the open-owner on that file. If byte-range locks are held, the client SHOULD release all locks before sending a CLOSE operation. 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 that the client has the least access. For example, if one client opened a file with OPEN4_SHARE_DENY_BOTH and another client accesses the file via a filehandle obtained through LOOKUP, the second client could only read the file using the special read bypass stateid. The second client could not WRITE the file at all because it would not have a valid stateid from OPEN and the special anonymous stateid would not be allowed access.

14.9. 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" value 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 OPEN4_SHARE_ACCESS_BOTH). 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 that happen while the CLOSE is pending. Note that the client, when sending 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 situations in which there is no change to the access and deny bits for the file.

14.10. 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.

14.11. 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.

* 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.

* To reclaim byte-range locks, a LOCK operation with the reclaim parameter set to true is used.

Reclaims of opens associated with delegations are discussed in Section 15.2.1.

15. 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.

15.1. 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 references to the server to find that no conflicts exist are 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 byte-range locking. Sending LOCK and LOCKU operations as well as the READ and WRITE operations necessary to make data caching consistent with the locking semantics (see Section 15.3.2) 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 byte-range locking by applications.

The NFSv4.1 protocol provides more aggressive caching strategies with the following design goals:

* Compatibility with a large range of server semantics.

* Providing the same caching benefits as previous versions of the NFS protocol when unable to support the more aggressive model.

* Requirements for aggressive caching are organized so that a large portion of the benefit can be obtained even when not all of the requirements can be met.

The appropriate requirements for the server are discussed in later sections in which specific forms of caching are covered (see Section 15.4).

15.2. 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 BIND_CONN_TO_SESSION, and the client is required to maintain it. Because the backchannel may be down, even temporarily, correct protocol operation does not depend on them. Preliminary testing of

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.

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.

* 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.

* Operations done outside the NFSv4.1 protocol, due to, for example, access by other protocols including other minor version of NFSv4, or by local access, also need to result in delegation recall when they make analogous changes to file system data, including the delegated file’s contents, its attributes and the set of names linked to that file. 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.

* 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:

* For GETATTR, see Section 23.7.4.

* For LINK, see Section 23.9.4.

* For OPEN, see Section 23.16.4.

* For READ, see Section 23.22.4.

* For REMOVE, see Section 23.25.4.

* For RENAME, see Section 23.26.4.

* For SETATTR, see Section 23.30.4.

* For WRITE, see Section 23.32.4.

On recall, the client holding the delegation needs to 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 15.4). 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 states for the file allow 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.

15.2.1. Delegation Recovery

There are three situations that delegation recovery needs to deal with:

* client restart

* server restart

* network partition (full or backchannel-only)

In the event the client restarts, establishment of a new clientid associated with the new client instance or failure to renew the lease will result in the revocation of byte-range locks and share reservations. Delegations, however, may be treated somewhat differently. It is also possible for the same sorts of revocation to occur as a result of lease non-renewal.

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

To allow for this type of client recovery, the server MAY provide a special period to allow the clients to recover the delegations obtained before the restart. This special period will often be longer the typical lease expiration period. As a result, requests from other clients that conflict with these delegations would 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. Such a longer interval would increase the window for clients to restart and consult stable storage so that the delegations can be returned after the data is appropriately flushed to the server.

This special period, although analogous to the grace period used after server restart, is distinct from it. For OPEN delegations, such delegations are reclaimed using OPEN with a claim type of CLAIM_DELEGATE_PREV or CLAIM_DELEG_PREV_FH (see Sections 15.5 and 23.16 f or discussion of OPEN delegation and the details of OPEN, respectively). Although these types of OPENs are considered reclaim-type operations they are not, like other sorts of reclaims limited to the grace period. They are intended for use during the special delegation recovery period, and are not directly affected by possible existence of a server grace period.

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 confirm a client ID created by EXCHANGE_ID. Instead, the server 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 and/or CLAIM_DELEG_PREV_FH 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:

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

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

* The use of callbacks should not 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 OPEN_DELEGATE_WRITE delegation, there are issues because data may have been modified by the client whose delegation is revoked and separately by other clients. See Section 15.5.1 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 13.4.3). 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.

15.3. 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 on which those using these facilities depend.

15.3.1. Data Caching and OPENs

In order to avoid invalidating the sharing assumptions on which applications rely, 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 an OPEN delegation (see Section 15.4), two additional rules apply. Note that these rules are obeyed in practice by many NFSv3 clients.

* First, cached data present on a client must be revalidated after doing an OPEN. Revalidating means that the client fetches the change attribute from the server, compares it with the cached change attribute, and if different, declares the cached data (as well as the cached attributes) as invalid. This is to ensure that the data for the OPENed file is still correctly reflected in the client's cache. This validation must be done at least when the client's OPEN operation includes a deny of OPEN4_SHARE_DENY_WRITE or OPEN4_SHARE_DENY_BOTH, thus terminating a period in which other clients may have had the opportunity to open the file with OPEN4_SHARE_ACCESS_WRITE/OPEN4_SHARE_ACCESS_BOTH access. Clients may choose to do the revalidation more often (i.e., at OPENs specifying a deny mode of OPEN4_SHARE_DENY_NONE) to parallel the NFSv3 protocol's practice for the benefit of users assuming this degree of cache revalidation.

  Since the change attribute is updated for data and metadata modifications, some client implementers may be tempted to use the time_modify attribute and not the change attribute to validate cached data, so that metadata changes do not spuriously invalidate clean data. The implementer is cautioned 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. Thus, any cache validation approach by the client MUST include the use of the change attribute.

* Second, modified data must be flushed to the server before closing a file OPENed for OPEN4_SHARE_ACCESS_WRITE. This is complementary to the first rule. If the data is not flushed at CLOSE, the revalidation done after the client OPENs a file is unable to achieve its purpose. The other aspect to flushing the data before close is that the data must be committed to stable storage, at the server, before the CLOSE operation is requested by the client. In the case of a server restart and a CLOSEd file, it may not be possible to retransmit the data to be written to the file, hence, this requirement.

15.3.2. Data Caching and File Locking

For those applications that choose to use byte-range 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 byte-range locking is used in a way that matches in an equivalent way the actual READ and WRITE operations executed. This is as opposed to byte-range 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 ranges and protecting access to the two byte-ranges by byte-range locks on bytes zero and one. A WRITE_LT lock on byte zero of the file would represent the right to perform READ and WRITE operations on the first byte-range. A WRITE_LT lock on byte one of the file would represent the right to perform READ and WRITE operations on the second byte-range. 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 byte-range locking environment are:

* First, when a client obtains a byte-range lock for a particular byte-range, the data cache corresponding to that byte-range (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 byte-range. A client might choose to invalidate all of the 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 byte-range.

* Second, before releasing a WRITE_LT lock for a byte-range, all modified data for that byte-range 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 byte-range outside the unlocked area. This, in turn, may be part of a byte-range 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 that 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 that the client possesses may not be valid.

The data that is written to the server as a prerequisite to the unlocking of a byte-range 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 that use byte-range locking in non-standard ways (e.g., using a byte-range lock as a global semaphore) by flushing to the server more data upon a LOCKU than is covered by the locked range. This may include modified data within files other than the one for which the unlocks are being done. In such cases, the client must not interfere with applications whose READs and WRITEs are being done only within the bounds of byte-range locks that 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 LOCKU operation. However, it would not be valid to write the entire block in which that single written byte was located since it includes an area that is not locked and might be locked by another client. Client implementations can avoid this problem by dividing files with modified data into those for which all modifications are done to areas covered by an appropriate byte-range lock and those for which there are modifications not covered by a byte-range lock. Any writes done for the former class of files must not include areas not locked and thus not modified on the client.

15.3.3. Data Caching and Mandatory File Locking

Client-side data caching needs to respect mandatory byte-range locking when it is in effect. The presence of mandatory byte-range locking for a given file is indicated when the client gets back NFS4ERR_LOCKED from a READ or WRITE operation on a file for which it has an appropriate share reservation. When mandatory locking is in effect for a file, the client must check for an appropriate byte-range lock for data being read or written. If a byte-range lock exists for the range being read or written, the client may satisfy the request using the client's validated cache. If an appropriate byte-range 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 byte-range, the request should be subdivided into multiple pieces with each byte-range (locked or not) treated appropriately.

15.3.4. 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 that 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:

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

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

* If GETATTR directed to the two filehandles does not return the fileid attribute for both of the handles, then it cannot be determined whether the two objects are the same. Therefore, operations that depend on that knowledge (e.g., client-side data caching) cannot be done reliably. Note that if GETATTR does not return the fileid attribute for both filehandles, it will return it for neither of the filehandles, since the fsid for both filehandles is the same.

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

* Otherwise, they are the same object.

15.4. 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, if the server receives 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 an OPEN should be delegated:

* 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.

* The client must have responded properly to previous recalls.

* There must be no current OPEN conflicting with the requested delegation.

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

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

* The existence of any server-specific semantics of OPEN/CLOSE that would make the required handling incompatible with the prescribed handling that the delegated client would apply (see below).

There are two types of OPEN delegations: OPEN_DELEGATE_READ and OPEN_DELEGATE_WRITE. An OPEN_DELEGATE_READ delegation allows a client to handle, on its own, requests to open a file for reading that do not deny OPEN4_SHARE_ACCESS_READ access to others. Multiple OPEN_DELEGATE_READ delegations may be outstanding simultaneously and do not conflict. An OPEN_DELEGATE_WRITE delegation allows the client to handle, on its own, all opens. Only one OPEN_DELEGATE_WRITE delegation may exist for a given file at a given time, and it is inconsistent with any OPEN_DELEGATE_READ delegations.

When a client has either type of 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 an OPEN_DELEGATE_WRITE delegation, it may modify the file data locally since no other client will be accessing the file’s data. The client holding an OPEN_DELEGATE_WRITE delegation may only locally affect file attributes that 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 an OPEN_DELEGATE_READ delegation, opens that cannot be handled locally (opens that are for OPEN4_SHARE_ACCESS_WRITE/OPEN4_SHARE_ACCESS_BOTH or that deny OPEN4_SHARE_ACCESS_READ access) must be sent to the server.

When an OPEN delegation is made, the reply to the OPEN contains an OPEN delegation structure that specifies the following:

* the type of delegation (OPEN_DELEGATE_READ or OPEN_DELEGATE_WRITE).

* space limitation information to control flushing of data on close (OPEN_DELEGATE_WRITE delegation only; see Section 15.4.1)

* an nfsace4 specifying read and write permissions

* a stateid to represent the delegation

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 the OPEN delegation being denied so that the checks can be made by the server itself.

* The access and deny bits for the request and the file as described in Section 14.7.

* The read and write permissions as determined below.

The nfsace4 passed with delegation can be used to avoid frequent ACCESS calls. The permission check should be as follows:

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

* If the nfsace4 indicates that the open may not be done, then an ACCESS request must be sent to the server to obtain the definitive answer.

The server may return an nfsace4 that is more restrictive than the actual ACL of the file. This includes an nfsace4 that specifies denial of all access. Note that some common practices such as mapping the traditional user "root" to the user "nobody" (see Section 11.13) 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.

15.4.1. 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 "OPEN_DELEGATE_READ delegation" provides a guarantee that no OPEN for OPEN4_SHARE_ACCESS_WRITE/OPEN4_SHARE_ACCESS_BOTH, and thus no write, has occurred. Similarly, when closing a file opened for OPEN4_SHARE_ACCESS_WRITE/OPEN4_SHARE_ACCESS_BOTH and if an OPEN_DELEGATE_WRITE 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 a 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 returned 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., the 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 OPEN_DELEGATE_WRITE delegations with very restrictive space limitations. The limitations may be defined in a way that will always force modified data to be flushed to the server on close.

With respect to authentication, flushing modified data to the server after a CLOSE has occurred may be problematic. For example, the user of the application may have logged off the client, and unexpired authentication credentials may not be present. In this case, the client may need to take special care to ensure that local unexpired credentials will in fact be available. 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.

15.4.2. Open Delegation and File Locks

When a client holds an OPEN_DELEGATE_WRITE delegation, lock operations are performed locally. This includes those required for mandatory byte-range 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 an OPEN_DELEGATE_READ delegation, lock operations are not performed locally. All lock operations, including those requesting non-exclusive locks, are sent to the server for resolution.

15.4.3. Handling of CB_GETATTR

The server needs to employ special handling for a GETATTR where the target is a file that has an OPEN_DELEGATE_WRITE delegation in effect. The reason for this is that the client holding the OPEN_DELEGATE_WRITE delegation may have modified the data, and the server needs to reflect this change to the second client that submitted the GETATTR. Therefore, the client holding the OPEN_DELEGATE_WRITE delegation needs to be interrogated. The server will use the CB_GETATTR operation. The only attributes that the server can reliably query via CB_GETATTR are size and change.

Since CB_GETATTR is being used to satisfy another client's GETATTR request, the server only needs to know if the client holding the delegation has a modified version of the file. If the client's copy of the delegated file is not modified (data or size), the server can satisfy the second client's GETATTR request from the attributes stored locally at the server. If the file is modified, the server only needs to know about this modified state. If the server determines that the file is currently modified, it will respond to the second client's GETATTR as if the file had been modified locally at the server.

Since the form of the change attribute is determined by the server and is opaque to the client, the client and server need to agree on a method of communicating the modified state of the file. For the size attribute, the client will report its current view of the file size. For the change attribute, the handling is more involved.

For the client, the following steps will be taken when receiving an OPEN_DELEGATE_WRITE delegation:

* The value of the change attribute will be obtained from the server and cached. Let this value be represented by c.

* The client will create a value greater than c that will be used for communicating that modified data is held at the client. Let this value be represented by d.

* When the client is queried via CB_GETATTR for the change attribute, it checks to see if it holds modified data. If the file is modified, the value d is returned for the change attribute value. If this file is not currently modified, the client returns the value c for the change attribute.

For simplicity of implementation, the client MAY for each CB_GETATTR return the same value d. This is true even if, between successive CB_GETATTR operations, the client again modifies the file's data or

metadata in its cache. The client can return the same value because the only requirement is that the client be able to indicate to the server that the client holds modified data. Therefore, the value of d may always be c + 1.

While the change attribute is opaque to the client in the sense that it has no idea what units of time, if any, the server is counting change with, it is not opaque in that the client has to treat it as an unsigned integer, and the server has to be able to see the results of the client's changes to that integer. Therefore, the server MUST encode the change attribute in network order when sending it to the client. The client MUST decode it from network order to its native order when receiving it, and the client MUST encode it in 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 an OPEN_DELEGATE_WRITE delegation:

* Upon providing an OPEN_DELEGATE_WRITE delegation, the server will cache a copy of the change attribute in the data structure it uses to record the delegation. Let this value be represented by sc.

* When a second client sends a GETATTR operation on the same file to the server, the server obtains the change attribute from the first client. Let this value be cc.

* If the value cc is equal to sc, the file is not modified and the server returns the current values for change, time_metadata, and time_modify (for example) to the second client.

* If the value cc is NOT equal to sc, the file is currently modified at the first client and most likely will be modified at the server at a future time. The server then uses its current time to construct attribute values for time_metadata and time_modify. A new value of sc, which we will call nsc, is computed by the server, such that nsc >= 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.

15.4.4. Recall of Open Delegation

The following events necessitate recall of an OPEN delegation:

* potentially conflicting OPEN request (or a READ or WRITE operation done with a special stateid)
* SETATTR sent by another client
* REMOVE request for the file
* RENAME request for the file as either the 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:

* If the file associated with the delegation is no longer open and no previous CLOSE operation has been sent to the server, a CLOSE operation must be sent to the server.
* If a file has other open references at the client, then OPEN operations must be sent to the server. The appropriate stateids will be provided by the server for subsequent use by the client since the delegation stateid will 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 Section 23.16, which describes the OPEN operation, for details.)
* If there are granted byte-range locks, the corresponding LOCK operations need to be performed. This applies to the OPEN_DELEGATE_WRITE delegation case only.

* For an OPEN_DELEGATE_WRITE delegation, if at the time of recall the file is not open for OPEN4_SHARE_ACCESS_WRITE/ OPEN4_SHARE_ACCESS_BOTH, all modified data for the file must be flushed to the server. If the delegation had not existed, the client would have done this data flush before the CLOSE operation.

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

* With the OPEN_DELEGATE_WRITE delegation in place, it is possible that the file was truncated during the duration of the delegation. For example, the truncation could have occurred as a result of an OPEN 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 OPEN_DELEGATE_WRITE delegation, byte-range locking imposes some additional requirements. To precisely maintain the associated invariant, it is required to flush any modified data in any byte-range for which a WRITE_LT lock was released while the OPEN_DELEGATE_WRITE delegation was in effect. However, because the OPEN_DELEGATE_WRITE delegation implies no other locking by other clients, a simpler implementation is to flush all modified data for the file (as described just above) if any WRITE_LT lock has been released while the OPEN_DELEGATE_WRITE delegation was in effect.

An implementation need not wait until delegation recall (or the decision 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. An exception is when the client has no more internal opens of the file. In this case, sending a CLOSE is useful because it reduces resource utilization on the client and server. 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.

15.4.5. 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 an OPEN_DELEGATE_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 revoked.

15.4.6. 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 byte-range 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 OPEN4_SHARE_ACCESS_BOTH with a deny of OPEN4_SHARE_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 the two OPENs are consistent with each other. On the other hand, if an OPEN denying write access 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 15.5.1 for additional details.

15.4.7. 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 that 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, because existing byte-range locks are subordinate to an open, those byte-range locks also 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, this occurs for existing byte-range locks subordinate to an open.

15.5. Data Caching and Revocation

When locks and delegations are revoked, the assumptions upon which successful caching depends 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 that 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 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 byte-range lock after the revocation of the byte-range 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 here is that an invariant on which an application depends 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.

15.5.1. Revocation Recovery for Write Open Delegation

Revocation recovery for an OPEN_DELEGATE_WRITE delegation poses the special issue of modified data in the client cache while the file is not open. In this situation, any client that 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 namespace to ease recovery. Note that when the client can determine that the file has not been modified by any other client, or when the client has a complete cached copy of the

file in question, such a saved copy of the client's view of the file may be of particular value for recovery. In another case, recovery using a copy of the file based partially on the client's cached data and partially on the server's copy as modified by other clients will be anything but straightforward, so clients may avoid saving file contents in these situations or specially mark the results 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.

15.6. 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 (as well as the directory's contents) are to be cached on the same basis as any other pathnames.

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 should not be 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 another 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 incoherencies mentioned above can be reasonably managed. These rules are derived from the practice of previous NFS protocols.

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

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

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

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

A client may validate its cached version of attributes for a file by fetching 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 [write_atime]. 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 the client either would eventually have to write the access time to the server with bad performance effects or 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.

15.7. Data and Metadata Caching and Memory Mapped Files

Some operating environments include the capability for an application to map a file's content into the application's address space. Each time the application accesses a memory location that corresponds to a block that has not been loaded into the address space, a page fault occurs and the file is read (or if the block does not exist in the file, the block is allocated and then instantiated in the application's address space).

As long as each memory-mapped access to the file requires a page fault, the relevant attributes of the file that are used to detect access and modification (time_access, time_metadata, time_modify, and change) will be updated. However, in many operating environments, when page faults are not required, these attributes will not be updated on reads or updates to the file via memory access (regardless of whether the file is local or is accessed remotely). A client or server MAY fail to update attributes of a file that is being accessed via memory-mapped I/O. This has several implications:

* 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 or not its cache is stale. A server that knows that the file is memory-mapped could always pessimistically return updated values for change so as to force the application to always get the most up-to-date data and metadata for the file. However, due to the negative performance implications of this, such behavior is OPTIONAL.

* If the memory-mapped file is not being modified on the server, and instead is just being read by an application via the memory-mapped interface, the client will not see an updated time_access attribute. However, in many operating environments, neither will any process running on the server. Thus, NFS clients are at no disadvantage with respect to local processes.

* If there is another client that is memory mapping the file, and if that client is holding an OPEN_DELEGATE_WRITE delegation, the same set of issues as discussed in the previous two bullet points apply. However, it should be noted that it is very unlikely that such a delegation will be held since it is normally required that the file be open for read to be mapped into memory. Only if the file were not open and accessed using a special stateid could the delegation be retained while the file in question is mapped into another client's memory. For this reason, such use is highly undesirable.

  In this situation, when a server does a CB_GETATTR to a file that the client has modified in its cache, the reply from CB_GETATTR would not necessarily be accurate, assuming the delegation is not recalled at this point. 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, it might not be possible to live up to this weak obligation. A client MAY return stale information in CB_GETATTR whenever the file is memory-mapped, if another client is accessing the file without opening it.

15.8. Name and Directory Caching without Directory Delegations

The NFSv4.1 directory delegation facility (described in Section 15.9 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. These topics are discussed in the next two subsections.

15.8.1. 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 for 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 in 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 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. If two such operations complete in a different order than they were actually performed, that 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 can be 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.

15.8.2. 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:

* Cached READDIR information for a directory that 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 READDIR that contributes to the cache.

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

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

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

15.9. Directory Delegations and Notifications

15.9.1. Motivation for Directory Delegations

Directory caching for the NFSv4.1 protocol when directory delegations are not available, is similar to file and directory caching in previous versions. Clients typically cache directory information for a duration determined by the client. At the end of that predefined period, 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. As a result, 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 making an RPC

call. By caching name and attributes information about most recently looked up entries in a Directory Name Lookup Cache (DNLC), clients are able to avoid sending LOOKUP/GETATTR calls to the server every time such 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.

Since, 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, it is sensible to define and implement such facilities in NFSv4.1.

15.9.2. Directory Caching Features

Delegation of directory contents is an OPTIONAL feature of NFSv4.1. Possession of a delegation can be taken advantage of in a number of ways:

* It can be used to provide a recallable assurance that the directory contents have not changed, allowing LOOKUP results (whether successful or not) and READDIR results to be cached, in order to enable these operations to be performed locally.

  This mode of operation in which directory contents are fixed is often referred to as the "pure recall" model since any change in the directory contents results in the delegation being recalled. This mode of operation is most effectively used on large directories which are infrequently changed.

* The client can request, as part of requesting a delegation, that notifications be provided to update the clients view of the directory contents to match that of the server. See Section 15.9.7 for details. This mode of operation allows directory delegations to be effectively used in handling large directories that experience a significant stream of updates.

* Independently of the mode of operation selected, notifications to inform the client of attribute changes can be requested. See Section 15.9.7 for details.

15.9.3. Directory Delegation Mechanics

The GET_DIR_DELEGATION (Section 23.39) operation is used by clients to request directory delegation. The delegation is read-only and the client is not provided any means to make changes to the directory other than by performing NFSv4.1 operations that modify the directory.

As part of obtaining a delegation, the client specifies, using the bit numbers within the notify_type4 enum that appears below, its choices regarding notification of events related to the reporting of events affecting the delegation. Some, although not all, directly specify the use of particular notification types, to be used to inform the client of events that could otherwise result in recall of the delegation.

It is important to note that this enum is subject to extension and has been extended relative to the set of bits defined in [RFC8881]. The distinction between bits that were defined earlier and those added later is important to enable interoperation between clients and servers when one might have been written based on the earlier specification. Although no implementations based on the earlier specification are known, the possibility of their existence cannot be excluded.

```
/*
 * Directory notification types and associated flags
 */
enum notify_type4 {
        /*
         * Present in RFCs 5661, 8881
         */
        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,
        /*
         * Added in NFSv4.1 bis document
         */
        NOTIFY4_GFLAG_EXTEND = 6,
        NOTIFY4_AUFLAG_VALID = 7,
        NOTIFY4_AUFLAG_USER = 8,
        NOTIFY4_AUFLAG_GROUP = 9,
        NOTIFY4_AUFLAG_OTHER = 10,
        NOTIFY4_CHANGE_AUTH = 11,
        NOTIFY4_CFLAG_ORDER = 12,
        NOTIFY4_AUFLAG_GANOW = 13,
        NOTIFY4_AUFLAG_GALATER = 14,
        NOTIFY4_CHANGE_GA = 15,
        NOTIFY4_CHANGE_AMASK = 16

};
```

Of the newer bits, only NOTIFY4_GFLAG_EXTEND, NOTIFY4_CHANGE_AUTH, NOTIFY4_CFLAG_ORDER, NOTIFY4_CHANGE_GA, and NOTIFY4_CHANGE_AMASK can appear when requesting a delegation. When any of these are set the server, it is possible that the server is unaware of their existence and will ignore them. If the client sets NOTIFY4_GFLAG_EXTEND in the request and it is returned set in the response, the client and server can interact assuming that each is aware of the newer bits. For more details about dealing with possibility of implementations of multiple versions of this feature interacting, see Section 15.9.6.

Of these bits the following subsets should be noted:

* The bits NOTIFY4_CHANGE_CHILD_ATTRS , NOTIFY4_CHANGE_DIR_ATTRS, NOTIFY4_REMOVE_ENTRY, NOTIFY4_ADD_ENTRY, NOTIFY4_RENAME_ENTRY, and NOTIFY4_CHANGE_COOKIE_VERIFIER all have associated notification messages and were defined in [RFC8881]

When these bits are set when requesting a delegation, the server is being notified of the client's desire to have the corresponding notification sent rather than recalling the delegation. when the server sets these bits in the response, it is indicating its agreement to provide these notifications.

* The bits NOTIFY4_CHANGE_AUTH, NOTIFY4_CHANGE_GA, and NOTIFY4_CHANGE_AMASK also have associated notification messages.

  These notifications can be requested as in the case above. However, it is possible that the server is unaware of their existence.

* The bit NOTIFY4_GFLAG_EXTEND denotes a flags to be exchanged as part of requesting a delegation.

* The bits NOTIFY4_CFLAG_ORDER denotes a flag that can be set as part of requesting a delegation but has no role in requests.

* The bits NOTIFY4_AUFLAG_VALID, NOTIFY4_AUFLAG_USER, NOTIFY4_AUFLAG_GROUP, NOTIFY4_AUFLAG_OTHER, NOTIFY4_AUFLAG_GANOW and NOTIFY4_AUFLAG_GALATER, can be set in the response but have no role in requests.

Of these bits only NOTIFY4_GFLAG_EXTEND is of general applicability and applies to multiple functions discussed in the subsections below. The other are discussed in more detail as grouped below:

* The bits NOTIFY4_REMOVE_ENTRY, NOTIFY4_ADD_ENTRY, NOTIFY4_RENAME_ENTRY, NOTIFY4_CHANGE_COOKIE_VERIFIER, and NOTIFY4_CFLAG_ORDER concern the maintenance of cached directory contents and are discussed in Section 15.9.7

* The bits NOTIFY4_CHANGE_CHILD_ATTRS, NOTIFY4_CHANGE_DIR_ATTRS, and NOTIFY4_CHANGE_AMASK concern the maintenance of cached file object attributes and are discussed in Section 15.9.8

* The bits NOTIFY4_AUFLAG_VALID, NOTIFY4_AUFLAG_USER, NOTIFY4_AUFLAG_GROUP, NOTIFY4_AUFLAG_OTHER, NOTIFY4_CHANGE_AUTH, NOTIFY4_AUFLAG_GANOW, NOTIFY4_AUFLAG_GALATER, NOTIFY4_CHANGE_GA concern the management of authorizations for the cached use of file contents and file attributes are discussed in Section 15.9.9.

The holder is assured of certain thing not being changed while the directory is held, as described below.

* That the set of entries within the directory not be changed without sending a requested notification to the client, informing the client of the change.

* That the order of directory entries or the cookie values associated with specific directory entry with the client being informed (via a NOTIFY4_CHANGE_COOKIE_VERIFIER notification) of the possibility of change.

Delegations can be recalled by the server at any time and are always recalled before a directory is removed.

15.9.4. Directory Delegation Authorization Requirements

When cached data is used locally in place of LOOKUP, GETATTR, or, READDIR operations, the authorization constraints that would normally by imposed by the server have to be applied by the client. The discussion is complicated by the fact that, while facilities have been designed to accomplish that are described in this document, the treatment in earlier specifications did not provide facilities to help the client do this correctly and had little to say on the issue. As a result, clients were faced with the choice of ignoring difficult authorization issues or burdening the implementation with authorization checking that would undercut the performance benefits of the feature.

As a result, we are faced with the issue of how to accommodate implementations that are now known to have troubling problems that were not recognized when the feature was first described in a Proposed Standard. Normally, one tries to accommodate such situations by recommending against approaches now known to be flawed while considering, as a valid reason to bypass the recommendation, the reliance of the implementer on an approved Proposed Standard at the time. In this case we have a different approach, because of the following distinctive factors:

* Unlike the case of implementers being told that use of AUTH_SYS in the clear, is an "OPTIONAL means of authentication" with the implication that such use does not result in potentially unacceptable security vulnerabilities, here there is no direct suggestion that neglecting these difficulties is acceptable. Instead, while lack of attention to security issues might have led people astray, they were not specifically asked to adopt a flawed approach to security but chose to adopt one on their own. As a result, while we will make certain allowances to accommodate such early implementations, there is no known paradigm that could be cited as valid but discouraged.

* The set of implementations involved is likely to be quite small and might be empty or only consist of experimental implementations not widely distributed.

The approach we take here is the same one we take to servers that do not support the extensions described in Section 15.9.9 and to clients that interact with such servers. It has the following elements:

* Clients are free in deciding whether to use directory delegations to take account of the problems with earlier approaches in deciding whether to use this feature.

* Neglecting the possibility of authorization failure on GETATTR when directory entry attributes are cached is not to be considered disabling. This includes situation in which the server supports the ACE mask bit ACE4_READ_ATTRIBUTES.

* Use of directory attributes for the clients to do its own authorization needs to be discouraged for a number of reasons.

  Prime among these is the possibility that the acl attribute might be set for the directory, making it impossible for the client to its own authorization checking.

* Even in the case of a file system on which none of attributes acl, dacl , and sacl is supported, the use of client-side authorization is not justifiable, since the attributes can change subsequently and the potential delay for the update of directory attributes has no upper bounds.

* When clients use ACCESS to do authorization checks, as they should, allowance needs to be made for them to cache positive results, since without that ability, you might as well fetch the data over the wire anyway.

In this discussion, we will consider how various pair of implementations have dealt and will deal with this issue using combination of server guarantees and the use over-the-wire ACCESS checks and the potential caching of these results. Some things to note about potential implementations based on earlier specifications:

* It is unlikely that the unfortunate effects of authorization failure were considered at all. Since the issue was introduced by the inclusion of very-rarely-implemented ACE mask bit ACE4_READ_ATTRIBTES, it is likely that this issue was imply ignored.

* It is reasonable to suppose that clients were expected to request authorization checks using ACCESS and that clients were prepared to cache these determinations.

* It is likely that server-based guarantees were never provided.

In the current description, the protocol has been extended to address these gaps. As a result when both client and server based on this description, the following apply.

* The server provide a guarantee that GETATTRs can be done locally without concern for the possibility of denial or the need to perform action based on AUDIT or alarm ACEs, and that the client will be notified when the server becomes aware of circumstances making that guarantee inappropriate.

  That guarantee is trivial to provide for the large set of servers that do not support the ACE mask bit ACE4_READ_ATTRIBUTES.

  For servers that do support that mask bit, the server could provide the guarantee by a scan of the directory for files with troublesome ACLs. However because of the performance effects of requiring that scan to grant a delegation, the sever is allowed to delay that guarantee until after the delegation is granted.

* Authorization for LOOKUP and READDIR is fundamentally the responsibility of the client to ascertain using ACCESS calls.

  To reduce the burden of those calls, the server is expected to provide information about various classes of users for which such authorization check are unnecessary.

  To further reduce the burden of those calls, the result of authorization checks can be cached until the server notifies the need to clear caching for those classes due to a change in authorization-related attributes.

  The server provides notifications when there are changes in the groups of users for which authorization checks are needed.

If we expand the discussion to apply to all implementations including potential client and server implementations written based on earlier specifications, the following constraints apply:

* Servers SHOULD provide the services described above.

In this context, the only valid reason to bypass the recommendation is the implementer’s reliance on an earlier specification in which such authorization-checking assistance was not provided for. This includes cases in which the planning for the implementation was based on an earlier specification.

* Clients SHOULD use these facilities when they are available.

  In this context, the only valid reason to bypass the recommendation is the implementer’s reliance on an earlier specification in which such authorization-checking assistance was not provided for.

* When clients do not use these facilities, they MAY avoid use of directory delegations. However, if they choose to use this cached data they MUST do their own authorization checks, using ACCESS.

  Clients are free to cache the results of such authorization checks but MUST limit the lifetime of such cached results to a period of a few seconds.

15.9.5. Directory Delegation Authorization Support

In providing support for authorization of local operations effecting, using cached data, the equivalents of LOOKUP and READDIR operations, the following issues must be dealt with:

* Because of the complexity and current vagueness, the client could not realistically determine authorization by looking at the directory’s attributes, even if it were not prohibited from examining the ACL, as it is now.

* The possibility of change in authorization-related attributes would make repeated ACCESS call necessary, unless facilities are provided to avoid these when possible.

Note that the same issues apply to authorization of GETATTR-equivalent local operations, but that, in that case, there are the following additional issues to deal with:

* The only potential reason to not grant such access derives from the possible use of the ACE mask bit ACE4_READ_ATTRIBUTES.

  Only where that bit is supported in ACLs and used to either deny access or require audit or alarm on this operation is there any possibility of not letting this operation be done unconditionally.

* Since permissions would need to be checked for each individual object rather than for the directory as a whole, it is harder to avoid unnecessary ACCESS calls in situations where the possibility of denial exists.

  The possible existence of multiply-linked file adds further difficult since it is possible that an ACL could be changed for such an object in case in which the affected directories might not be known.

* There are very few ACL implementations supporting use of the ACE mask bit ACE4_READ_ATTRIBUTES and no known uses of it.

  As a result we have to be prepared to efficiently deal with the simple case where sophisticated support is unnecessary, as well as providing reasonable support to deal with the possibility of it becoming more widely implemented.

To provide better support for authorization of LOOKUP/READDIR, we do the following:

* When the delegation is created, the server returns information about sets of users for which explicit authorization checks can be avoided.

  The flags NOTIFY4_AUFLAG_OWNER, NOTIFY4_AUFLAG_GROUP, and NOTIFY4_AUFLAG_OTHERS indicate the ability to avoid authorization checks for LOOKUP and READDIR by the owner of the file, other members of the owning group, and others, respectively.

* When there is a change in one or more of the directory's authorization-related attributes, the client is notified of the new authorization handling scheme using the NOTIFY4_CHANGE_AUTH notification.

  The notification provides changes that apply separately to the owning user, other users in the owning group, and others. For each such group, there are separate bits controlling the need for explicit ACCESS checks for LOOKUP and for READDIR, and directing the client whether to flush cached results for previous ACCESS checks.

To provide adequate support for authorization of local GETATTR we define a set of GETATTR authorization states in enum below and late describe how the client is transitioned between these states in response to attribute changes that happen on objects within the directory.

```
/*
 * GETATTR authorization states
 */
enum authga_state4 {
        AUTHGA4_UNKNOWN = 0,
        AUTHGA4_ALLOK   = 1,
        AUTHGA4_SOMEOK  = 2
};
```

When a directory delegation is granted, the client uses the flags returned to establish an initial authorization state as follows:

* If the flag NOTIFY4_AUFLAG_GANOW is set, the client is being told that GETATTRs can now be done without explicit ACCESS checks, so the delegation can be put in authorization state AUTHGA4_ALLOK from its inception.

  The server can do this if ACE4_READ_ATTRIBUTES is not supported and also if it has scanned the directory to make sure that no current ACEs use that mask and that there are no multiple-linked files that make it possible that such ACEs will be set without the directory delegation holder being notified.

* Otherwise, if the flag NOTIFY4_AUFLAG_GALATER is set, the client is being told that GETATTRs now require explicit ACCESS checks, but that the situation is expected to change and it will notified of that using a NOTIFY4_CHANGE_GA notification. In this case, the delegation is be put in authorization state AUTHGA4_UNKNOWN at its inception.

  The server can do this to avoid waiting for a scan of the directory looking for troublesome ACLs or multiply-linked linked file that might get troublesome ACLs using one of the other links. The scan can go on with the client being notified of the new status later.

* If neither of these bits is set, then the server is indicating the absence of support for avoiding use of ACCESS to check for GETATTR authorization. In this case, the delegation is be put in authorization state AUTHGA4_UNKNOWN with no expectation of change, requiring explicit authorization checks as attributes are accessed.

Once this initial state is set, it can be modified as described below, as the server’s knowledge of the set of files that require explicit authorization checks changes in response to file system changes.

* When a file within the directory is assigned an ACL that can interfere with the client providing cached attributes without ACCESS checks, the client can be notified of that change of status using a NOTIFY4_CHANGE_GA notification.

* Similarly, when a file within the directory is becomes reachable via an additional link, making it possible that it will subsequently be assigned an ACL without being aware of the directory delegation, there is also a need for the client to be notified. Since such an ACL could interfere with the client providing cached attributes without ACCESS checks, the client is also notified of that change of status using a NOTIFY4_CHANGE_GA notification.

15.9.6. Directory Delegation Feature Version Management

As part work undertaken to respecify NFSv4 minor version one to reflect implementation experience since the publication of [RFC5661], it was necessary to make certain protocol extensions in order to correct problems that had resulted in a lack of implementation of the Directory Delegation feature in the years since its initial introduction.

These extensions took the form of additions to the enum notify_type4 as described in [RFC8178]. These new values,

* Provide new notifications including a set focused on providing authorization support to allow operations to be performed locally without impacting needed authorization semantics.

  Provided a new notification to allow server to deal with excessive backchannel traffic for attribute updates without delegation recall.

* Created flags to be sent by the client as part of delegation request and by the server as part of delegation creation.

  These flags allowed necessary version control, improved authorization handling and a more flexible approach to the provision of position information in content update notifications.

These new notifications and flags are described, together with the older ones, in Sections 15.9.7 through 15.9.9

The flag NOTIFY4_GFLAG_EXTEND has a special role in the management of versions, in order to support interoperation of implementations written to conform to [RFC8881] and to the those written to conform to the updated definition:

* When NOTIFY4_GFLAG_EXTEND is set in a request, the client is indicating that it is aware of the additional flags and notifications.

* When NOTIFY4_GFLAG_EXTEND is set in a response, the server is indicating that it is aware of the additional flags and notifications. and that the delegation is to be handled in accord with the updated specification of the feature.

15.9.7. Directory Content Notifications

The notification types NOTIFY4_ADD_ENTRY, NOTIFY4_REMOVE_ENTRY, NOTIFY4_RENAME_ENTRY, and NOTIFY4_CHANGE_COOKIE_VERIFIER, discussed below, are provided to inform the delegation holder of changes in the contents of directories. Since the holder can use these notifications to keep his view of the directory contents in sync with that of the server, delegations are not recalled when the client has requested an appropriate content notification. For details regarding the specifics of the relevant notification messages, see the appropriate subsection of Section 25.4.

* NOTIFY4_ADD_ENTRY is used to indicate the creation of a new directory entry, as a result of an OPEN creating a new file, a CREATE operation, a LINK operation, or a cross-directory RENAME operation.

  It is described in Section 25.4.4

* NOTIFY4_REMOVE_ENTRY is used indicat the deletion of an existing directory entry, as a result of a REMOVE operation or a cross-directory RENAME operation.

  It is described in Section 25.4.5

* NOTIFY4_RENAME ENTRY isused to indicae the renaming of an existing directory entry, as a result of a within-directory RENAME operation.

  It is described in Section 25.4.6

* NOTIFY4_CHANGE_COOKIE_VERIFIER is used to notify the client of changes other than those involved changes in the set of directory entries to be cached.

These include, in addition to cookie verifier changes, any changes in cookies for cached entries, even if the verifier was not changed, and changes in directory entry order if the client has indicated its need to maintain its cache in the same order as the server's directory entries.

It is described in Section 25.4.8

In addition, the flag NOTIFY4_CFLAG_ORDER, although it has no associated notification, can be specified together with the bitmask used to specify notifications. When set, it indicates that the client intends to maintain its version of the directory contents in the same order used by the server, This affects the form of position information in content notifications(see below) and whether changes in directory entry order result in NOTIFY4_CHANGE_COOKIE_VERIFIER messages.

The implementation sections for a number of operations describe situations in which notification or delegation recall would be required under some common circumstances. When these events result in delegation recall, a set of caveats similar to those listed in Section 15.2 apply. Note that in these cases, the operation does not wait for the delegation to be returned or revoked, as it does in other cases of delegation recall.

* For CREATE, see Section 23.4.4.

* For LINK, see Section 23.9.4.

* For OPEN, see Section 23.16.4.

* For REMOVE, see Section 23.25.4.

* For RENAME, see Section 23.26.4.

* For SETATTR, see Section 23.30.4.

In the NOTIFY4_ADD_ENTRY, NOTIFY4_REMOVE_ENTRY and NOTIFY4_RENAME_ENTRY notifications, there is position information. This information, which would indicate where in the directory the entry is being added/removed might be sent to the client in a number of ways.

(A): Full position information, as described below, can be implemented on all servers and assumes the client is interested in mimicking the server's entry order and directory cookies.

If the file is added such that there is at least one entry before it, the server will 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.

In either case, the server will set the nad_last_entry flag to TRUE iff this entry is added to the end of the directory.

The client needs to be able to accept notifications with position information. If the information is not needed, it can be ignored.

(B): Position information that takes advantage of the fact that the server that always returns monotonically increasing values for directory offset cookies. Since the new cookie defines the new entry's position, the position information can be sent as described below:

A nad_new_entry_cookie of length 1 with the new cookie in it provides the necessary position information while a nad_prev_entry of length 1 with an invalid notify_entry4 indicated by the ne_file component of length 0 is used. The nad_last_entry value conveys and should be ignored.

When the server send a notification with position information in this format, it telling the client that cookies are monotonically increasing as one proceeds through the directory and this is expected to remain the case. If that ceases to be true, the delegation must recalled.

(C): No position information, useful for clients not interested in the server directory entry ordering is provided as follows:

A nad_new_entry_cookie of length 0. A nad_prev_entry of length 1 with an invalid notify_entry4 indicated by the ne_file component of length 0. The nad_last_entry can be any value and should be ignored. For this option, if nad_old_entry is of length 1, the nrm_old_entry_cookie field in it will always be set to 0.

This form of position information us used in the case in which the client has indicated no interest in keeping its entry order in sync with that of the server.

The form in which the position order is presented in content notifications depends on determining the subset of acceptable formats, as described below, with the server then selecting an appropriate from that set.

Unless notification extensions are known to both client and server (See Section 15.9.6), form (A) is the only acceptable format.

* Full position information (Form (A)) is always an acceptable format.

* Cookie-based position information (Form (B)) is only acceptable if the server's directory cookies are monotonically increasing with directory entry position.

* Absent position information (Form (C)) is only acceptable if the client is not concerned with entry order (i.e NOTIFY4_CFLAG_ORDER is not set.

15.9.8. Directory Attribute Notifications

The following types of notification are used to inform the client of attribute changes:

* NOTIFY4_CHANGE_CHILD_ATTRS is used to provide updated attributes to continue to enable the client to continue to validly cache attributes and respond locally to the need to provide attribute values.

  These notification are subject to delay and batching so as to provide reasonably up-to-date attribute caches without excessive network traffic.

* NOTIFY4_CHANGE_DIR_ATTRS is used to provide updated attributes for the directory itself in order to continue to enable the client to validly cache these attributes and respond locally to the need to provide attribute values in those situations in which an up-to date value is not needed.

  This information can be useful to provide local READDIIR response for APIs which expect it to be present (e.g., as a directory entry for ".").

While it might be supposed that notifications of changes in authorization-related attributes could be used in the authorization of fetches of cached directory contents in performing LOOKUPs and READDIRs locally, this not a viable approach for reasons explained in Section 15.9.4. The better approach, where available, is to use the facilities presented in Section 15.9.9.

* NOTIFY4_CHANGE_AMASK is used to provide updated sets of masks for the attribute updates being provided.

  These notifications, while asynchronous, are not subject to delay or batching.

  This form of notification can be used by the server to reduce or eliminate child attribute notifications, without delegation recall.

For details regarding the specifics of the relevant notification messages, see the appropriate subsection of Section 25.4.

* NOTIFY4_CHANGE_CHILD_ATTRS and NOTIFY4_CHANGE_DIR_ATTRS are described in Section 25.4.7

* NOTIFY4_CHANGE_AMASK is described in Section 25.4.9.

Two of these forms of notification are subject to batching and delays to avoid excessive traffic. While the caller specifies delay parameters when requesting a delegation, attributes provide lower limits for acceptable delays. See Section 11.15 for a description of these attributes.

When excessive traffic is caused by frequent updates for specific attributes, the server has the option of reducing or eliminating the set of attributes using the NOTIFY4_CHANGE_AMASK notification. It also has the option of recalling a delegation in such cases.

15.9.9. Directory Delegation Authorization-related Information

The following types of notification are used to inform the client of the need for changes in the authorization of LOOKUP, READDIR, and GETATTR operations satisfied locally: For details regarding the specifics of the relevant notification messages, see the appropriate subsection of Section 25.4.

* NOTIFY4_CHANGE_AUTH is used to indicate changes in the sets of users for which authorization for local equivalents of LOOKUP and READDIR operations can be done without an explicit ACCESS call.

In addition, it sometimes signals that cached records of previous ACCESS calls need to be flushed.

This notification is described in Section 25.4.10.

* NOTIFY4_CHANGE_GA is used to indicate changes in the required handling of authorization for the local equivalents of GETATTR operations.

  This notification is described in Section 25.4.11.

In addition to the bits used to request notifications, the bits listed below have an important role in managing attribute notifications:

* NOTIFY4_AUFLAG_VALID indicates, in the response to a request to provide a directory delegation, an indication of whether the three bits NOTIFY4_AUFLAG_OWNER, NOTIFY4_AUFLAG_GROUP, and NOTIFY4_AUFLAG_OTHERS have been provided.

  When this bit is not set, these three bits are ignored and the client needs to do its own explicit ACCESS checks until advised otherwise.

* NOTIFY4_AUFLAG_OWNER indicates, when set, that the owner of the directory can do the equivalents of LOOKUP and READDIR without explicit ACCESS checks.

* NOTIFY4_AUFLAG_GROUP indicates, when set, that users that are members of the owning group of the directory who are not the owner of the directory can do the equivalents of LOOKUP and READDIR without explicit ACCESS checks.

* NOTIFY4_AUFLAG_OTHER neither the owner of the directory nor a member of the owning group of the directory can do the equivalents of LOOKUP and READDIR without explicit ACCESS checks.

* NOTIFY4_AUFLAG_GANOW indicates, when set, that the equivalent of GETATTR can be done locally, for all the objects within the directory, without explicit ACCESS checks.

  This state of affairs is subject to change when necessary. Such changes are communicated using the NOTIFY4_CHANGE_GA notification.

* NOTIFY4_AUFLAG_GALATER indicates, when set, that the equivalent of GETATTR can only be done locally, for all the objects within the directory, using explicit ACCESS checks.

This state of affairs is expected to change when later, when the server completes a scan of the directory for files whose ACLs might contain ACEs preventing such local use or multiple links allowing ACL changes where the existence of the delegation might not be noticed by the serve. Such changes are communicated using the NOTIFY4_CHANGE_GA notification.

15.9.10. Directory Delegation Recall

When necessary the server will recall the directory delegation by sending a callback to the client. It uses the same callback procedure as used for recalling file delegations. The server will recall the delegation in the following situations:

* If there is a need to send a content update notification or an authorization update and it is not possible to send that type of notification.

  The server will wait for the delegation to be returned or revoked if the notification was one that needed to be sent synchronously.

* If a client removes a directory for which a delegation has been granted.

If the server determines the existence of a delegation for a directory is causing too many notifications to be sent out, it may decide to not hand out delegations for that directory and/or recall those already granted. In the case of attribute update notifications, it also has the option of reducing update frequency or limiting set of attributes about which the client is to be notified.

15.9.11. 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 OPEN_DELEGATE_READ delegations and notifications. As a result, no provision is made for reclaiming directory delegations in the event of client or server restart. The client needs to establish a directory delegation in the same fashion as was done initially.

16. Multi-Server Namespace

NFSv4.1 supports attributes that allow a namespace to extend beyond the boundaries of a single server. It is desirable that clients and servers support construction of such multi-server namespaces. Use of such multi-server namespaces is OPTIONAL; however, and for many purposes, single-server namespaces are perfectly acceptable. The use of multi-server namespaces can provide many advantages by separating a file system's logical position in a namespace from the (possibly changing) logistical and administrative considerations that cause a particular file system to be located on a particular server via a single network access path that has to be known in advance or determined using DNS.

16.1. Terminology

In this section as a whole (i.e., within all of Section 16), the phrase "client ID" always refers to the 64-bit shorthand identifier assigned by the server (a clientid4) and never to the structure that the client uses to identify itself to the server (called an nfs_client_id4 or client_owner in NFSv4.0 and NFSv4.1, respectively). The opaque identifier within those structures is referred to as a "client id string".

16.1.1. Terminology Related to Trunking

It is particularly important to clarify the distinction between trunking detection and trunking discovery. The definitions we present are applicable to all minor versions of NFSv4, but we will focus on how these terms apply to NFS version 4.1.

* Trunking detection refers to ways of deciding whether two specific network addresses are connected to the same NFSv4 server. The means available to make this determination depends on the protocol version, and, in some cases, on the client implementation.

  In the case of NFS version 4.1 and later minor versions, the means of trunking detection are as described in this document and are available to every client. Two network addresses connected to the same server can always be used together to access a particular server but cannot necessarily be used together to access a single session. See below for definitions of the terms "server-trunkable" and "session-trunkable".

* Trunking discovery is a process by which a client using one network address can obtain other addresses that are connected to the same server. Typically, it builds on a trunking detection facility by providing one or more methods by which candidate addresses are made available to the client, who can then use trunking detection to appropriately filter them.

  Despite the support for trunking detection, there was no description of trunking discovery provided in [RFC5661] or [RFC8881], making it necessary to provide those means in this document.

The combination of a server network address and a particular connection type to be used by a connection is referred to as a "server endpoint". Although using different connection types may result in different ports being used, the use of different ports by multiple connections to the same network address in such cases is not the essence of the distinction between the two endpoints used. This is in contrast to the case of port-specific endpoints, in which the explicit specification of port numbers within network addresses is used to allow a single server node to support multiple NFS servers.

Two network addresses connected to the same server are said to be server-trunkable. Two such addresses support the use of client ID trunking, as described in Section 7.5.

Two network addresses connected to the same server such that those addresses can be used to support a single common session are referred to as session-trunkable. Note that two addresses may be server-trunkable without being session-trunkable, and that, when two connections of different connection types are made to the same network address and are based on a single file system location entry, they are always session-trunkable, independent of the connection type, as specified by Section 7.5, since their derivation from the same file system location entry, together with the identity of their network addresses, assures that both connections are to the same server and will return server-owner information, allowing session trunking to be used.

16.1.2. Terminology Related to File System Location

Regarding the terminology that relates to the construction of multi-server namespaces out of a set of local per-server namespaces:

* Each server has a set of exported file systems that may be accessed by NFSv4 clients. Typically, this is done by assigning each file system a name within the pseudo-fs associated with the server, although the pseudo-fs may be dispensed with if there is

only a single exported file system. Each such file system is part of the server's local namespace, and can be considered as a file system instance within a larger multi-server namespace.

* The set of all exported file systems for a given server constitutes that server's local namespace.

* In some cases, a server will have a namespace more extensive than its local namespace by using features associated with attributes that provide file system location information. These features, which allow construction of a multi-server namespace, are all described in individual sections below and include referrals (Section 16.5.6), migration (Section 16.5.5), and replication (Section 16.5.4).

* A file system present in a server's pseudo-fs may have multiple file system instances on different servers associated with it. All such instances are considered replicas of one another. Whether such replicas can be used simultaneously is discussed in Section 16.11.1, while the level of coordination between them (important when switching between them) is discussed in Sections 16.11.2 through 16.11.8 below.

* When a file system is present in a server's pseudo-fs, but there is no corresponding local file system, it is said to be "absent". In such cases, all associated instances will be accessed on other servers.

Regarding the terminology that relates to attributes used in trunking discovery and other multi-server namespace features:

* File system location attributes include the fs_locations and fs_locations_info attributes.

* File system location entries provide the individual file system locations within the file system location attributes. Each such entry specifies a server, in the form of a hostname or an address, and an fs name, which designates the location of the file system within the server's local namespace. A file system location entry designates a set of server endpoints to which the client may establish connections. There may be multiple endpoints because a hostname may map to multiple network addresses and because multiple connection types may be used to communicate with a single network address. However, except where explicit port numbers are used to designate a set of servers within a single server node, all such endpoints MUST designate a way of connecting to a single server. The exact form of the location entry varies with the particular file system location attribute used, as described in Section 16.2.

  The network addresses used in file system location entries typically appear without port number indications and are used to designate a server at one of the standard ports for NFS access, e.g., 2049 for TCP or 20049 for use with RPC-over-RDMA. Port numbers may be used in file system location entries to designate servers (typically user-level ones) accessed using other port numbers. In the case where network addresses indicate trunking relationships, the use of an explicit port number is inappropriate since trunking is a relationship between network addresses. See Section 16.5.2 for details.

* File system location elements are derived from location entries, and each describes a particular network access path consisting of a network address and a location within the server's local namespace. Such location elements need not appear within a file system location attribute, but the existence of each location element derives from a corresponding location entry. When a location entry specifies an IP address, there is only a single corresponding location element. File system location entries that contain a hostname are resolved using DNS, and may result in one or more location elements. All location elements consist of a location address that includes the IP address of an interface to a server and an fs name, which is the location of the file system within the server's local namespace. The fs name can be empty if the server has no pseudo-fs and only a single exported file system at the root filehandle.

* Two file system location elements are said to be server-trunkable if they specify the same fs name and the location addresses are such that the location addresses are server-trunkable. When the corresponding network paths are used, the client will always be able to use client ID trunking, but will only be able to use session trunking if the paths are also session-trunkable.

* Two file system location elements are said to be session-trunkable if they specify the same fs name and the location addresses are such that the location addresses are session-trunkable. When the corresponding network paths are used, the client will be able to able to use either client ID trunking or session trunking.

Discussion of the term "replica" is complicated by the fact that the term was used in [RFC5661] with a meaning different from that used in this document. In short, in [RFC5661] each replica is identified by a single network access path, while in the current document, a set of network access paths that have server-trunkable network addresses and the same root-relative file system pathname is considered to be a single replica with multiple network access paths.

Each set of server-trunkable location elements defines a set of available network access paths to a particular file system. When there are multiple such file systems, each of which containing the same data, these file systems are considered replicas of one another. Logically, such replication is symmetric, since the fs currently in use and an alternate fs are replicas of each other. Often, in other documents, the term "replica" is not applied to the fs currently in use, despite the fact that the replication relation is inherently symmetric.

16.2. File System Location Attributes

NFSv4.1 contains attributes that provide information about how a given file system may be accessed (i.e., at what network address and namespace position). As a result, file systems in the namespace of one server can be associated with one or more instances of that file system on other servers. These attributes contain file system location entries specifying a server address target (either as a DNS name representing one or more IP addresses or as a specific IP address) together with the pathname of that file system within the associated single-server namespace.

The fs_locations_info attribute allows specification of one or more file system instance locations where the data corresponding to a given file system may be found. In addition to the specification of file system instance locations, this attribute provides helpful information to do the following:

* Guide choices among the various file system instances provided (e.g., priority for use, writability, currency, etc.).

* Help the client efficiently effect as seamless a transition as possible among multiple file system instances, when and if that should be necessary.

* Guide the selection of the appropriate connection type to be used when establishing a connection.

Within the fs_locations_info attribute, each fs_locations_server4 entry corresponds to a file system location entry: the fls_server field designates the server, and the fl_rootpath field of the encompassing fs_locations_item4 gives the location pathname within the server's pseudo-fs.

The fs_locations attribute defined in NFSv4.0 is also a part of NFSv4.1. This attribute 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 preferable because it provides more information.

Within the fs_locations attribute, each fs_location4 contains a file system location entry with the server field designating the server and the rootpath field giving the location pathname within the server's pseudo-fs.

16.3. 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 that once was present, and has only become absent later, this is only one possibility. A position in the namespace may be permanently absent with the set of file system(s) designated by the location attributes being the only realization. The name NFS4ERR_MOVED reflects an earlier, more limited conception of its function, but this error will be returned whenever the referenced file system is absent, whether it has moved or 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 is returned, to indicate that the file system is absent on the current server.

Because a GETFH cannot succeed if the current filehandle is within an absent file system, filehandles within an absent file system cannot be transferred to the client. When a client does have filehandles within an absent file system, it is the result of obtaining them when the file system was present, and having the file system become absent subsequently.

It should be noted that because the check for the current filehandle being within an absent file system happens at the start of every operation, operations that change the current filehandle so that it is within an absent file system will not result in an error. This allows such combinations as PUTFH-GETATTR and LOOKUP-GETATTR to be used to get attribute information, particularly location attribute information, as discussed below.

The file system attribute fs_status can be used to interrogate the present/absent status of a given file system.

16.4. 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.

16.4.1. 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 OPTIONAL attributes, they should be available 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 change.

fsid: This attribute should be provided so that the client can determine file system boundaries, including, in particular, the boundary between present and absent file systems. This value must be different from any other fsid on the current server and need have no particular relationship to fsids on any particular destination to which the client might be directed.

mounted_on_fileid: For objects at the top of an absent file system, this attribute needs to be available. Since the fileid is within the present parent file system, there should be no need to reference the absent file system to provide this information.

Other attributes SHOULD NOT be made available for absent file systems, even when it is possible to provide them. The server should not assume that more information is always better and should avoid gratuitously providing additional information.

When a GETATTR operation includes a bit mask for one of the attributes fs_locations, fs_locations_info, or fs_status, but where the bit mask includes attributes that are not supported, GETATTR will not return an error, but will return the mask of the actual attributes supported with the results.

Handling of VERIFY/NVERIFY is similar to GETATTR in that if the attribute mask does not include fs_locations, 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.

16.4.2. 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:

* 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.

* 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.

* 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.

* 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.

16.5. Uses of File System Location Information

The file system location attributes (i.e., fs_locations and fs_locations_info), together with the possibility of absent file systems, provide a number of important facilities for reliable, manageable, and scalable data access.

When a file system is present, these attributes can provide the following:

* The locations of alternative replicas 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 replica impossible or otherwise impractical. Provisioning and use of such alternate replicas is referred to as "replication" and is discussed in Section 16.5.4 below.

* The network address(es) to be used to access the current file system instance or replicas of it. Client use of this information is discussed in Section 16.5.2 below.

Under some circumstances, multiple replicas may be used simultaneously to provide higher-performance access to the file system in question, although the lack of state sharing between servers may be an impediment to such use.

When a file system is present but becomes absent, clients can be given the opportunity to have continued access to their data using a different replica. In this case, a continued attempt to use the data in the now-absent file system will result in an NFS4ERR_MOVED error, and then the successor replica or set of possible replica choices can be fetched and used to continue access. Transfer of access to the new replica location is referred to as "migration" and is discussed in Section 16.5.4 below.

When a file system is currently absent, 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 remote instance, in place of a file system not previously present, is called a "pure referral" and is discussed in Section 16.5.6 below.

Because client support for attributes related to file system location is OPTIONAL, a server may choose 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 23.35.3).

16.5.1. Combining Multiple Uses in a Single Attribute

A file system location attribute will sometimes contain information relating to the location of multiple replicas, which may be used in different ways:

* File system location entries that relate to the file system instance currently in use provide trunking information, allowing the client to find additional network addresses by which the instance may be accessed.

* File system location entries that provide information about replicas to which access is to be transferred.

* Other file system location entries that relate to replicas that are available to use in the event that access to the current replica becomes unsatisfactory.

In order to simplify client handling and to allow the best choice of replicas to access, the server should adhere to the following guidelines:

* All file system location entries that relate to a single file system instance should be adjacent.

* File system location entries that relate to the instance currently in use should appear first.

* File system location entries that relate to replica(s) to which migration is occurring should appear before replicas that are available for later use if the current replica should become inaccessible.

16.5.2. File System Location Attributes and Trunking

Trunking is the use of multiple connections between a client and server in order to increase the speed of data transfer. A client may determine the set of network addresses to use to access a given file system in a number of ways:

* When the name of the server is known to the client, it may use DNS to obtain a set of network addresses to use in accessing the server.

* The client may fetch the file system location attribute for the file system. This will provide either the name of the server (which can be turned into a set of network addresses using DNS) or a set of server-trunkable location entries. Using the latter

alternative, the server can provide addresses it regards as desirable to use to access the file system in question. Although these entries can contain port numbers, these port numbers are not used in determining trunking relationships. Once the candidate addresses have been determined and EXCHANGE_ID done to the proper server, only the value of the so_major_id field returned by the servers in question determines whether a trunking relationship actually exists.

When the client fetches a location attribute for a file system, it should be noted that the client may encounter multiple entries for a number of reasons, such that when it determines trunking information, it may need to bypass addresses not trunkable with one already known.

The server can provide location entries that include either names or network addresses. It might use the latter form because of DNS-related security concerns or because the set of addresses to be used might require active management by the server.

Location entries used to discover candidate addresses for use in trunking are subject to change, as discussed in Section 16.5.7 below. The client may respond to such changes by using additional addresses once they are verified or by ceasing to use existing ones. The server can force the client to cease using an address by returning NFS4ERR_MOVED when that address is used to access a file system. This allows a transfer of client access that is similar to migration, although the same file system instance is accessed throughout.

16.5.3. File System Location Attributes and Connection Type Selection

Because of the need to support multiple types of connections, clients face the issue of determining the proper connection type to use when establishing a connection to a given server network address. In some cases, this issue can be addressed through the use of the connection "step-up" facility described in Section 23.36. However, because there are cases in which that facility is not available, the client may have to choose a connection type with no possibility of changing it within the scope of a single connection.

The two file system location attributes differ as to the information made available in this regard. The fs_locations attribute provides no information to support connection type selection. As a result, clients supporting multiple connection types would need to attempt to establish connections using multiple connection types until the one preferred by the client is successfully established.

The fs_locations_info attribute includes the FSLI4TF_RDMA flag, which is convenient for a client wishing to use RDMA. When this flag is set, it indicates that RPC-over-RDMA support is available using the specified location entry. A client can establish a TCP connection and then convert that connection to use RDMA by using the step-up facility.

Irrespective of the particular attribute used, when there is no indication that a step-up operation can be performed, a client supporting RDMA operation can establish a new RDMA connection, and it can be bound to the session already established by the TCP connection, allowing the TCP connection to be dropped and the session converted to further use in RDMA mode, if the server supports that.

16.5.4. File System Replication

The fs_locations and fs_locations_info attributes provide alternative file system 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 set of alternate locations by interrogating the fs_locations or fs_locations_info attribute, with the latter being preferred.

In the event that the occurrence of server failures, communications problems, or other difficulties make continued access to the current file system impossible or otherwise impractical, the client can use the alternate locations as a way to get continued access to its data.

The alternate locations may be physical replicas of the (typically read-only) file system data supplemented by possible asynchronous propagation of updates. Alternatively, they may 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 the difference between replicas affects file system transitions can be 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 in later sections.

Although the location attributes provide some information about the nature of the inter-replica transition, many aspects of the semantics of possible asynchronous updates are not currently described by the protocol, which makes it necessary for clients using replication to switch among replicas undergoing change to familiarize themselves with the semantics of the update approach used. Due to this lack of specificity, many applications may find the use of migration more appropriate because a server can propagate all updates made before an established point in time to the new replica as part of the migration event.

16.5.4.1. File System Trunking Presented as Replication

In some situations, a file system location entry may indicate a file system access path to be used as an alternate location, where trunking, rather than replication, is to be used. The situations in which this is appropriate are limited to those in which both of the following are true:

* The two file system locations (i.e., the one on which the location attribute is obtained and the one specified in the file system location entry) designate the same locations within their respective single-server namespaces.

* The two server network addresses (i.e., the one being used to obtain the location attribute and the one specified in the file system location entry) designate the same server (as indicated by the same value of the so_major_id field of the eir_server_owner field returned in response to EXCHANGE_ID).

When these conditions hold, operations using both access paths are generally trunked, although trunking may be disallowed when the attribute fs_locations_info is used:

* When the fs_locations_info attribute shows the two entries as not having the same simultaneous-use class, trunking is inhibited, and the two access paths cannot be used together.

  In this case, the two paths can be used serially with no transition activity required on the part of the client, and any transition between access paths is transparent. In transferring access from one to the other, the client acts as if communication were interrupted, establishing a new connection and possibly a new session to continue access to the same file system.

* Note that for two such location entries, 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 location entries with different handle, fileid, write-verifier, change, and readdir classes, indicates a serious problem. The client, if it allows transition to the file system instance at all, must not treat any transition as a transparent one. The server SHOULD NOT indicate that these two entries (for the same file system on the same server) belong to different handle, fileid, write-verifier, change, and readdir classes, whether or not the two entries are shown belonging to the same simultaneous-use class.

These situations were recognized by [RFC5661], even though that document made no explicit mention of trunking:

* It treated the situation that we describe as trunking as one of simultaneous use of two distinct file system instances, even though, in the explanatory framework now used to describe the situation, the case is one in which a single file system is accessed by two different trunked addresses.

* It treated the situation in which two paths are to be used serially as a special sort of "transparent transition". However, in the descriptive framework now used to categorize transition situations, this is considered a case of a "network endpoint transition" (see Section 16.9).

16.5.5. File System Migration

When a file system is present and becomes inaccessible using the current access path, the NFSv4.1 protocol provides a means by which clients can be given the opportunity to have continued access to their data. This may involve using a different access path to the existing replica or providing a path to a different replica. The new access path or the location of the new replica is specified by a file system location attribute. The ensuing migration of access includes the ability to retain locks across the transition. Depending on circumstances, this can involve:

* The continued use of the existing clientid when accessing the current replica using a new access path.

* Use of lock reclaim, taking advantage of a per-fs grace period.

* Use of Transparent State Migration.

Typically, a client will be accessing the file system in question, get an NFS4ERR_MOVED error, and then use a file system location attribute to determine the new access path for the data. When fs_locations_info is used, additional information will be available that will define the nature of the client’s handling of the transition to a new server.

In most instances, servers will choose to migrate all clients using a particular file system to a successor replica at the same time to avoid cases in which different clients are updating different replicas. However, migration of an individual client can be helpful in providing load balancing, as long as the replicas in question are such that they represent the same data as described in Section 16.11.8.

* In the case in which there is no transition between replicas (i.e., only a change in access path), there are no special difficulties in using of this mechanism to effect load balancing.

* In the case in which the two replicas are sufficiently coordinated as to allow a single client coherent, simultaneous access to both, there is, in general, no obstacle to the use of migration of particular clients to effect load balancing. Generally, such simultaneous use involves cooperation between servers to ensure that locks granted on two coordinated replicas cannot conflict and can remain effective when transferred to a common replica.

* In the case in which a large set of clients is accessing a file system in a read-only fashion, it can be helpful to migrate all clients with writable access simultaneously, while using load balancing on the set of read-only copies, as long as the rules in Section 16.11.8, which are designed to prevent data reversion, are followed.

In other cases, the client might not have sufficient guarantees of data similarity or coherence to function properly (e.g., the data in the two replicas is similar but not identical), and the possibility that different clients are updating different replicas can exacerbate the difficulties, making the use of load balancing in such situations a perilous enterprise.

The protocol does not specify how the file system will be moved between servers or how updates to multiple replicas will be coordinated. It is anticipated that a number of different server-to-server coordination 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.

In the case of various forms of server clustering, the new location may be another server providing access to the same physical file system. The client’s responsibilities in dealing with this transition will depend on whether a switch between replicas has occurred and the means the server has chosen to provide continuity of locking state. These issues will be discussed in detail below.

Although a single successor location is typical, multiple locations may be provided. When multiple locations are provided, the client will typically use the first one provided. If that is inaccessible for some reason, later ones can be used. In such cases, the client might consider the transition to the new replica to be a migration event, even though some of the servers involved might not be aware of the use of the server that was inaccessible. In such a case, a client might lose access to locking state as a result of the access transfer.

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.

16.5.6. Referrals

Referrals allow the server to associate a file system namespace entry located on one server with a file system located on another server. When this includes the use of pure referrals, servers are provided a way of placing a file system in a location within the namespace essentially without respect to its physical location on a particular server. This allows a single server or a set of servers to present a multi-server namespace that encompasses file systems located on a wider range of servers. Some likely uses of this facility include establishment of site-wide or organization-wide namespaces, with the eventual possibility of combining such together into a truly global namespace, such as the one provided by AFS (the Andrew File System) [AFS].

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 upon receiving the error NFS4ERR_MOVED, the actual location or locations of the file system can be determined by fetching a locations attribute.

The file system location 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 might 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.

If the fs_locations_info attribute lists multiple possible targets, the relationships among them may be important to the client in selecting which one to use. The same rules specified in Section 16.5.5 below regarding multiple migration targets 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 pure referrals to all of the included file systems. Alternatively, a single multi-server namespace may be administratively segmented with separate referral file systems (on separate servers) for each separately administered portion of the namespace. The top-level referral file system or any segment may use replicated referral file systems for higher availability.

Generally, multi-server namespaces are for the most part uniform, in that the same data made available to one client at a given location in the namespace is made available to all clients at that namespace location. However, there are facilities provided that allow different clients to be directed to different sets of data, for reasons such as enabling adaptation to such client characteristics as CPU architecture. These facilities are described in Section 16.17.3.

Note that it is possible, when providing a uniform namespace, to provide different location entries to different clients in order to provide each client with a copy of the data physically closest to it or otherwise optimize access (e.g., provide load balancing).

16.5.7. Changes in File System Location Attributes

Although clients will typically fetch a file system location attribute when first accessing a file system and when NFS4ERR_MOVED is returned, a client can choose to fetch the attribute periodically, in which case, the value fetched may change over time.

For clients not prepared to access multiple replicas simultaneously (see Section 16.11.1), the handling of the various cases of location change are as follows:

* Changes in the list of replicas or in the network addresses associated with replicas do not require immediate action. The client will typically update its list of replicas to reflect the new information.

* Additions to the list of network addresses for the current file system instance need not be acted on promptly. However, to prepare for a subsequent migration event, the client can choose to take note of the new address and then use it whenever it needs to switch access to a new replica.

* Deletions from the list of network addresses for the current file system instance do not require the client to immediately cease use of existing access paths, although new connections are not to be established on addresses that have been deleted. However, clients can choose to act on such deletions by preparing for an eventual shift in access, which becomes unavoidable as soon as the server returns NFS4ERR_MOVED to indicate that a particular network access path is not usable to access the current file system.

For clients that are prepared to access several replicas simultaneously, the following additional cases need to be addressed. As in the cases discussed above, changes in the set of replicas need not be acted upon promptly, although the client has the option of adjusting its access even in the absence of difficulties that would lead to the selection of a new replica.

* When a new replica is added, which may be accessed simultaneously with one currently in use, the client is free to use the new replica immediately.

* When a replica currently in use is deleted from the list, the client need not cease using it immediately. However, since the server may subsequently force such use to cease (by returning NFS4ERR_MOVED), clients might decide to limit the need for later state transfer. For example, new opens might be done on other replicas, rather than on one not present in the list.

16.6. Trunking without File System Location Information

In situations in which a file system is accessed using two server-trunkable addresses (as indicated by the same value of the so_major_id field of the eir_server_owner field returned in response to EXCHANGE_ID), trunked access is allowed even though there might not be any location entries specifically indicating the use of trunking for that file system.

This situation was recognized by [RFC5661], although that document made no explicit mention of trunking and treated the situation as one of simultaneous use of two distinct file system instances. In the explanatory framework now used to describe the situation, the case is one in which a single file system is accessed by two different trunked addresses.

16.7. Users and Groups in a Multi-Server Namespace

As in the case of a single-server environment (see Section 11.13), when an owner or group name of the form "id@domain" is assigned to a file, there is an implicit promise to return that same string when the corresponding attribute is interrogated subsequently. In the case of a multi-server namespace, that same promise applies even if server boundaries have been crossed. Similarly, when the owner attribute of a file is derived from the security principal that created the file, that attribute should have the same value even if the interrogation occurs on a different server from the file creation.

Similarly, the set of security principals recognized by all the participating servers needs to be the same, with each such principal having the same credentials, regardless of the particular server being accessed.

In order to meet these requirements, those setting up multi-server namespaces will need to limit the servers included so that:

* In all cases in which more than a single domain is supported, the requirements stated in [RFC8000] are to be respected.

* All servers support a common set of domains that includes all of the domains clients use and expect to see returned as the domain portion of an owner or group in the form "id@domain". Note that, although this set most often consists of a single domain, it is possible for multiple domains to be supported.

* All servers, for each domain that they support, accept the same set of user and group ids as valid.

* All servers recognize the same set of security principals. For each principal, the same credential is required, independent of the server being accessed. In addition, the group membership for each such principal is to be the same, independent of the server accessed.

Note that there is no requirement in general that the users corresponding to particular security principals have the same local representation on each server, even though it is most often the case that this is so.

When AUTH_SYS is used, the following additional requirements must be met:

* Only a single NFSv4 domain can be supported through the use of AUTH_SYS.

* The "local" representation of all owners and groups must be the same on all servers. The word "local" is used here since that is the way that numeric user and group ids are described in Section 11.13. However, when AUTH_SYS or stringified numeric owners or groups are used, these identifiers are not truly local, since they are known to the clients as well as to the server.

Similarly, when stringified numeric user and group ids are used, the "local" representation of all owners and groups must be the same on all servers, even when AUTH_SYS is not used.

16.8. Additional Client-Side Considerations

When clients make use of servers that implement referrals, replication, and migration, care should be taken that a user who mounts a given file system that includes a referral or a relocated file system continues to see a coherent picture of that user-side file system despite the fact that it contains a number of server-side file systems that may be on different servers.

One important issue is upward navigation from the root of a server-side file system to its parent (specified as ".." in UNIX), in the case in which it transitions to that file system as a result of referral, migration, or a transition as a result of replication. When the client is at such a point, and it needs to ascend to the parent, it must go back to the parent as seen within the multi-server namespace rather than sending a LOOKUPP operation to the server, which would result in the parent within that server’s single-server namespace. In order to do this, the client needs to remember the filehandles that represent such file system roots and use these instead of sending a LOOKUPP operation to the current server. This

will allow the client to present to applications a consistent namespace, where upward navigation and downward navigation are consistent.

Another issue concerns refresh of referral locations. When referrals are used extensively, they may change as server configurations change. It is expected that clients will cache information related to traversing referrals so that future client-side requests are resolved locally without server communication. This is usually rooted in client-side name look up caching. Clients should periodically purge this data for referral points in order to detect changes in location information. 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.

16.9. Overview of File Access Transitions

File access transitions are of two types:

* Those that involve a transition from accessing the current replica to another one in connection with either replication or migration. How these are dealt with is discussed in Section 16.11.

* Those in which access to the current file system instance is retained, while the network path used to access that instance is changed. This case is discussed in Section 16.10.

16.10. Effecting Network Endpoint Transitions

The endpoints used to access a particular file system instance may change in a number of ways, as listed below. In each of these cases, the same fsid, client IDs, filehandles, and stateids are used to continue access, with a continuity of lock state. In many cases, the same sessions can also be used.

The appropriate action depends on the set of replacement addresses that are available for use (i.e., server endpoints that are server-trunkable with one previously being used).

* When use of a particular address is to cease, and there is also another address currently in use that is server-trunkable with it, requests that would have been issued on the address whose use is to be discontinued can be issued on the remaining address(es). When an address is server-trunkable but not session-trunkable with the address whose use is to be discontinued, the request might need to be modified to reflect the fact that a different session will be used.

* When use of a particular connection is to cease, as indicated by receiving NFS4ERR_MOVED when using that connection, but that address is still indicated as accessible according to the appropriate file system location entries, it is likely that requests can be issued on a new connection of a different connection type once that connection is established. Since any two non-port-specific server endpoints that share a network address are inherently session-trunkable, the client can use BIND_CONN_TO_SESSION to access the existing session with the new connection.

* When there are no potential replacement addresses in use, but there are valid addresses session-trunkable with the one whose use is to be discontinued, the client can use BIND_CONN_TO_SESSION to access the existing session using the new address. Although the target session will generally be accessible, there may be rare situations in which that session is no longer accessible when an attempt is made to bind the new connection to it. In this case, the client can create a new session to enable continued access to the existing instance using the new connection, providing for the use of existing filehandles, stateids, and client ids while supplying continuity of locking state.

* When there is no potential replacement address in use, and there are no valid addresses session-trunkable with the one whose use is to be discontinued, other server-trunkable addresses may be used to provide continued access. Although the use of CREATE_SESSION is available to provide continued access to the existing instance, servers have the option of providing continued access to the existing session through the new network access path in a fashion similar to that provided by session migration (see Section 16.12). To take advantage of this possibility, clients can perform an initial BIND_CONN_TO_SESSION, as in the previous case, and use CREATE_SESSION only if that fails.

16.11. Effecting File System Transitions

There are a range of situations in which there is a change to be effected in the set of replicas used to access a particular file system. Some of these may involve an expansion or contraction of the set of replicas used as discussed in Section 16.11.1 below.

For reasons explained in that section, most transitions will involve a transition from a single replica to a corresponding replacement replica. When effecting replica transition, some types of sharing between the replicas may affect handling of the transition as described in Sections 16.11.2 through 16.11.8 below. The attribute fs_locations_info provides helpful information to allow the client to determine the degree of inter-replica sharing.

With regard to some types of state, the degree of continuity across the transition depends on the occasion prompting the transition, with transitions initiated by the servers (i.e., migration) offering much more scope for a nondisruptive transition than cases in which the client on its own shifts its access to another replica (i.e., replication). This issue potentially applies to locking state and to session state, which are dealt with below as follows:

* An introduction to the possible means of providing continuity in these areas appears in Section 16.11.9 below.

* Transparent State Migration is introduced in Section 16.12. The possible transfer of session state is addressed there as well.

* The client handling of transitions, including determining how to deal with the various means that the server might take to supply effective continuity of locking state, is discussed in Section 16.13.

* The source and destination servers' responsibilities in effecting Transparent State Migration of locking and session state are discussed in Section 16.14.

16.11.1. File System Transitions and Simultaneous Access

The fs_locations_info attribute (described in Section 16.17) may indicate that two replicas may be used simultaneously, although some situations in which such simultaneous access is permitted are more appropriately described as instances of trunking (see Section 16.5.4.1). Although situations in which multiple replicas may be accessed simultaneously are somewhat similar to those in which a single replica is accessed by multiple network addresses, there are important differences since locking state is not shared among multiple replicas.

Because of this difference in state handling, many clients will not have the ability to take advantage of the fact that such replicas represent the same data. Such clients will not be prepared to use multiple replicas simultaneously but will access each file system using only a single replica, although the replica selected might make multiple server-trunkable addresses available.

Clients who are prepared to use multiple replicas simultaneously can divide opens among replicas however they choose. Once that choice is made, any subsequent transitions will treat the set of locking state associated with each replica as a single entity.

For example, if one of the replicas become unavailable, access will be transferred to a different replica, which is also capable of simultaneous access with the one still in use.

When there is no such replica, the transition may be to the replica already in use. At this point, the client has a choice between merging the locking state for the two replicas under the aegis of the sole replica in use or treating these separately until another replica capable of simultaneous access presents itself.

16.11.2. 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 cooperation 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 the fh_expire_type attribute and supersedes the specification of the FH4_VOL_MIGRATION bit, which only affects behavior when fs_locations_info is not available.

When there is cooperation 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.

16.11.3. 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 would be consistent across the transition, while in the analogous multi-vendor transitions they would 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 APIs 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). The result 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.

16.11.4. Fsids and File System Transitions

Since fsids are generally only unique on a per-server basis, it is likely that they will change during a file system transition. 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.

16.11.4.1. File System Splitting

When a file system transition is made and the fs_locations_info indicates that the file system in question might 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 might choose to 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 can 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.

16.11.5. 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 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.

16.11.6. Write Verifiers and File System Transitions

In a file system transition, the two file systems might be cooperating in the handling of unstably written data. Clients can determine if this is the case by seeing if the two file systems belong to the same write-verifier class. When this is the case, write verifiers returned from one system may be compared to those returned by the other and superfluous writes can be 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, the two verifiers should be treated as not equal even when the values are identical.

16.11.7. READDIR Cookies and Verifiers and File System Transitions

In a file system transition, the two file systems might be consistent in their handling of READDIR cookies and verifiers. Clients can determine if this is the case by seeing if the two file systems belong to the same readdir class. When this is the case, readdir class, READDIR cookies, and verifiers from one system will be recognized by the other, and READDIR operations started on one server can 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 were rejected.

16.11.8. 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 either the same data or data that 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 might be listed, while in another, only those that are up-to-date would 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:

* 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 at the same time, regardless of whether the interrogated instance is the one on which the modification was done. This allows a client to use these replicas simultaneously without any special adaptation to the fact that there are multiple replicas, beyond adapting to the fact that locks obtained on one replica are maintained separately (i.e., under a different client ID). In this case, locks (whether share reservations or byte-range locks) and delegations obtained on one replica are immediately reflected on all replicas, in the sense that access from all other servers is prevented regardless of the replica used. However, because the servers are not required to treat two associated client IDs as representing the same client, it is best to access each file using only a single client ID.

* When one replica is designated as the successor instance to another existing instance after the return of NFS4ERR_MOVED (i.e., the case of migration), the client may depend on the fact that all

changes written to stable storage on the original instance are written to stable storage of the successor (uncommitted writes are dealt with in Section 16.11.6 above).

* 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 of this guarantee varies based on the value of the fss_type field that is reported as part of the fs_status attribute (see Section 16.18). Since these file systems are presumed to be unsuitable for simultaneous use, there is no specification of how locking is handled; in general, locks obtained on one file system will be separate from those on others. Since these are expected to be read-only file systems, this is not likely to pose an issue for clients or applications.

When none of these special situations applies, there is no basis within the protocol for the client to make assumptions about the contents of a replica file system or its relationship to previous file system instances. Thus, switching between nominally identical read-write file systems would not be possible because either the client does not use the fs_locations_info attribute, or the server does not support it.

16.11.9. Lock State and File System Transitions

While accessing a file system, clients obtain locks enforced by the server, which may prevent actions by other clients that are inconsistent with those locks.

When access is transferred between replicas, clients need to be assured that the actions disallowed by holding these locks cannot have occurred during the transition. This can be ensured by the methods below. Unless at least one of these is implemented, clients will not be assured of continuity of lock possession across a migration event:

* Providing the client an opportunity to re-obtain his locks via a per-fs grace period on the destination server, denying all clients using the destination file system the opportunity to obtain new locks that conflict with those held by the transferred client as long as that client has not completed its per-fs grace period. Because the lock reclaim mechanism was originally defined to

support server reboot, it implicitly assumes that filehandles will, upon reclaim, be the same as those at open. In the case of migration, this requires that source and destination servers use the same filehandles, as evidenced by using the same server scope (see Section 7.4) or by showing this agreement using fs_locations_info (see Section 16.11.2 above).

Note that such a grace period can be implemented without interfering with the ability of non-transferred clients to obtain new locks while it is going on. As long as the destination server is aware of the transferred locks, it can distinguish requests to obtain new locks that contrast with existing locks from those that do not, allowing it to treat such client requests without reference to the ongoing grace period.

* Locking state can be transferred as part of the transition by providing Transparent State Migration as described in Section 16.12.

Of these, Transparent State Migration provides the smoother experience for clients in that there is no need to go through a reclaim process before new locks can be obtained; however, it requires a greater degree of inter-server coordination. In general, the servers taking part in migration are free to provide either facility. However, when the filehandles can differ across the migration event, Transparent State Migration is the only available means of providing the needed functionality.

It should be noted that these two methods are not mutually exclusive and that a server might well provide both. In particular, if there is some circumstance preventing a specific lock from being transferred transparently, the destination server can allow it to be reclaimed by implementing a per-fs grace period for the migrated file system.

16.11.9.1. Security Issues Related to Reclaiming Lock State after File System Transitions

Although it is possible for a client reclaiming state to misrepresent its state in the same fashion as described in Section 13.4.2.1.1, most implementations providing for such reclamation in the case of file system transitions will have the ability to detect such misrepresentations. This limits the ability of unauthenticated clients to execute denial-of-service attacks in these circumstances. Nevertheless, the rules stated in Section 13.4.2.1.1 regarding principal verification for reclaim requests apply in this situation as well.

Typically, implementations that support file system transitions will have extensive information about the locks to be transferred. This is because of the following:

* Since failure is not involved, there is no need to store locking information in persistent storage.

* There is no need, as there is in the failure case, to update multiple repositories containing locking state to keep them in sync. Instead, there is a one-time communication of locking state from the source to the destination server.

* Providing this information avoids potential interference with existing clients using the destination file system by denying them the ability to obtain new locks during the grace period.

When such detailed locking information, not necessarily including the associated stateids, is available:

* It is possible to detect reclaim requests that attempt to reclaim locks that did not exist before the transfer, rejecting them with NFS4ERR_RECLAIM_BAD (Section 20.1.9.4). SN

* It is possible when dealing with non-reclaim requests, to determine whether they conflict with existing locks, eliminating the need to return NFS4ERR_GRACE (Section 20.1.9.2) on non-reclaim requests.

It is possible for implementations of grace periods in connection with file system transitions not to have detailed locking information available at the destination server, in which case, the security situation is exactly as described in Section 13.4.2.1.1.

16.11.9.2. 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 its lease where there is locking state that may have been relocated to the new server, the client must find out about lease relocation before that lease expire. To accomplish this, the SEQUENCE operation will return the status bit SEQ4_STATUS_LEASE_MOVED if responsibility for any of the renewed locking state 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 from a server, for each file system of the server for which the client has locking state, the client should perform an operation. For simplicity, the client may choose to reference all file systems, but what is important is that it must reference all file systems for which there was locking state where that state has moved. Once the client receives an NFS4ERR_MOVED error for each such file system, the server will clear the SEQ4_STATUS_LEASE_MOVED indication. 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 that receives NFS4ERR_MOVED on each file system, in order to clear the SEQ4_STATUS_LEASE_MOVED indication.

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 that lease on the new server, unless information in the 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.

16.11.9.3. Transitions and the Lease_time Attribute

In order that the client may appropriately manage its lease 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 a lease granted by the source server. Upon transitions in which state is transferred transparently, the client is under no obligation to refetch 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 of 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.

16.12. Transferring State upon Migration

When the transition is a result of a server-initiated decision to transition access, and the source and destination servers have implemented appropriate cooperation, it is possible to do the following:

* Transfer locking state from the source to the destination server in a fashion similar to that provided by Transparent State Migration in NFSv4.0, as described in [RFC7931]. Server responsibilities are described in Section 16.14.2.

* Transfer session state from the source to the destination server. Server responsibilities in effecting such a transfer are described in Section 16.14.3.

The means by which the client determines which of these transfer events has occurred are described in Section 16.13.

16.12.1. Transparent State Migration and pNFS

When pNFS is involved, the protocol is capable of supporting:

* Migration of the Metadata Server (MDS), leaving the Data Servers (DSs) in place.

* Migration of the file system as a whole, including the MDS and associated DSs.

* Replacement of one DS by another.

* Migration of a pNFS file system to one in which pNFS is not used.

* Migration of a file system not using pNFS to one in which layouts are available.

Note that migration, per se, is only involved in the transfer of the MDS function. Although the servicing of a layout may be transferred from one data server to another, this not done using the file system location attributes. The MDS can effect such transfers by recalling or revoking existing layouts and granting new ones on a different data server.

Migration of the MDS function is directly supported by Transparent State Migration. Layout state will normally be transparently transferred, just as other state is. As a result, Transparent State Migration provides a framework in which, given appropriate inter-MDS data transfer, one MDS can be substituted for another.

Migration of the file system function as a whole can be accomplished by recalling all layouts as part of the initial phase of the migration process. As a result, I/O will be done through the MDS during the migration process, and new layouts can be granted once the client is interacting with the new MDS. An MDS can also effect this sort of transition by revoking all layouts as part of Transparent State Migration, as long as the client is notified about the loss of locking state.

In order to allow migration to a file system on which pNFS is not supported, clients need to be prepared for a situation in which layouts are not available or supported on the destination file system and so direct I/O requests to the destination server, rather than depending on layouts being available.

Replacement of one DS by another is not addressed by migration as such but can be effected by an MDS recalling layouts for the DS to be replaced and issuing new ones to be served by the successor DS.

Migration may transfer a file system from a server that does not support pNFS to one that does. In order to properly adapt to this situation, clients that support pNFS, but function adequately in its absence, should check for pNFS support when a file system is migrated and be prepared to use pNFS when support is available on the destination.

16.13. Client Responsibilities When Access Is Transitioned

For a client to respond to an access transition, it must become aware of it. The ways in which this can happen are discussed in Section 16.13.1, which discusses indications that a specific file system access path has transitioned as well as situations in which additional activity is necessary to determine the set of file systems that have been migrated. Section 16.13.2 goes on to complete the discussion of how the set of migrated file systems might be determined. Sections 16.13.3 through 16.13.5 discuss how the client should deal with each transition it becomes aware of, either directly or as a result of migration discovery.

The following terms are used to describe client activities:

* "Transition recovery" refers to the process of restoring access to a file system on which NFS4ERR_MOVED was received.

* "Migration recovery" refers to that subset of transition recovery that applies when the file system has migrated to a different replica.

* "Migration discovery" refers to the process of determining which file system(s) have been migrated. It is necessary to avoid a situation in which leases could expire when a file system is not accessed for a long period of time, since a client unaware of the migration might be referencing an unmigrated file system and not renewing the lease associated with the migrated file system.

16.13.1. Client Transition Notifications

When there is a change in the network access path that a client is to use to access a file system, there are a number of related status indications with which clients need to deal:

* If an attempt is made to use or return a filehandle within a file system that is no longer accessible at the address previously used to access it, the error NFS4ERR_MOVED is returned.

  Exceptions are made to allow such filehandles to be used when interrogating a file system location attribute. This enables a client to determine a new replica’s location or a new network access path.

This condition continues on subsequent attempts to access the file system in question. The only way the client can avoid the error is to cease accessing the file system in question at its old server location and access it instead using a different address at which it is now available.

* Whenever a client sends a SEQUENCE operation to a server that generated state held on that client and associated with a file system no longer accessible on that server, the response will contain the status bit SEQ4_STATUS_LEASE_MOVED, indicating that there has been a lease migration.

  This condition continues until the client acknowledges the notification by fetching a file system location attribute for the file system whose network access path is being changed. When there are multiple such file systems, a location attribute for each such file system needs to be fetched. The location attribute for all migrated file systems needs to be fetched in order to clear the condition. Even after the condition is cleared, the client needs to respond by using the location information to access the file system at its new location to ensure that leases are not needlessly expired.

Unlike NFSv4.0, in which the corresponding conditions are both errors and thus mutually exclusive, in NFSv4.1 the client can, and often will, receive both indications on the same request. As a result, implementations need to address the question of how to coordinate the necessary recovery actions when both indications arrive in the response to the same request. It should be noted that when processing an NFSv4 COMPOUND, the server will normally decide whether SEQ4_STATUS_LEASE_MOVED is to be set before it determines which file system will be referenced or whether NFS4ERR_MOVED is to be returned.

Since these indications are not mutually exclusive in NFSv4.1, the following combinations are possible results when a COMPOUND is issued:

* The COMPOUND status is NFS4ERR_MOVED, and SEQ4_STATUS_LEASE_MOVED is asserted.

  In this case, transition recovery is required. While it is possible that migration discovery is needed in addition, it is likely that only the accessed file system has transitioned. In any case, because addressing NFS4ERR_MOVED is necessary to allow the rejected requests to be processed on the target, dealing with it will typically have priority over migration discovery.

* The COMPOUND status is NFS4ERR_MOVED, and SEQ4_STATUS_LEASE_MOVED is clear.

  In this case, transition recovery is also required. It is clear that migration discovery is not needed to find file systems that have been migrated other than the one returning NFS4ERR_MOVED. Cases in which this result can arise include a referral or a migration for which there is no associated locking state. This can also arise in cases in which an access path transition other than migration occurs within the same server. In such a case, there is no need to set SEQ4_STATUS_LEASE_MOVED, since the lease remains associated with the current server even though the access path has changed.

* The COMPOUND status is not NFS4ERR_MOVED, and SEQ4_STATUS_LEASE_MOVED is asserted.

  In this case, no transition recovery activity is required on the file system(s) accessed by the request. However, to prevent avoidable lease expiration, migration discovery needs to be done.

* The COMPOUND status is not NFS4ERR_MOVED, and SEQ4_STATUS_LEASE_MOVED is clear.

  In this case, neither transition-related activity nor migration discovery is required.

Note that the specified actions only need to be taken if they are not already going on. For example, when NFS4ERR_MOVED is received while accessing a file system for which transition recovery is already occurring, the client merely waits for that recovery to be completed, while the receipt of the SEQ4_STATUS_LEASE_MOVED indication only needs to initiate migration discovery for a server if such discovery is not already underway for that server.

The fact that a lease-migrated condition does not result in an error in NFSv4.1 has a number of important consequences. In addition to the fact that the two indications are not mutually exclusive, as discussed above, there are number of issues that are important in considering implementation of migration discovery, as discussed in Section 16.13.2.

Because SEQ4_STATUS_LEASE_MOVED is not an error condition, it is possible for file systems whose access paths have not changed to be successfully accessed on a given server even though recovery is necessary for other file systems on the same server. As a result, access can take place while:

* The migration discovery process is happening for that server.

* The transition recovery process is happening for other file systems connected to that server.

16.13.2. Performing Migration Discovery

Migration discovery can be performed in the same context as transition recovery, allowing recovery for each migrated file system to be invoked as it is discovered. Alternatively, it may be done in a separate migration discovery thread, allowing migration discovery to be done in parallel with one or more instances of transition recovery.

In either case, because the lease-migrated indication does not result in an error, other access to file systems on the server can proceed normally, with the possibility that further such indications will be received, raising the issue of how such indications are to be dealt with. In general:

* No action needs to be taken for such indications received by any threads performing migration discovery, since continuation of that work will address the issue.

* In other cases in which migration discovery is currently being performed, nothing further needs to be done to respond to such lease migration indications, as long as one can be certain that the migration discovery process would deal with those indications. See below for details.

* For such indications received in all other contexts, the appropriate response is to initiate or otherwise provide for the execution of migration discovery for file systems associated with the server IP address returning the indication.

This leaves a potential difficulty in situations in which the migration discovery process is near to completion but is still operating. One should not ignore a SEQ4_STATUS_LEASE_MOVED indication if the migration discovery process is not able to respond to the discovery of additional migrating file systems without additional aid. A further complexity relevant in addressing such situations is that a lease-migrated indication may reflect the server's state at the time the SEQUENCE operation was processed, which may be different from that in effect at the time the response is received. Because new migration events may occur at any time, and because a SEQ4_STATUS_LEASE_MOVED indication may reflect the situation in effect a considerable time before the indication is received, special care needs to be taken to ensure that SEQ4_STATUS_LEASE_MOVED indications are not inappropriately ignored.

A useful approach to this issue involves the use of separate externally-visible migration discovery states for each server. Separate values could represent the various possible states for the migration discovery process for a server:

* Non-operation, in which migration discovery is not being performed.

* Normal operation, in which there is an ongoing scan for migrated file systems.

* Completion/verification of migration discovery processing, in which the possible completion of migration discovery processing needs to be verified.

Given that framework, migration discovery processing would proceed as follows:

* While in the normal-operation state, the thread performing discovery would fetch, for successive file systems known to the client on the server being worked on, a file system location attribute plus the fs_status attribute.

* If the fs_status attribute indicates that the file system is a migrated one (i.e., fss_absent is true, and fss_type != STATUS4_REFERRAL), then a migrated file system has been found. In this situation, it is likely that the fetch of the file system location attribute has cleared one of the file systems contributing to the lease-migrated indication.

* In cases in which that happened, the thread cannot know whether the lease-migrated indication has been cleared, and so it enters the completion/verification state and proceeds to issue a COMPOUND to see if the SEQ4_STATUS_LEASE_MOVED indication has been cleared.

* When the discovery process is in the completion/verification state, if other requests get a lease-migrated indication, they note that it was received. Later, the existence of such indications is used when the request completes, as described below.

When the request used in the completion/verification state completes:

* If a lease-migrated indication is returned, the discovery continues normally. Note that this is so even if all file systems have been traversed, since new migrations could have occurred while the process was going on.

* Otherwise, if there is any record that other requests saw a lease-migrated indication while the request was occurring, that record is cleared, and the verification request is retried. The discovery process remains in the completion/verification state.

* If there have been no lease-migrated indications, the work of migration discovery is considered completed, and it enters the non-operating state. Once it enters this state, subsequent lease-migrated indications will trigger a new migration discovery process.

It should be noted that the process described above is not guaranteed to terminate, as a long series of new migration events might continually delay the clearing of the SEQ4_STATUS_LEASE_MOVED indication. To prevent unnecessary lease expiration, it is appropriate for clients to use the discovery of migrations to effect lease renewal immediately, rather than waiting for the clearing of the SEQ4_STATUS_LEASE_MOVED indication when the complete set of migrations is available.

Lease discovery needs to be provided as described above. This ensures that the client discovers file system migrations soon enough to renew its leases on each destination server before they expire. Non-renewal of leases can lead to loss of locking state. While the consequences of such loss can be ameliorated through implementations of courtesy locks, servers are under no obligation to do so, and a conflicting lock request may mean that a lock is revoked unexpectedly. Clients should be aware of this possibility.

16.13.3. Overview of Client Response to NFS4ERR_MOVED

This section outlines a way in which a client that receives NFS4ERR_MOVED can effect transition recovery by using a new server or server endpoint if one is available. As part of that process, it will determine:

* Whether the NFS4ERR_MOVED indicates migration has occurred, or whether it indicates another sort of file system access transition as discussed in Section 16.10 above.

* In the case of migration, whether Transparent State Migration has occurred.

* Whether any state has been lost during the process of Transparent State Migration.

* Whether sessions have been transferred as part of Transparent State Migration.

During the first phase of this process, the client proceeds to examine file system location entries to find the initial network address it will use to continue access to the file system or its replacement. For each location entry that the client examines, the process consists of five steps:

1. Performing an EXCHANGE_ID directed at the location address. This operation is used to register the client owner (in the form of a client_owner4) with the server, to obtain a client ID to be used subsequently to communicate with it, to obtain that client ID's confirmation status, and to determine server_owner4 and scope for the purpose of determining if the entry is trunkable with the address previously being used to access the file system (i.e., that it represents another network access path to the same file system and can share locking state with it).

2. Making an initial determination of whether migration has occurred. The initial determination will be based on whether the EXCHANGE_ID results indicate that the current location element is server-trunkable with that used to access the file system when access was terminated by receiving NFS4ERR_MOVED. If it is, then migration has not occurred. In that case, the transition is dealt with, at least initially, as one involving continued access to the same file system on the same server through a new network address.

3. Obtaining access to existing session state or creating new sessions. How this is done depends on the initial determination of whether migration has occurred and can be done as described in Section 16.13.4 below in the case of migration or as described in Section 16.13.5 below in the case of a network address transfer without migration.

4. Verifying the trunking relationship assumed in step 2 as discussed in Section 7.5.1. Although this step will generally confirm the initial determination, it is possible for verification to invalidate the initial determination of network address shift (without migration) and instead determine that migration had occurred. There is no need to redo step 3 above, since it will be possible to continue use of the session established already.

5. Obtaining access to existing locking state and/or re-obtaining it. How this is done depends on the final determination of whether migration has occurred and can be done as described below in Section 16.13.4 in the case of migration or as described in Section 16.13.5 in the case of a network address transfer without migration.

Once the initial address has been determined, clients are free to apply an abbreviated process to find additional addresses trunkable with it (clients may seek session-trunkable or server-trunkable addresses depending on whether they support client ID trunking). During this later phase of the process, further location entries are examined using the abbreviated procedure specified below:

A: Before the EXCHANGE_ID, the fs name of the location entry is examined, and if it does not match that currently being used, the entry is ignored. Otherwise, one proceeds as specified by step 1 above.

B: In the case that the network address is session-trunkable with one used previously, a BIND_CONN_TO_SESSION is used to access that session using the new network address. Otherwise, or if the bind operation fails, a CREATE_SESSION is done.

C: The verification procedure referred to in step 4 above is used. However, if it fails, the entry is ignored and the next available entry is used.

16.13.4. Obtaining Access to Sessions and State after Migration

In the event that migration has occurred, migration recovery will involve determining whether Transparent State Migration has occurred. This decision is made based on the client ID returned by the EXCHANGE_ID and the reported confirmation status.

* If the client ID is an unconfirmed client ID not previously known to the client, then Transparent State Migration has not occurred.

* If the client ID is a confirmed client ID previously known to the client, then any transferred state would have been merged with an existing client ID representing the client to the destination server. In this state merger case, Transparent State Migration might or might not have occurred, and a determination as to whether it has occurred is deferred until sessions are established and the client is ready to begin state recovery.

* If the client ID is a confirmed client ID not previously known to the client, then the client can conclude that the client ID was transferred as part of Transparent State Migration. In this transferred client ID case, Transparent State Migration has occurred, although some state might have been lost.

Once the client ID has been obtained, it is necessary to obtain access to sessions to continue communication with the new server. In any of the cases in which Transparent State Migration has occurred, it is possible that a session was transferred as well. To deal with that possibility, clients can, after doing the EXCHANGE_ID, issue a BIND_CONN_TO_SESSION to connect the transferred session to a connection to the new server. If that fails, it is an indication that the session was not transferred and that a new session needs to be created to take its place.

In some situations, it is possible for a BIND_CONN_TO_SESSION to succeed without session migration having occurred. If state merger has taken place, then the associated client ID may have already had a set of existing sessions, with it being possible that the session ID of a given session is the same as one that might have been migrated. In that event, a BIND_CONN_TO_SESSION might succeed, even though there could have been no migration of the session with that session ID. In such cases, the client will receive sequence errors when the slot sequence values used are not appropriate on the new session. When this occurs, the client can create a new a session and cease using the existing one.

Once the client has determined the initial migration status, and determined that there was a shift to a new server, it needs to re-establish its locking state, if possible. To enable this to happen without loss of the guarantees normally provided by locking, the destination server needs to implement a per-fs grace period in all cases in which lock state was lost, including those in which Transparent State Migration was not implemented. Each client for which there was a transfer of locking state to the new server will have the duration of the grace period to reclaim its locks, from the time its locks were transferred.

Clients need to deal with the following cases:

* In the state merger case, it is possible that the server has not attempted Transparent State Migration, in which case state may have been lost without it being reflected in the SEQ4_STATUS bits. To determine whether this has happened, the client can use TEST_STATEID to check whether the stateids created on the source server are still accessible on the destination server. Once a single stateid is found to have been successfully transferred, the client can conclude that Transparent State Migration was begun, and any failure to transport all of the stateids will be reflected in the SEQ4_STATUS bits. Otherwise, Transparent State Migration has not occurred.

* In a case in which Transparent State Migration has not occurred, the client can use the per-fs grace period provided by the destination server to reclaim locks that were held on the source server.

* In a case in which Transparent State Migration has occurred, and no lock state was lost (as shown by SEQ4_STATUS flags), no lock reclaim is necessary.

* In a case in which Transparent State Migration has occurred, and some lock state was lost (as shown by SEQ4_STATUS flags), existing stateids need to be checked for validity using TEST_STATEID, and reclaim used to re-establish any that were not transferred.

For all of the cases above, RECLAIM_COMPLETE with an rca_one_fs value of TRUE needs to be done before normal use of the file system, including obtaining new locks for the file system. This applies even if no locks were lost and there was no need for any to be reclaimed.

16.13.5. Obtaining Access to Sessions and State after Network Address Transfer

The case in which there is a transfer to a new network address without migration is similar to that described in Section 16.13.4 above in that there is a need to obtain access to needed sessions and locking state. However, the details are simpler and will vary depending on the type of trunking between the address receiving NFS4ERR_MOVED and that to which the transfer is to be made.

To make a session available for use, a BIND_CONN_TO_SESSION should be used to obtain access to the session previously in use. Only if this fails, should a CREATE_SESSION be done. While this procedure mirrors that in Section 16.13.4 above, there is an important difference in that preservation of the session is not purely optional but depends on the type of trunking.

Access to appropriate locking state will generally need no actions beyond access to the session. However, the SEQ4_STATUS bits need to be checked for lost locking state, including the need to reclaim locks after a server reboot, since there is always a possibility of locking state being lost.

16.14. Server Responsibilities Upon Migration

In the event of file system migration, when the client connects to the destination server, that server needs to be able to provide the client continued access to the files it had open on the source server. There are two ways to provide this:

* By provision of an fs-specific grace period, allowing the client the ability to reclaim its locks, in a fashion similar to what would have been done in the case of recovery from a server restart. See Section 16.14.1 for a more complete discussion.

* By implementing Transparent State Migration possibly in connection with session migration, the server can provide the client immediate access to the state built up on the source server on the destination server.

  These features are discussed separately in Sections 16.14.2 and 16.14.3, which discuss Transparent State Migration and session migration, respectively.

All the features described above can involve transfer of lock-related information between source and destination servers. In some cases, this transfer is a necessary part of the implementation, while in other cases, it is a helpful implementation aid, which servers might

or might not use. The subsections below discuss the information that would be transferred but do not define the specifics of the transfer protocol. This is left as an implementation choice, although standards in this area could be developed at a later time.

16.14.1. Server Responsibilities in Effecting State Reclaim after Migration

In this case, the destination server needs no knowledge of the locks held on the source server. It relies on the clients to accurately report (via reclaim operations) the locks previously held, and does not allow new locks to be granted on migrated file systems until the grace period expires. Disallowing of new locks applies to all clients accessing these file systems, while grace period expiration occurs for each migrated client independently.

During this grace period, clients have the opportunity to use reclaim operations to obtain locks for file system objects within the migrated file system, in the same way that they do when recovering from server restart, and the servers typically rely on clients to accurately report their locks, although they have the option of subjecting these requests to verification. If the clients only reclaim locks held on the source server, no conflict can arise. Once the client has reclaimed its locks, it indicates the completion of lock reclamation by performing a RECLAIM_COMPLETE specifying rca_one_fs as TRUE.

While it is not necessary for source and destination servers to cooperate to transfer information about locks, implementations are well advised to consider transferring the following useful information:

* If information about the set of clients that have locking state for the transferred file system is made available, the destination server will be able to terminate the grace period once all such clients have reclaimed their locks, allowing normal locking activity to resume earlier than it would have otherwise.

* Locking summary information for individual clients (at various possible levels of detail) can detect some instances in which clients do not accurately represent the locks held on the source server.

16.14.2. Server Responsibilities in Effecting Transparent State Migration

The basic responsibility of the source server in effecting Transparent State Migration is to make available to the destination server a description of each piece of locking state associated with the file system being migrated. In addition to client id string and verifier, the source server needs to provide for each stateid:

* The stateid including the current sequence value.

* The associated client ID.

* The handle of the associated file.

* The type of the lock, such as open, byte-range lock, delegation, or layout.

* For locks such as opens and byte-range locks, there will be information about the owner(s) of the lock.

* For recallable/revocable lock types, the current recall status needs to be included.

* For each lock type, there will be associated type-specific information. For opens, this will include share and deny mode while for byte-range locks and layouts, there will be a type and a byte-range.

Such information will most probably be organized by client id string on the destination server so that it can be used to provide appropriate context to each client when it makes itself known to the client. Issues connected with a client impersonating another by presenting another client's client id string can be addressed using NFSv4.1 state protection features, as described in Section 26.

A further server responsibility concerns locks that are revoked or otherwise lost during the process of file system migration. Because locks that appear to be lost during the process of migration will be reclaimed by the client, the servers have to take steps to ensure that locks revoked soon before or soon after migration are not inadvertently allowed to be reclaimed in situations in which the continuity of lock possession cannot be assured.

* For locks lost on the source but whose loss has not yet been acknowledged by the client (by using FREE_STATEID), the destination must be aware of this loss so that it can deny a request to reclaim them.

* For locks lost on the destination after the state transfer but before the client's RECLAIM_COMPLETE is done, the destination server should note these and not allow them to be reclaimed.

An additional responsibility of the cooperating servers concerns situations in which a stateid cannot be transferred transparently because it conflicts with an existing stateid held by the client and associated with a different file system. In this case, there are two valid choices:

* Treat the transfer, as in NFSv4.0, as one without Transparent State Migration. In this case, conflicting locks cannot be granted until the client does a RECLAIM_COMPLETE, after reclaiming the locks it had, with the exception of reclaims denied because they were attempts to reclaim locks that had been lost.

* Implement Transparent State Migration, except for the lock with the conflicting stateid. In this case, the client will be aware of a lost lock (through the SEQ4_STATUS flags) and be allowed to reclaim it.

When transferring state between the source and destination, the issues discussed in Section 7.2 of [RFC7931] must still be attended to. In this case, the use of NFS4ERR_DELAY may still be necessary in NFSv4.1, as it was in NFSv4.0, to prevent locking state changing while it is being transferred. See Section 20.1.1.3 for information about appropriate client retry approaches in the event that NFS4ERR_DELAY is returned.

There are a number of important differences in the NFS4.1 context:

* The absence of RELEASE_LOCKOWNER means that the one case in which an operation could not be deferred by use of NFS4ERR_DELAY no longer exists.

* Sequencing of operations is no longer done using owner-based operation sequences numbers. Instead, sequencing is session-based.

As a result, when sessions are not transferred, the techniques discussed in Section 7.2 of [RFC7931] are adequate and will not be further discussed.

16.14.3. Server Responsibilities in Effecting Session Transfer

The basic responsibility of the source server in effecting session transfer is to make available to the destination server a description of the current state of each slot with the session, including the following:

* The last sequence value received for that slot.

* Whether there is cached reply data for the last request executed and, if so, the cached reply.

When sessions are transferred, there are a number of issues that pose challenges in terms of making the transferred state unmodifiable during the period it is gathered up and transferred to the destination server:

* A single session may be used to access multiple file systems, not all of which are being transferred.

* Requests made on a session may, even if rejected, affect the state of the session by advancing the sequence number associated with the slot used.

As a result, when the file system state might otherwise be considered unmodifiable, the client might have any number of in-flight requests, each of which is capable of changing session state, which may be of a number of types:

1. Those requests that were processed on the migrating file system before migration began.

2. Those requests that received the error NFS4ERR_DELAY because the file system being accessed was in the process of being migrated.

3. Those requests that received the error NFS4ERR_MOVED because the file system being accessed had been migrated.

4. Those requests that accessed the migrating file system in order to obtain location or status information.

5. Those requests that did not reference the migrating file system.

It should be noted that the history of any particular slot is likely to include a number of these request classes. In the case in which a session that is migrated is used by file systems other than the one migrated, requests of class 5 may be common and may be the last request processed for many slots.

Since session state can change even after the locking state has been fixed as part of the migration process, the session state known to the client could be different from that on the destination server, which necessarily reflects the session state on the source server at an earlier time. In deciding how to deal with this situation, it is helpful to distinguish between two sorts of behavioral consequences of the choice of initial sequence ID values:

* The error NFS4ERR_SEQ_MISORDERED is returned when the sequence ID in a request is neither equal to the last one seen for the current slot nor the next greater one.

  In view of the difficulty of arriving at a mutually acceptable value for the correct last sequence value at the point of migration, it may be necessary for the server to show some degree of forbearance when the sequence ID is one that would be considered unacceptable if session migration were not involved.

* Returning the cached reply for a previously executed request when the sequence ID in the request matches the last value recorded for the slot.

  In the cases in which an error is returned and there is no possibility of any non-idempotent operation having been executed, it may not be necessary to adhere to this as strictly as might be proper if session migration were not involved. For example, the fact that the error NFS4ERR_DELAY was returned may not assist the client in any material way, while the fact that NFS4ERR_MOVED was returned by the source server may not be relevant when the request was reissued and directed to the destination server.

An important issue is that the specification needs to take note of all potential COMPOUNDs, even if they might be unlikely in practice. For example, a COMPOUND is allowed to access multiple file systems and might perform non-idempotent operations in some of them before accessing a file system being migrated. Also, a COMPOUND may return considerable data in the response before being rejected with NFS4ERR_DELAY or NFS4ERR_MOVED, and may in addition be marked as sa_cachethis. However, note that if the client and server adhere to rules in Section 20.1.1.3, there is no possibility of non-idempotent operations being spuriously reissued after receiving NFS4ERR_DELAY response.

To address these issues, a destination server MAY do any of the following when implementing session transfer:

* Avoid enforcing any sequencing semantics for a particular slot until the client has established the starting sequence for that slot on the destination server.

* For each slot, avoid returning a cached reply returning NFS4ERR_DELAY or NFS4ERR_MOVED until the client has established the starting sequence for that slot on the destination server.

* Until the client has established the starting sequence for a particular slot on the destination server, avoid reporting NFS4ERR_SEQ_MISORDERED or returning a cached reply that contains either NFS4ERR_DELAY or NFS4ERR_MOVED and consists solely of a series of operations where the response is NFS4_OK until the final error.

Because of the considerations mentioned above, including the rules for the handling of NFS4ERR_DELAY included in Section 20.1.1.3, the destination server can respond appropriately to SEQUENCE operations received from the client by adopting the three policies listed below:

* Not responding with NFS4ERR_SEQ_MISORDERED for the initial request on a slot within a transferred session because the destination server cannot be aware of requests made by the client after the server handoff but before the client became aware of the shift. In cases in which NFS4ERR_SEQ_MISORDERED would normally have been reported, the request is to be processed normally as a new request.

* Replying as it would for a retry whenever the sequence matches that transferred by the source server, even though this would not provide retry handling for requests issued after the server handoff, under the assumption that, when such requests are issued, they will never be responded to in a state-changing fashion, making retry support for them unnecessary.

* Once a non-retry SEQUENCE is received for a given slot, using that as the basis for further sequence checking, with no further reference to the sequence value transferred by the source server.

16.15. 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 look up, but will have cached information regarding the upper levels of the name hierarchy. However, these examples are chosen to make the required behavior clear and easy to put within the scope of a small number of requests, without getting into a discussion of the details of how specific clients might choose to cache things.

16.15.1. 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 that the file system has moved, or it may be that the target server is functioning mainly, or solely, to refer clients to the servers on which various file systems are located.

* PUTROOTFH
* LOOKUP "this"
* LOOKUP "is"
* LOOKUP "the"
* LOOKUP "path"
* GETFH
* GETATTR (fsid, fileid, size, time_modify)

Under the given circumstances, the following will be the result.

* PUTROOTFH --> NFS_OK. The current fh is now the root of the pseudo-fs.
* LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.
* LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.

* LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.

* 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.

* GETFH --> NFS4ERR_MOVED. Fails because current fh is in an absent file system at the start of the operation, and the specification makes no exception for GETFH.

* 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 that in this example, the client did not obtain filehandles and attribute information (e.g., fsid) for the intermediate directories, so that it would not be sure where the absent file system starts. It could be the case, for example, that /this/is/the is the root of the moved file system and that the reason that the look up of "path" succeeded is that the file system was not absent on that operation but was moved between the last LOOKUP and the GETFH (since COMPOUND is not atomic). Even if we had the fsids for all of the intermediate directories, we could have no way of knowing that /this/is/the/path was the root of a new file system, since we don't yet have its fsid.

In order to get the necessary information, let us re-send the chain of LOOKUPs with GETFHs and GETATTRs to at least get the fsids so we can be sure where the appropriate file system boundaries are. The client could choose to get fs_locations_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 NFS4ERR_MOVED was, and was not, 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 other reliable source information on the boundary of the file system that is moved. If, for example, the file system /this/is had moved, we would have a case of migration rather than referral, and once the boundaries of the migrated file system was clear we could fetch fs_locations_info.

* We are fetching fs_locations_info because the fact that we got an NFS4ERR_MOVED at this point means that it is most likely that this is a referral and we need the destination. Even if it is the case that /this/is/the is a file system that has migrated, we will still need the location information for that file system.

OP14: GETFH --> NFS4ERR_MOVED

* Fails because current fh is in an absent file system at the start of the operation, and the specification makes no exception for GETFH. Note that this means the server will never send the client a filehandle from within an absent file system.

Given the above, the client knows where the root of the absent file system is (/this/is/the/path) by noting where the change of fsid occurred (between "the" and "path"). The fs_locations_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 transient information with the function of enabling the referral.

16.15.2. Referral Example (READDIR)

Another context in which a client may encounter referrals is when it does a READDIR on a directory in which some of the sub-directories are the roots of absent file systems.

Suppose such a directory is read as follows:

* PUTROOTFH
* LOOKUP "this"
* LOOKUP "is"
* LOOKUP "the"
* 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 the attributes cannot be provided, the result will be an NFS4ERR_MOVED error on the READDIR, with the detailed results as follows:

* PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.
* LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.
* LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.
* LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.
* READDIR (fsid, size, time_modify, mounted_on_fileid) --> NFS4ERR_MOVED. Note that the same error would have been returned if /this/is/the had migrated, but it is returned because the directory contains the root of an absent file system.

So now suppose that we re-send with rdattr_error:

* PUTROOTFH
* LOOKUP "this"
* LOOKUP "is"

* LOOKUP "the"

* READDIR (rdattr_error, fsid, size, time_modify, mounted_on_fileid)

The results will be:

* PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.

* LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.

* LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.

* LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.

* 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.

Suppose we do another READDIR to get fs_locations_info (although we could have used a GETATTR directly, as in Section 16.15.1).

* PUTROOTFH

* LOOKUP "this"

* LOOKUP "is"

* LOOKUP "the"

* READDIR (rdattr_error, fs_locations_info, mounted_on_fileid, fsid, size, time_modify)

The results would be:

* PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.

* LOOKUP "this" --> NFS_OK. The current fh is for /this and is within the pseudo-fs.

* LOOKUP "is" --> NFS_OK. The current fh is for /this/is and is within the pseudo-fs.

* LOOKUP "the" --> NFS_OK. The current fh is for /this/is/the and is within the pseudo-fs.

* 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:

* rdattr_error (value: NFS_OK)

* fs_locations_info

* mounted_on_fileid (value: unique fileid within referring file system)

* 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.

16.16. The Attribute fs_locations

The fs_locations attribute is structured in the following way:

```
struct fs_location4 {
        utf8str_mixed   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, IPv6 address, or a zero-length string. An IPv4 or IPv6 address is represented as a universal address (see Section 9.3.9 and [RFC5665]), 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 the fs_locations attribute each contain an array of such locations. Since the namespace of each server may be constructed differently, the "fs_root" field is provided. The path represented by fs_root represents the location of the file system in the current server's namespace, i.e., that of the server from which the fs_locations attribute was obtained. The fs_root path is meant to aid the client by clearly referencing the root of the file system whose locations are being reported, no matter what object within the current file system the current filehandle designates. The fs_root is simply the pathname the client used to reach the object on the current server (i.e., the object to which the fs_locations attribute applies).

When the fs_locations attribute is interrogated and there are no 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 an fs_root value of /a/b/c and two entries in fs_locations. One entry in fs_locations will be for itself (servA) and the other will be for servB with a path of /x/y/z. With this information, the client is able to substitute /x/y/z for the /a/b/c at the beginning of its access path and construct /x/y/z/d to use for the new server.

Note that there is no requirement that the number of components in each rootpath be the same; there is no relation between the number of components in rootpath or fs_root, and none of the components in a rootpath and fs_root have to be the same. In the above example, we could have had a third element in the locations array, with server equal to "servC" and rootpath equal to "/I/II", and a fourth element in locations with server equal to "servD" and rootpath equal to "/aleph/beth/gimel/daleth/he".

The relationship between fs_root to a rootpath is that the client replaces the pathname indicated in fs_root for the current server for the substitute indicated in rootpath for the new server.

For an example of a referred or migrated file system, suppose there is a file system located at serv1. At serv1, the file system is located at /az/buky/vedi/glagoli. The client finds that object at glagoli has migrated (or is a referral). The client gets the fs_locations attribute, which contains an fs_root of /az/buky/vedi/glagoli, and one element in the locations array, with server equal to serv2, and rootpath equal to /izhitsa/fita. The client replaces /az/buky/vedi/glagoli with /izhitsa/fita, and uses the latter pathname on serv2.

Thus, the server MUST return an fs_root that is equal to the path the client used to reach the object to which the fs_locations attribute applies. Otherwise, the client cannot determine the new path to use on the new server.

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.

* 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.

* 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.

* 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:

* When the transition is due to migration, that is, the client was directed to a 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.

* When the transition is due to failover to another replica, that is, the client selected another replica without receiving an 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 26 for a discussion on the recommendations for the security flavor to be used by any GETATTR operation that requests the fs_locations attribute.

16.17. The Attribute fs_locations_info

The fs_locations_info attribute is intended as a more functional replacement for the fs_locations attribute, which will continue to exist and be supported. Clients can use it to get a more complete set of data about alternative file system locations, including additional network paths to access replicas in use and additional replicas. When the server does not support fs_locations_info, fs_locations can be used to get a subset of the data. A server that supports fs_locations_info MUST support fs_locations as well.

There is additional data present in fs_locations_info that is not available in fs_locations:

* Attribute continuity information. This information will allow a client to select a replica that meets the transparency requirements of the applications accessing the data and to

leverage optimizations due to the server guarantees of attribute continuity (e.g., if the change attribute of a file of the file system is continuous between multiple replicas, the client does not have to invalidate the file's cache when switching to a different replica).

* File system identity information that 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 distinct synchronized replicas of the same file data.

  Note that having two replicas with common identity information is distinct from the case of two (trunked) paths to the same replica.

* Information that will bear on the suitability of various replicas, depending on the use that the client intends. For example, many applications need an absolutely up-to-date copy (e.g., those that write), while others may only need access to the most up-to-date copy reasonably available.

* Server-derived preference information for replicas, which can be used to implement load-balancing while giving the client the entire 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 rootpath on their respective servers. The lower-level structure in turn (fs_locations_item4) contains a specific pathname and information on one or more individual network access paths. For that last, lowest level, fs_locations_info has an fs_locations_server4 structure that contains per-server-replica information in addition to the file system location entry. This per-server-replica information includes a nominally opaque array, fls_info, within which specific pieces of information are located at the specific indices listed below.

Two fs_location_server4 entries that are within different fs_location_item4 structures are never trunkable, while two entries within in the same fs_location_item4 structure might or might not be trunkable. Two entries that are trunkable will have identical identity information, although, as noted above, the converse is not the case.

The attribute will always contain at least a single fs_locations_server entry. Typically, there 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 known to other servers. Further, compatibility attributes may differ. Filehandles might be of the same class going from replica A to replica B but not 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 access path
 */
struct  fs_locations_server4 {
        int32_t         fls_currency;
        opaque          fls_info<>;
        utf8str_mixed   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_CLWRITEVER        = 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 rootpath
    * within 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. 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.

16.17.1. The fs_locations_server4 Structure

The fs_locations_server4 structure consists of the following items in addition to the fls_server field, which specifies a network address or set of addresses to be used to access the specified file system. Note that both of these items (i.e., fls_currency and fls_info) specify attributes of the file system replica and should not be different when there are multiple fs_locations_server4 structures, each specifying a network path to the chosen replica, for the same replica.

When these values are different in two fs_locations_server4 structures, a client has no basis for choosing one over the other and is best off simply ignoring both entries, whether these entries apply to migration replication or referral. When there are more than two such entries, majority voting can be used to exclude a single erroneous entry from consideration. In the case in which trunking information is provided for a replica currently being accessed, the additional trunked addresses can be ignored while access continues on the address currently being used, even if the entry corresponding to that path might be considered invalid.

* An indication of how up-to-date the file system is (fls_currency) in seconds. 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 value of zero indicates that the 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 serves as a hint about how far the copy of the data would be expected to be behind the most up-to-date copy.

* 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.

* 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 16.16).

With the exception of the transport-flag field (at offset FSLI4BX_TFLAGS with the fls_info array), all of this data defined in this specification applies to the replica specified by the entry, rather than the specific network path used to access it. The classification of data in extensions to this data is discussed below.

Data within the fls_info array is in the form of 8-bit data items with constants giving the offsets within the array of various values describing this particular file system instance. This style of definition was chosen, in preference to explicit XDR structure definitions for these values, for a number of reasons.

* The kinds of data in the fls_info array, representing flags, file system classes, and priorities among sets of file systems representing the same data, are such that 8 bits provide a quite acceptable range of values. Even where there might be more than 256 such file system instances, having more than 256 distinct classes or priorities is unlikely.

* Explicit definition of the various specific data items within XDR would limit expandability in that any extension within would require yet another attribute, leading to specification and implementation clumsiness. In the context of the NFSv4 extension model in effect at the time fs_locations_info was designed (i.e., that which is described in [RFC5661]), this would necessitate a new minor version to effect any Standards Track extension to the data in fls_info.

The set of fls_info data is subject to expansion in a future minor version or in a Standards 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 or flag bits that are not defined in Standards Track RFCs.

In light of the new extension model defined in [RFC8178] and the fact that the individual items within fls_info are not explicitly referenced in the XDR, the following practices should be followed when extending or otherwise changing the structure of the data returned in fls_info within the scope of a single minor version:

* All extensions need to be described by Standards Track documents. There is no need for such documents to be marked as updating [RFC5661], [RFC8881], or this document.

* It needs to be made clear whether the information in any added data items applies to the replica specified by the entry or to the specific network paths specified in the entry.

* There needs to be a reliable way defined to determine whether the server is aware of the extension. This may be based on the length field of the fls_info array, but it is more flexible to provide fs-scope or server-scope attributes to indicate what extensions are provided.

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 RFCs, 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 will be constrained by the need for the relevant quantities not to cross XDR word boundaries.

The fls_info array currently contains:

* Two 8-bit flag fields, one devoted to general file-system characteristics and a second reserved for transport-related capabilities.

* Six 8-bit class values that define various file system equivalence classes as explained below.

* Four 8-bit priority values that 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:

* FSLI4GF_WRITABLE indicates that this file system target is writable, allowing it to be selected by clients that 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 16.11.6, 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 a read-only file system can cause problems for clients that select a migration or replication target based on the flag and then find themselves unable to write.

* FSLI4GF_CUR_REQ indicates that this replica is the one on which the request is being made. Only a single server entry may have this flag set and, in the case of a referral, no entry will have it set. Note that this flag might be set even if the request was made on a network access path different from any of those specified in the current entry.

* FSLI4GF_ABSENT indicates that this entry corresponds to 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 when a file system has been accessed at this location and has subsequently been migrated.

* FSLI4GF_GOING indicates that a replica, while still available, should not be used further. The client, if using it, should make an orderly transfer to another file system instance as expeditiously as possible. It is expected that file systems going out of service will be announced as FSLI4GF_GOING some time before the actual loss of service. It is also expected that the fli_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 avoid being referred to this instance whenever there is another choice.

This flag, like the other items within fls_info, applies to the replica rather than to a particular path to that replica. When it appears, a transition to a new replica, rather than to a different path to the same replica, is indicated.

* 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 to 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 actions 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 network path(s) specified by the entry:

* FSLI4TF_RDMA indicates that any specified network paths provide NFSv4.1 clients 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 Sections 16.9 through 16.14 and their various subsections 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 that 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 incrementing the value of the last class value assigned for that index.

* The field with byte index FSLI4BX_CLSIMUL defines the simultaneous-use class for the file system.

* The field with byte index FSLI4BX_CLHANDLE defines the handle class for the file system.

* The field with byte index FSLI4BX_CLFILEID defines the fileid class for the file system.

* The field with byte index FSLI4BX_CLWRITEVER defines the write-verifier class for the file system.

* The field with byte index FSLI4BX_CLCHANGE defines the change class for the file system.

* 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 should be conservative about using this mechanism, particularly when the environment is one in which client communication characteristics are neither tightly controlled nor 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.

* The field at byte index FSLI4BX_READRANK gives the rank value to be used for read-only access.

* The field at byte index FSLI4BX_READORDER gives the order value to be used for read-only access.

* The field at byte index FSLI4BX_WRITERANK gives the rank value to be used for writable access.

* 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.

16.17.2. The fs_locations_info4 Structure

The fs_locations_info4 structure, encoding the fs_locations_info attribute, contains the following:

* The fli_flags field, which contains general flags that affect the interpretation of this fs_locations_info4 structure and all fs_locations_item4 structures within it. The only flag currently defined is FSLI4IF_VAR_SUB. All bits in the fli_flags field that are not defined should always be returned as zero.

* The fli_fs_root field, which contains the pathname of the root of the current file system on the current server, just as it does in the fs_locations4 structure.

* An array called fli_items of fs_locations4_item structures, which contain information about replicas of the current file system. Where the current file system is actually present, or has been present, i.e., this is not a referral situation, one of the fs_locations_item4 structures will contain an fs_locations_server4 for the current server. This structure will have FSLI4GF_ABSENT set if the current file system is absent, i.e., normal access to it will return NFS4ERR_MOVED.

* 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 an 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 that 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. It is to be expected that this may have a different fli_valid_for value, which the client should then use in the same fashion as the previous value. Because a refetch of the attribute causes information from all component entries to be refetched, the server will typically provide a low value for this field if any of the replicas are likely to go out of service in a short time frame. Note that, because of the ability of the server to return NFS4ERR_MOVED to trigger the use of different paths, when alternate trunked paths are available, there is generally no need to use low values of fli_valid_for in connection with the management of alternate paths to the same replica.

The FSLI4IF_VAR_SUB flag within fli_flags controls whether variable substitution is to be enabled. See Section 16.17.3 for an explanation of variable substitution.

16.17.3. The fs_locations_item4 Structure

The fs_locations_item4 structure contains a pathname (in the field fli_rootpath) that 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, and the part after is to be a case-insensitive alphanumeric string.

Where the domain is "ietf.org", only variable names defined in this document or subsequent Standards Track RFCs 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 cases 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 that 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 APIs, 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 variables could result in the 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 or not the FSLI4IF_VAR_SUB flag within the associated fs_locations_info4 structure is set.

16.18. 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:

* 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.

* 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.

* STATUS4_UPDATED, which indicates an image that cannot be updated by the user writing to it but that 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.

* 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.

* STATUS4_REFERRAL, which indicates that the file system in 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 is made to the data, and MUST be revoked if this is not possible. Similarly, if an OPEN is inconsistent with data that is changed (the OPEN has OPEN4_SHARE_DENY_WRITE/OPEN4_SHARE_DENY_BOTH and the data is changed), that OPEN 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 the 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 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, who created it, etc. Making this information available in this attribute in a human-readable string will be helpful for applications and system administrators and will also serve 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 includes 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.

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 16.17) 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 fss_version that 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. This is to avoid the possibility that the fs_status that the client holds will be one for an earlier image, which would cause the client to accept a new file system instance that is later than that but still earlier than the updated data read by the client.

In order to accept valid images reliably, the client 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 the partial order) and using the last. The client may then, when switching among file system instances, decline to use an instance that 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.

17. Parallel NFS (pNFS)

17.1. Introduction

pNFS is an OPTIONAL feature within NFSv4.1; the pNFS feature set allows direct access file data’s location including the possibility of unmediated 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 (compared to the server’s throughput capability), this can provide significantly better file access performance. The relationship among multiple clients, a single server, and multiple file storage devices for pNFS (some of which act as data servers, by providing support for file access protocols). is shown in Figure 1.

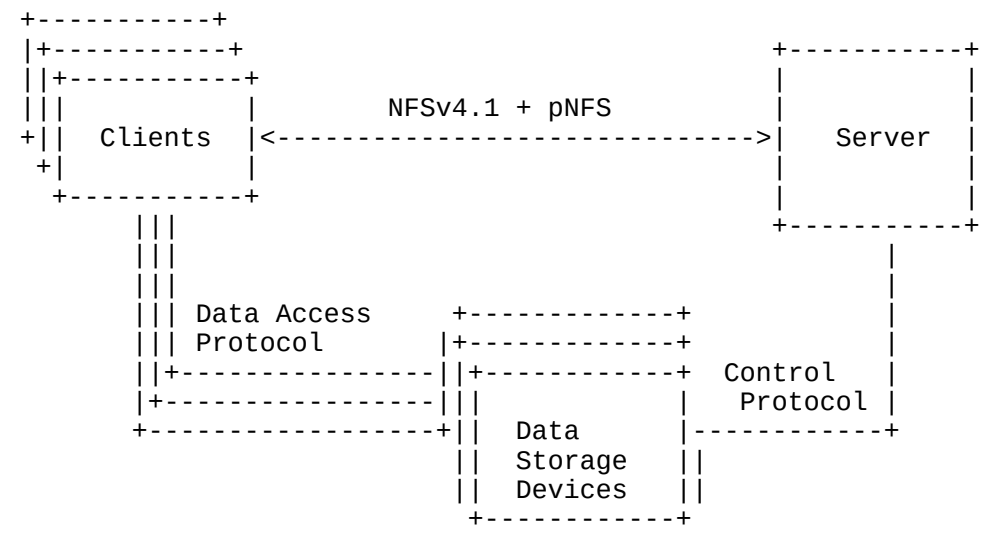


Figure 1

In this model, the clients, the metadata server, and data storage devices work together to provide file data access and to deny it to those not appropriately authorized. This is in contrast to NFSv4 without pNFS, where this is primarily the server's responsibility while some of this responsibility may be delegated to the client under strictly specified conditions. See Section 17.2.5 for a discussion of the Data Access Protocols. See Section 17.2.6 for a discussion of Control Protocols.

pNFS involves OPTIONAL operations that manage protocol objects called 'layouts' (Section 17.2.7) that contain a byte-range and location information. Layouts are managed in a fashion similar to 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 designated 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 17.5.5. Byte-range locking issues are discussed in Sections 17.2.9 and 17.5.1.

17.2. 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 that 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 17.2.1. The metadata functionality is implemented by an NFSv4.1 server that supports pNFS and the operations described in Section 23; such a server is called a metadata server (Section 17.2.2).

The data functionality is implemented by one or more data storage devices, each of which are accessed by the client via a file access protocol. A subset (defined in Section 18.6) of NFSv4.1 is one such protocol. New terms are introduced to the NFSv4.1 nomenclature and existing terms are clarified to allow for the description of the pNFS feature.

17.2.1. 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.

17.2.2. Metadata Server

An NFSv4.1 server that supports the pNFS feature. A variety of architectural choices exist 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 together with some portion of file data.

17.2.3. pNFS Client

An NFSv4.1 client that supports pNFS operations and supports at least one data access protocol for performing I/O to data storage devices.

17.2.4. Data Storage Devices

A data storage device stores a regular file's data, but while metadata management is done by the metadata server. A data storage device could be another NFSv4.1 server, an object-based storage device (OSD), a block device accessed over a System Area Network (SAN, e.g., either FiberChannel or iSCSI SAN), or some other entity.

17.2.5. Data Access Protocol

As noted in Figure 1, the data access protocol is provided as a method used by the client to store and retrieve located on data storage devices.

The NFSv4.1 pNFS feature has been structured to allow a variety of data access protocols to be defined and used. The one actually used depends on the type of layout (see Section 17.2.7)

One possible data access protocol is NFSv4.1 itself (as documented in Section 18). Other options for the data access protocol are described elsewhere and include:

* Block/volume protocols such as Internet SCSI (iSCSI) [RFC7143] and FCP [FCP-2]. 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. See [RFC5663] for a layout specification that allows pNFS to use block/volume storage protocols.

* Object protocols such as OSD over iSCSI or Fibre Channel [OSD-T10]. See [RFC5664] for a layout specification that allows pNFS to use object storage protocols.

* Other NFS versions as described in [RFC8435].

It is possible that multiple data access protocols are available to both client and server and it may be possible that a client and server do not have a matching data access protocol available to them. In this case, there is no way for the client to obtain a layout. Whether a layout is available or not, the pNFS server MUST support normal NFSv4.1 access to any file accessible by the pNFS feature. As a result, there is continued interoperability between an NFSv4.1 client and server, regardless of whether that feature is usable in any particular case.

17.2.6. Control Protocol

As illustrated in Figure 1, a control protocol is used between the metadata server and file data devicess. 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, the management of state required by the data storage devices to perform client access control, and, depending on the file access protocol, the enforcement of authentication and authorization so that restrictions that would be enforced by the metadata server could also be enforced by the data storafe devices.

A particular control protocol is not REQUIRED by NFSv4.1 but requirements are placed on the control protocol for maintaining attributes such as modify time, the change attribute, and the end-of-file (EOF) position. Note that if pNFS is layered over a clustered, parallel file system (e.g., PVFS [PVFS]), the mechanisms that enable clustering and parallelism in that file system serve the same role as a control protocol. The differences in communication approaches allow these mechanisms to be treated as if they were structured as a control protocol.

In the flexible files layout [RFC8435], it is often stated that, when used in the "loose" coupling mode, there is no control protocol, which is at variance with the way control protocols are treated in this document, which requires that certain activities described be done and considers the means that are used to perform them as constituting a control protocol. In fact, these activities are performed using the same base protocol as the data access protocol, albeit in a different mode with greater privileges. In this document, we describe such arrangements as having no "separate control protocol.

17.2.7. Layout Types

A layout describes the mapping of a file’s data to the file data devuices that hold the data. A layout is said to belong to a specific layout type (data type layouttype4, see Section 9.3.13). The layout type allows for variants to handle different data access protocols, such as those associated with block/volume [RFC5663], object [RFC5664], and file (Section 18) layout types. A metadata server MUST support at least one layout type. A private sub-range of the layout type namespace is also defined. Values from the private layout type range MAY be used for internal testing or experimentation (see Section 9.3.13).

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 a 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.

17.2.8. Layout

A layout defines how a file’s data is organized on one or more data storage devices. There are many potential layout types; each of the layout types is differentiated by the data access protocol used to access data and by the aggregation scheme that lays out the file data on the various data storage devices. A layout is precisely identified by the 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.

17.2.9. Layout Iomode

The layout iomode (data type layoutiomode4, see Section 9.3.20) indicates to the metadata server the client's intent to perform either just READ operations or a mixture 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 23.43). For example, for block/volume-based protocols, block allocation could occur when a LAYOUTIOMODE4_RW 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 LAYOUTIOMODE4_READ and LAYOUTIOMODE4_RW iomodes are being returned or recalled, respectively.

A data storage device may validate I/O for consistency with the iomode. Whether this happens depends on the layout type. In this case, if the client's layout iomode is inconsistent with the I/O being performed, the data storage device may reject the client's I/O with an error indicating that a new layout with the correct iomode should be obtained via LAYOUTGET. For example, if a client gets a layout with a LAYOUTIOMODE4_READ iomode and performs a WRITE to a data storage device, the device is allowed to reject that WRITE.

The use of the layout iomode does not conflict with OPEN share modes or byte-range LOCK operations; open share mode and byte-range lock conflicts are enforced as they are without the use of pNFS and are logically separate from issues related to pNFS layouts. Open share modes and byte-range locks are the preferred method for restricting user access to data files. For example, an OPEN of OPEN4_SHARE_ACCESS_WRITE does not conflict with a LAYOUTGET containing an iomode of LAYOUTIOMODE4_RW performed by another client. Applications that depend on writing into the same file concurrently may use byte-range locking to serialize their accesses.

17.2.10. Device IDs

The device ID (data type deviceid4, see Section 9.3.14) 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 23.40) 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-based 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 17.7.4 for a description of how recovery works in that situation.

A device ID is established by referencing it in the result of a GETDEVICELIST or LAYOUTGET operation and can be deleted by the server as soon as there are no layouts referring to the device ID.

If the client requested notifications for device ID mappings, the server SHOULD send CB_NOTIFY_DEVICEID notifications for device ID deletions or changes to the device-ID-to-device-address mappings to any client which has used the device-ID in question at least once, irrespective of whether the client has any layouts currently referring to it. If the server does not support or the client does not request notifications for device ID mappings, the client SHOULD periodically retired unused device IDs.

Given that GETDEVICELIST does not support requesting notifications a server that implements GETDEVICELIST MUST NOT advertise support for NOTIFY_DEVICEID4_CHANGE notification in GETDEVICEINFO, and client using GETDEVICELIST can not rely on NOTIFY_DEVICEID4_CHANGE or NOTIFY_DEVICEID4_DELETE notifications to work reliably.

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 the 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 25.12), 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, see Section 25.12.

17.3. 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 22.

These are the fore channel pNFS operations:

GETDEVICEINFO (Section 23.40), as noted previously (Section 17.2.10), returns the mapping of device ID to storage device address.

GETDEVICELIST (Section 23.41) allows clients to fetch all device IDs for a specific file system.

LAYOUTGET (Section 23.43) is used by a client to get a layout for a file.

LAYOUTCOMMIT (Section 23.42) is used to inform the metadata server of the client's intent to commit data that has been written to the storage device (the storage device as originally indicated in the return value of LAYOUTGET).

LAYOUTRETURN (Section 23.44) is used to return layouts for a file, a file system ID (FSID), or a client ID.

These are the backchannel pNFS operations:

CB_LAYOUTRECALL (Section 25.3) recalls a layout, all layouts belonging to a file system, or all layouts belonging to a client ID.

CB_RECALL_ANY (Section 25.6) tells a client that it needs to return some number of recallable objects, including layouts, to the metadata server.

CB_RECALLABLE_OBJ_AVAIL (Section 25.7) 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 25.12) notifies the client of changes to device IDs.

17.4. pNFS Attributes

A number of attributes specific to pNFS are listed and described in Section 11.16.

17.5. Layout Semantics

17.5.1. Guarantees Provided by Layouts

Layouts grant to the client the ability to access data located at a data storage device using the associated data access 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 (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 file using the data storage device 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 data storage device, a layout must be held by the client. If a device receives an I/O request 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 data access 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 data access protocol in use, device-specific 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 [OSD-T10]. 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 OPEN, LOCK, and ACCESS operations as described above. The degree to which it is possible for the client to circumvent these 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 byte-range 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 byte-range locks. For example, if one client obtains a mandatory byte-range lock and a second client accesses the on the data storage device, the device MUST appropriately restrict I/O for the range of the mandatory byte-range lock. If the device is incapable of providing this check in the presence of mandatory byte-range locks, then the metadata server MUST NOT grant potentially overlapping layouts and mandatory byte-range locks simultaneously.

17.5.2. Getting a Layout

A client obtains a layout using 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 23.43.3. As needed, a client may send multiple LAYOUTGET operations. These might result in multiple overlapping, non-conflicting layouts (see Section 17.2.8).

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 an open stateid, 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 17.5.3) on future invocations of LAYOUTGET on the file, and the "seqid" MUST NOT be set to zero. The client MUST serialize LAYOUTGET operations using a non-layout stateid with any other operation affecting the layout state on the file, including CB_LAYOUTRECALL, to allow consistent initialization of the layout state. 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 attribute (Section 11.16.4) 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 17 in Section 23.16.3 summarizes how a client is allowed to send an exclusive create.

17.5.3. Layout Stateid

As with all other stateids, the layout stateid consists of a "seqid" and "other" field. Once a layout stateid is established, 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 that 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 17.5.5.2.

The client's receipt of a "seqid" is not sufficient for it to be passed back to the metdata server. The client needs to fully process the operation in which the new seqid is seen before using it in further communication with the metadata server. For LAYOUTGET results, if the client is recording details of layout tanges received (See (Section 17.5.3.1 for information about specifics), it MUST first update its record of what ranges of the file's layout it has before using the seqid in this way. For LAYOUTRETURN results, the client MUST eliminate 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 in a message to the metadata server. The fundamental basis of these requirements regarding client processing is that the "seqid" is used to define 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 their 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.

A client MAY always forget its layout state and associated layout stateid at any time (See also Section 17.5.3.1). When this happens, the client MUST use a non-layout stateid on a subsequent LAYOUTGET operation. Doing so will inform the server that the client has no more layouts of that layout type on the file and that its respective layout state can be released before issuing a new layout in response to LAYOUTGET.

17.5.3.1. Requirements Regerding Retention of Layout Information.

It might reasonably be assumed that pNFS client state (layout ranges and iomode) for a file would exactly matches that of the pNFS metadata server for that file. This assumption would imply that any callback of a layout results in a LAYOUTRETURN or set of LAYOUTRETURNs that exactly match the range specfied in the callback, since the client and metadatta server would necessarily agree about the details of the state being maintained.

It is important to understand that the above assumption is not a protocol requirement and that the prototol does allow the the client and metadata server to drop much of this information, in order to limit the need for additional storage and internode communication that mainaining the abovementioned assumption would require.

The obligations of the client and metadata server regarding the retention of layout information are discussed below. It is important to understand that data servers often need to be aware of the details of layouts gotten and that individual layout types might impose requirements regarding their retention by the the client and how they might be used by the specifc storge protool used by the layout type.

The relevant protocol requirements discussed here are limited to those necessary to effect cancellation of layouts to prevent multiple clients using conflicting layouts and to prevent use of layouts when changes initiated by the metdata server make their further use incorrect or otherwise inadvisable. The relevant requirements and non-requirements can be summarized as follows:

* Within the scope of the base pNFS feature, and the use of LAYOUTRETURN and CB_LAYOUTRECALL, both the client and the metadata server are free to drop detailed information about layout ranges and IO modes at any time.

  Any restrictions on the client in this regard derive from the layout type used and its possible need for the client to have such information in order to effect IO operations.

The client MAY do so at any time as long as it does not use these dropped ranges sbsequenly or drop them while IO operations based on them are still bein processed.

Even with the requirement above applying to 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 does not have strict control as to when client will return a recalled layout, the server needs to be able to unilaterally terminate the client's access to the storage devices as descibed by the layout. In terminting access, the server needs to deal with the possibility of lingering I/O requests, i.e., I/O requests that are still in flight to storage devices identified the cancelled layout. All layout type specifications need to specify whether unilateral layout revocation by the metadata server is supported; if it is, the specification needs to must also describe how lingering writes are dealt with. For example, storage devices identified by the canceled layout could be fenced off from the client that held the layout or the data server could be instrycted to sppress access that was previously allowed by the cancelled layout.

Similarly, the metdata seraver MAY also discard detailed information about layout ranges and IO modes. However, in doing so, it MUST NOT forget that it could have oustandning layout for any part of the file so that it is unaware of layout that might be retained by the clients. When that knowledge is lost, the layout MUST be returned or revoked so that it canot be used subsequently.

* The freedom to forget layout details referred to above can be extended to allow the client to eliminate its knowledge of the existence of a layout for a particular file. In doing so, it must ensure that no IOs will use that layout subsequently

* Because of clients and server are allowed to independently discard layouts they were prevously jointly aware of, the processing of layout recalls is more complicated than it would otherwise be. See Section 17.5.5.1 for further discussion.

The remainder of this section discusses how clients and metadata servers might deal with cases in which the other party chooses not to maintain detailed knowledge of layouts that it is aware of and might recall. For example,

* In situations in which conflicts that require callbacks are very rare, a server can use a multi-file callback to recover per-client resources (e.g., via an FSID recall or a multi-file recall within a single CB_COMPOUND). In this case, the result may be significantly less client-server pNFS traffic, although layouts might be recalled even if there might be no need to do so.

* It might also 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 one possible extreme case, a server could manage conflicts on a per-file basis, only sending whole-file callbacks even though clients may request and be granted sub-file ranges.

17.5.4. Committing a Layout

Allowing for varying storage protocol 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 18.4.4 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, in general, it is necessary to resynchronize 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 data committed by a LAYOUTCOMMIT operation is specific to the type of layout because that scope 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 that 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. The metadata server 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-based layouts, synchronization of attributes between the metadata and storage devices, primarily the

size attribute, is not required, but the use of LAYOUTCOMMIT provides a way to optimize the synchronization. Indeed, if a LAYOUT4_NFSV4_1_FILES layout is ever revoked, the metadata server MUST direct all data servers to commit any modified data of the file to stable storage, and synchronize the file's size and time_modify attributes on the metadata server with the those on the data server

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 synchronized with the state that exists on the data storage devices that comprise that file's data as of the client's last sent operation. Thus, a client that queries the size of a file between a WRITE to a storage device and the LAYOUTCOMMIT might observe a size that does not reflect the actual data written.

The client MUST have a layout in order to send a LAYOUTCOMMIT operation.

17.5.4.1. 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 a 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; 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.

17.5.4.2. 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 a byte-range (denoted by the values of 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 data storage devices using 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., zeros are implicitly added between the file's previous EOF and the new EOF. (In many implementations, the zero-extended byte-range 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.

17.5.4.3. LAYOUTCOMMIT and layoutupdate

The LAYOUTCOMMIT argument contains a loca_layoutupdate field (Section 23.42.1) of data type layoutupdate4 (Section 9.3.18). 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 23.43.2) 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.

17.5.5. 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 25.3) and returned with LAYOUTRETURN (see Section 23.44). The CB_LAYOUTRECALL operation may recall a layout identified by a byte-range, all layouts associated with a file system ID (FSID), or all layouts associated with a client ID. Section 17.5.5.2 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; 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 LAYOUTIOMODE4_READ and LAYOUTIOMODE4_RW 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 file-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 passed, 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 of 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.

17.5.5.1. Layour recall/return interaction

As noted in Section 17.5.3.1, 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 that the client "thinks" it has.

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 properly stage the flushing of dirty data and commits and returns of layouts. Also, it indicates to the metadata server that the client is making progress.

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 follows up with a LAYOUTRETURN for what the server recalled (even though what was held does not match what being recalled) or alternatively return the NFS4ERR_NOMATCHING_LAYOUT error if it has no layout to return in the recalled range.

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 sending a LAYOUTRETURN operation for the entire requested range (even thouh it is not actually being "returned") or by returning an NFS4ERR_NOMATCHING_LAYOUT error to the CB_LAYOUTRECALL.

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 overlap 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.

Note that, in the case in which FSID or ALL are specfied, the client needs to do a LAYOUTRETURN for all files for which it holds a layout stateid but need not do so for files for which it does not, even though the metadata server might be aware of these stateids.

17.5.5.2. 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.

17.5.5.3. 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 exactly one higher than what the client currently has recorded, and the client has at least one LAYOUTGET and/or LAYOUTRETURN operation outstanding, or if the client

has a outstanding LAYOUTGET with a non-layout stateid, 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 25.3.)

In addition to the seqid-based mechanism, Section 7.6.3 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 sending 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.

17.5.5.3.1. 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 multiple parallel LAYOUTGET operations for the same file or multiple parallel LAYOUTRETURN operations for the same file or a mix of both. It is permissible for the client to send multiple parallel LAYOUTGET operations for the same file using the layout stateid or multiple parallel LAYOUTRETURN operations for the same file or a mix of both.

It is permissible for the client to use the current stateid (see Section 21.2.3.1.2) 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 it MUST NOT have other LAYOUTGET or LAYOUTRETURN operations outstanding at the same time for that same file.

17.5.5.3.2. 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 sending 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, or the client has an outstanding LAYOUTGET using a non-layout stateid; 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.

17.5.5.3.3. 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 the arguments of LAYOUTGET is one 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.

17.5.5.3.4. 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 7.6.1 when incrementing the layout stateid's "seqid". The

difference is not a valid value for "seqid"; when the value of a "seqid" is 0xFFFFFFFF, the next valid value will be 0x00000001. The modulo arithmetic is also used for the comparisons of "seqid" values in the processing of CB_LAYOUTRECALL events as described above in Section 17.5.5.3.3.

Just as the server validates the "seqid" in the event of CB_LAYOUTRECALL usage, as described in Section 17.5.5.3.3, 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: VALID_SEQID_RANGE = SERVER_CURRENT_SEQID - 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: VALID_SEQID_RANGE = summation over 1..N of (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 "-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) (accounting for zero 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.

17.5.5.3.5. Bulk Recall and Return

pNFS supports recalling and returning all layouts that are for files belonging to a particular fsid (LAYOUTRECALL4_FSID, 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 a 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_FSID, 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 those freed layout stateids for files with the referenced fsid again. Subsequently, for any file with the referenced fsid, to use a layout, the client MUST first send a LAYOUTGET operation in order to obtain a new layout stateid for that file.

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.

17.5.6. Revoking Layouts

Parallel NFS permits servers to revoke layouts from clients that fail to respond to recalls and/or fail to renew their lease in time. Depending on the layout type, the server might revoke the layout and might take certain actions with respect to the client's I/O to data servers.

17.5.7. 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.

17.6. 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 11.16.1) 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 17). 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 to which device ID it needs to send the I/O command 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 to 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.

17.7. 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.

17.7.1. Recovery from Client Restart

Client recovery for layouts is similar to client recovery for other lock and delegation state. When a 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.

17.7.2. 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, the client writes it to the metadata server using the FILE_SYNC4 flag for the WRITEs, or WRITE and COMMIT. Second, the client re-establishes a client ID and session with the server and obtains new layouts and device-ID-to-device-address mappings for the modified data ranges and then writes 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 17.7.4.

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 16.11.9.2. 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 the first two paragraphs of this section.

If sr_status_flags reports no loss of state, then the lease for the layouts that 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 both sent before the lease expires and arrive after the lease expires. See Section 17.7.3.

17.7.3. Dealing with Loss of Layout State on the Metadata Server

This is a description of the case where all of the following are true:

* the metadata server has not restarted

* a pNFS client's layouts have been discarded (usually because the client's lease expired) and are invalid

* 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, they MUST solve this by preventing 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 18.12), and solutions for other layout types are in their respective external specification documents.

17.7.4. Recovery from Metadata Server Restart

The pNFS client will discover that the metadata server has restarted via the methods described in Section 13.4.2 and discussed in a pNFS-specific context in Section 17.7.2, Paragraph 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.

* 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 23.32) operation, and then obtain layouts as desired.

* 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 point.

* 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 the client 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, and 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 that 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 23.42.1) 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 17.5.4.3. If the metadata server's consistency checks on loca_layoutupdate succeed, then the metadata server MUST commit the changed data that was written to the storage device within the scope of the LAYOUTCOMMIT operation. 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 may have lost all uncommitted data within the scope of the failed LAYOUTCOMMIT operation. A client can defend against this risk by caching all data, whether written synchronously or asynchronously in its memory, and by not releasing the cached data until a successful LAYOUTCOMMIT. This condition does not hold true for all layout types; for example, file-based storage devices need not suffer from this limitation.

* 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 point, the failure of LAYOUTCOMMIT means the data in the scope of that LAYOUTCOMMIT may have been lost. The defense against the risk is the same -- cache all written data on the client until a successful LAYOUTCOMMIT

17.7.5. 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, the metadata server must reliably determine that servicing the request will not conflict with an impending OPEN or with a LOCK where the file has mandatory byte-range locking enabled.

As mentioned previously, for expediency, the metadata server might reject some operations (namely, WRITE and LAYOUTGET) during its grace period, because the simplest correct approach is to 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 the metadata server may use this information during the recovery grace period to determine that a WRITE request is safe.

17.7.6. 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 that 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 rewrite 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.

17.8. 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., the same physical hardware is used to implement both a metadata and data server. See Section 18.1 for a description of 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 the metadata server and storage device is immaterial. This is because it is always clear to the pNFS client and server, from the upper-layer protocol being used (NFSv4.1 or non-NFSv4.1), to which role the request to the common server network address is directed.

17.9. Security Issues for pNFS

pNFS separates file system metadata and data and provides access to both. For security (and other) purposes, data within the NFSv4.1 protocol can be divided as follows:

* Non-pNFS-related metadata, typically accessed and modified by using non-pNFS operations directed to the primary server aka the metadata server.

* pNFS-related metadata which provides metadata used to access file data, potentially on another device or server.

  This information, in the form of layouts, is accessed and modified using special pNFS-related operations directed at the metadata server.

* File data, typically accessed using a data access protocol, which might or might not be an NFS protocol, using requests directed at what are called data servers or data storage devices, depending on the layout type.

  In other cases, data can be accessed just as it would be on a non-pNFS server, by making READ and WRITE requests directed to the metadata server.

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 that is accessing file data from a client, regardless of whether this access is directed to the metadata server or to data elsewhere using layouts provided by the primary server.

Despite this important commonality, the ways in which this is done, depends on the layout type. The layout type affects the data access protocol used and the way that the activities of the metadata server and those providing file data access are co-ordinated

* Security issues for data access protocols not layered on RPC is discussed in Section 17.9.1.

* Security issues for data access protocols which are minor versions of NFSv4 and are supported by a separate control protocol are discussed in Section 17.9.2.

* Security for data access protocols for versions of NFS without the support of a separate control protocol is discussed in Section 17.9.3.

Given the multiple types of entities whose co-ordinated effort is required to implement pNFS access, there are a number of inter-entity communications paths which need to be provided with sufficient security to resist attack.

Often, this will require authentication of the requester in order to properly make authorization decisions. When authentication of the principal making the request is not possible, authentication of network peers need to be combined with a trust relationship between the connected peers.

There are a number of possible types of communication paths whose use is possible in various pNFS configurations, depending on the layout type and the specific entities involved.

* When an RPC-based communication path is used, the same sorts of techniques described in the NFSv4-wide security document (expected to be derived from [I-D.dnoveck-nfsv4-security]), are adequate to provide the necessary confidentiality and protection against the execution of unauthorized requests. The details may differ depending on the specific protocol used.

* In other cases, freedom from unauthorized access can be effected by physical isolation of the communication path between the two entities. However, this physical isolation needs to be supplemented in order to make sure that hostile entities cannot gain access to either of these endpoint.

  One common and effective way of blocking such hostile access, is by making sure that the entity is configured so as to not be accessible for general services that can be compromised by external actors. This is often done if the entity is not implemented within a general-purpose operating system or is configured to not to be responsive to general internet traffic.

  Where it is not possible to totally block such access, external authentication of principals is necessary.

Given the three types of entities whose co-ordinated effort is required to implement pNFS access, there are a number of inter-entity communications path which need to be provided with sufficient security to resist attack.

* For communication between the client and the metadata server, the RPC-based security described in the NFSv4-wide security document (expected to be derived from [I-D.dnoveck-nfsv4-security]), is to be used as it is when pNFS is not involved. Note that the associated threat analysis is to be found in that same document.

* For communication between the client and the data storage devices, there are multiple possibilities to consider, based on the type of file access protocol used.

  For data access protocols not using RPC, in general it is not possible to determine whether particular request are appropriately authorized, since there might not be sufficient data present in the request to authenticate the sender or even identify it.

  In such cases, the clients and data storage devices require mutual authentication and they need to trust one another.

* For communication between the metadata server and stat storage device, there needs to be authentication of both peers and a trust relationship between them.

* When there is communication between multiple data storage devices, it is generally best to rely on the metadata servers authentication and its trust of each devices.

17.9.1. Security-related Handling for non-RPC Storage Protocols

These include the blocks layout [RFC5663], the SCSI layout [RFC8154], and the objects layout [RFC5664].

Because these storage protocols do not use RPC, the storage device has no way of determining the specific user making the request. Similarly the storage device has no way of determining the specific open with which a given IO request is associated. As a result, for these storage protocols, the client has the major responsibility for making sure that only valid requests are executed, by implementing the checking of requests to the storage device at the point of issue.

The important role of the client in enforcing these constraints makes authentication of the client peer (e.g. by the use of tls authentication) of critical importance. This is true not only in the case in which AUTH_SYS is used, but also in the RPCSEC_GSS case. In

that latter case, the credentials assure that we know the user making the request but we have no knowledge of the client implementation or any reason to view it as trustworthy in enforcing pNFS rules.

Given the reliance on the client, such storage protocols need to have some way of dealing with an unresponsive client when layouts need to be recalled. Typically, this involves some way for the metadata server to contact storage device directly (e.g. by effecting a persistent reservation) and locking out the unresponsive client (and possibly others).

17.9.2. Security-related Handling for RPC Storage Protocols that are NFSv4 Minor Versions Assisted by Control Protocols

These include not only the use of NFSv4.1 as a storage protocol, as described in Section 18 but also the use of all NFSv4 minor versions as data access protocols as described in [RFC8435], in the "tight" coupling mode.

The use of NFSv4 (with RPC) eliminates the difficulties that apply to Section 17.9.1:

* Because of the use of RPC, the data server is aware of the user of whose behalf the request is made so the same sort of checking made by the metadata server can be done by the data server. When RPCSEC_GSS is in use, authentication of that user is unproblematic. On the other hand, when AUTH_SYS is used to access the data server, authentication of this identity is a serious concern, especially when client host authentication is not available. Generally, the use of AUTH_SYS without client host authentication is to be avoided when accessing the data server, since you are trusting an unauthenticated client to "authenticate" a user’s identity.

* Because of the use of NFSv4, the data server is aware of the specific open with which each IO request is associated.

Because, in this sort of environment, the client is not responsible for checking the validity of IO requests, situations in which a layout becomes invalid are dealt with the ordinary recall mechanism used for other recallable locking objects. However, to deal with e possibility of an unresponsive client, the metadata server will typically have the option of contacting the data server directly. Because of the possibility of communication issues, the control protocol will often use a lease-like mechanism so that, in the absence of communication, layouts are cancelled, rather than being kept indefinitely.

[Author Aside]: Need TH review for the flexible-files case.

17.9.3. Security-related Handling for RPC Storage Protocols using NFS Versions together with Control Protocol Assistance

These include the use of NFSv3, NFSv4.0, and NFSv4.1 as storage protocols, as described in [RFC8435], in the "loose" coupling mode.

With regard to the difficulties discussed in Section 17.9.1, specifics may vary based on the particular storage protocol used and the possible use of client-host authentication.

* Since loose authentication is only described for the AUTH_SYS case, we have to assume that RPCSEC_GSS will not be used. As a result, it is RECOMMENDED that client host authentication be available to prevent attackers acting as if they were unauthenticated clients and presenting their requests for execution. The authentication needed to prevent this is the authentication of the clients to the data server. When it is not used, serious security difficulties arise because the clients are not authenticated to the data server which is executing their requests.

  The authentication of clients to the metadata server can ameliorate the problem, but the main value of doing so is the encryption of traffic between the client and the metadata server, provided by rpc-tls.

  When this problem exists, use of RPCSEC_GSS by the clients in accessing the metadata server, is not, by itself, helpful. The confidentiality of traffic between the client and the metadata server is necessary, whether that is provided by RPCSEC_GSS privacy services or by rpc-tls encryption.

* In the case of NFSv3 as a storage protocol, there is no way for the data server to determine the open with which each request is assigned. While it might be possible to make this determination in the case in which NFSv4 minor versions are used as storage protocol, it appears that the loose coupling option makes no provision for this either.

As a result, the situation is quite like that described in Section 17.9.1 even though the storage protocol is RPC-based, with clients, rather than data servers responsible for checking request validity. As a result, similar issues arise when clients do not respond properly when layouts are recalled. The metadata server has the ability to make such layouts unusable by changing ownership of the data files involved.

[Author Aside]: TH Needs to review this section.

18. NFSv4.1 as a Data Access Protocol in pNFS: the File Layout Type

This section describes the semantics and format of NFSv4.1 file-based layouts for pNFS. NFSv4.1 file-based layouts use the LAYOUT4_NFSV4_1_FILES layout type. The LAYOUT4_NFSV4_1_FILES type defines striping data across multiple NFSv4.1 data servers.

18.1. Client ID and Session Considerations

Sessions are a REQUIRED feature of NFSv4.1, and this applies 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:

* Metadata server (EXCHGID4_FLAG_USE_PNFS_MDS is set in the result eir_flags).

* Data server (EXCHGID4_FLAG_USE_PNFS_DS).

* 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. However, the server 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

Table 6

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 the 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 that 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 needs to 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 a 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 will 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 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 might 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.

18.1.1. Sessions Considerations for Data Servers

Section 7.11.2 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 18.6) 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 for which it is providing I/O, and it 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.

18.2. 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. A 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 a 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 a 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).

18.3. 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 11.16.4). 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
                                   = 0xFFFFFFC0;

   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 data 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 9.3.19. The client uses the layout hint in the layout_hint (Section 11.16.4) 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 18.4.4), which is controlled by the value of the expression nflh_util & NFL4_UFLG_DENSE ("&" represents the bitwise AND operator). 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 18.7), 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
 * data type device_addr4:
 */
struct nfsv4_1_file_layout_ds_addr4 {
        uint32_t        nflda_stripe_indices<>;
        multipath_list4 nflda_multipath_ds_list<>;
};
```

The nfsv4_1_file_layout_ds_addr4 data type represents the device address. It is composed of two fields:

1. nflda_multipath_ds_list: An array of lists of data servers, where each list can be one or more elements, and each element represents a data server address that may serve equally as the target of I/O operations (see Section 18.5). The length of this array might be different than the stripe count.

2. nflda_stripe_indices: An array of indices used to index into nflda_multipath_ds_list. The value of each element of nflda_stripe_indices MUST be less than the number of elements in nflda_multipath_ds_list. Each element of nflda_multipath_ds_list SHOULD be referred to by one or more elements of nflda_stripe_indices. The number of elements in nflda_stripe_indices is always equal to the stripe count.

```
/*
 * Encoded in the loc_body field of
 * data 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 that 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 nflda_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 18.4.1).

   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 18.4.4).

   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 zero.

5. nfl_fh_list: An array of data server filehandles for each list of data servers in each element of the nflda_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 nflda_multipath_ds_list. Thus, in this case, when sending an I/O operation to any data server in nflda_multipath_ds_list[X], the filehandle in nfl_fh_list[X] MUST be used.

     See the discussion on sparse packing in Section 18.4.4.

   * If dense packing is being used, the number of elements in nfl_fh_list MUST be the same as the number of elements in nflda_stripe_indices. Thus, when sending an I/O operation to any data server in nflda_multipath_ds_list[nflda_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 nflda_multipath_ds_list[nflda_stripe_indices[i]] and

nflda_multipath_ds_list[nflda_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 18.4.4.

The details on the interpretation of the layout are in Section 18.4.

18.4. Interpreting the File Layout

18.4.1. Determining the Stripe Unit Number

To find the stripe unit number that corresponds to the client's logical file offset, the pattern offset will also be used. The i'th stripe unit (SUi) is:

```
relative_offset = file_offset - nfl_pattern_offset;
SUi = floor(relative_offset / stripe_unit_size);
```

18.4.2. 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 (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 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 18.5) 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:

nflda_stripe_indices<> = { 2, 0, 1, 0 }

Now suppose the client gets a layout that has a device ID that maps to the above device address. The initial index contains

nfl_first_stripe_index = 2,

and the filehandle list is

nfl_fh_list = { 0x36, 0x87, 0x67 }.

If the client wants to write to SU0, the set of valid { network address, filehandle } combinations for SUi are determined by:

nfl_first_stripe_index = 2

So

idx = nflda_stripe_indices[(0 + 2) % 4]

= nflda_stripe_indices[2]

= 1

So

nflda_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 13 stripe 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

Table 7

18.4.3. 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 18.5) 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:

```
   nflda_stripe_indices<> = { 2, 0, 1, 0 }
```

Now suppose the client gets a layout that has a device ID that maps to the above device address. The initial index contains

```
   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 each stripe unit MUST use a different filehandle. If the client wants to write to SU1, the set of valid { network address, filehandle } combinations for SUi are determined by:

```
   nfl_first_stripe_index = 2
```

So

```
   j = (1 + 2) % 4 = 3

      idx = nflda_stripe_indices[j]

      = nflda_stripe_indices[3]

      = 0
```

So

```
   nflda_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 nflda_stripe_indices[1] = 0. Then nflda_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 13 stripe units are:

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

Table 8

18.4.4. 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) 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 will 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 sending 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 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 operation 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 will be packed. Using the same striping pattern and stripe unit size that were used for the sparse packing example, the corresponding dense packing example would have all stripe units of all data files filled as follows:

* 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.

* 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 nflda_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.

18.5. 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 7.5) 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 be 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 nflda_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 the possibility 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 nflda_multipath_ds_list. If less optimal network addresses exist to provide failover, 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 nflda_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 nflda_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 will 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 nflda_multipath_ds_list, they will designate the same data server, and the two data server addresses will support the implementation of client ID or session trunking (the latter is RECOMMENDED) as defined in Section 7.5. 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.

18.6. 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 consists of management operations. 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 17.5.1, 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 18.1 -- any attempted use of operations against a data server 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 it 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 that is ambiguous regarding its personality assignment. This includes 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 using such filehandles are within this class, with the following exception. The exception is that if the operation uses a stateid that is incompatible with a data-server personality (e.g., a special stateid or the stateid has a non-zero "seqid" field, see Section 18.10.1), the operation is in class 3, as described below. A COMPOUND containing multiple class 1 operations (and operations of no other class) MAY be sent to a server with multiple concurrent data server and non-data-server personalities.

2. An operation that is unambiguously referable to the data-server personality. This includes data-server I/O operations where the filehandle is one that can only be validly directed to the data-server personality.

3. An operation that is unambiguously referable to the non-data-server personality. This includes 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 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 17.5.4.2). Section 18.11 describes the mechanism by which the client is to handle data-server files that do not reflect the metadata server’s size.

18.7. COMMIT through Metadata Server

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 file layout type.

* 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 sometimes useful when file striping is implemented within the pNFS server (instead of the file system), with the individual data servers each implementing their own file systems.

* When the flag is TRUE, COMMIT operations MUST be sent to the metadata server, rather than to the individual data servers. This approach is sometimes useful when file striping is implemented within the clustered file system that is the backend to the pNFS server. In such an implementation, each COMMIT to each data server might result in repeated writes of metadata blocks to the detriment of write performance. Sending a single COMMIT to the metadata server can be more efficient when there exists a clustered file system capable of implementing such a coordinated 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 are 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 resending the WRITEs to the original data server or using another path to that data if the layout has not been recalled. Alternatively, the client can obtain a new layout or it could rewrite the data directly to 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 17.7.6 for recovery options.

18.8. 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.

18.9. LAYOUTCOMMIT on file layouts

For file layouts, WRITEs to a Data Server that return a stable_how4 value of FILE_SYNC4 guarantee that data and file system metadata are on stable storage. This implies that a LAYOUTCOMMIT is not needed in order to make the data and metadata visible to the metadata server and other clients.

For file layouts, when WRITE to the data server returns UNSTABLE4 or DATA_SYNC4 and the NFL4_UFLG_COMMIT_THRU_MDS flag is set, the client MUST send the COMMIT to the metadata server. A successful COMMIT to the metadata server guarantees that data and file system metadata are on stable storage. As a result, any time that NFS4_UFLG_COMMIT_THRU_MDS is set, a LAYOUTCOMMIT (of the byte range specified by the layout) is not needed.

For file layouts, when NFL4_UFLG_COMMIT_THRU_MDS flag is not set, and WRITE or COMMIT to the data server return DATA_SYNC4, the client MUST send the LAYOUTCOMMIT to the metadata server in order to synchronize file metadata.

The following table summarizes the conditions under which a LAYOUTCOMMIT is needed, and the effects of a COMMIT to a data server and metadata server.

NFL4_UFLG_ COMMIT_ THRU_MDS	WRITE to DS returns	Meaning of COMMIT to DS	Meaning of COMMIT to DS	LAYOUT COMMIT reequired
Not Set	UNSTABLE	DATA_SYNC4	Nothing	Yes
Not Set	DATA_SYNC4	Nothing	Nothing	Yes
Not Set	FILE_SYNC4	Nothing	Nothing	NO
Set	UNSTABLE	Nothing	FILE_SYNC4	NO
Set	DATA_SYNC4	Nothing	FILE_SYNC4	NO
Not Set	FILE_SYNC4	Nothing	Nothing	NO

Table 9

Note that a client can always demand FILE_SYNC4 or DATA_SYNC4 via WRITE arguments. Also note that specifying these stability levels might adversely impact performance.

If a LAYOUTCOMMIT is required, it should be sent before CLOSE to maintain close-to-open semantics. If required, it should be sent before LOCKU, OPEN_DOWNGRADE, LAYOUTRETURN, and when the application issues fsync() [fsync]. Again, if LAYOUTCOMMIT is required, it should be sent periodically to keep the file size and modification time approximately up-to-date. This allows the use of commands such as "tail -f" which copies its input file to the standard output and updates the output as new lines become available in the input file. It is the client implementation’s choice to determine how frequently LAYOUTCOMMIT is issued. Possible policies include every N’th COMMIT to a data server, every N’th unit of time, or after writing a stripe to a set of data servers.

Even if a required LAYOUTCOMMIT is not issued by the client, the data server and metadata servers have a set of responsibilities necessary to provide data consistency:

1) Data servers MUST commit data and synchronize modification and size attributes with the metadata server before a layout is revoked as described in section 12.5.4.

2) Data servers SHOULD commit data and synchronize modification and size attributes with the metadata server after the metadata server reboots. In theory the client should commit the data, but this avoids the problem where both the client and metadata server crash at the same time.

3) The metadata server MAY periodically poll data servers to synchronize modification and size attributes.

18.10. Metadata and Data Server State Coordination

18.10.1. 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 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 18.10.2. 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 sending the I/O

requests. The rules for doing so when referencing data servers are somewhat different from those discussed in Section 13.2.5, which apply when accessing metadata servers.

The following rules, applied in order of decreasing priority, govern the selection of the appropriate stateid:

* If the client holds a delegation for the file in question, the delegation stateid should be used.

* 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.

* If the data server had previously responded with NFS4ERR_LOCKED to use of the OPEN stateid, then the client should use the byte-range lock stateid whenever one exists for that open file with the current lock-owner.

* Special stateids should never be used. If they are used, the data server MUST reject the I/O with an NFS4ERR_BAD_STATEID error.

18.10.2. Data Server State Propagation

Since the metadata server, which handles byte-range lock and open-mode state changes as well as ACLs, might not be co-located with the data servers where I/O accesses 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.

18.10.2.1. Lock State Propagation

If the pNFS server supports mandatory byte-range locking, any mandatory byte-range 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 byte-range 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 byte-range lock state is not used for checking I/O accesses at the data servers, there is no semantic reason for propagating advisory byte-range 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 to avoid using the byte-range lock-based stateids for I/O. 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.

18.10.2.2. 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 OPEN_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.

18.10.2.3. 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 may be necessary to re-synchronize state such as the size attribute, and the setting of mtime/change/atime. See Section 17.5.4 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 23.16.4) 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 it 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.

18.11. Data Server Component File Size

A potential problem exists when a component data file on a particular data server has 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 zeroes back as a result of the READ. However, if the striping pattern directs the client to send the READ to a data server other than the one that received the client's original WRITE, the data server servicing the READ may believe that the file's size is still 0 bytes. In that event, the data server's READ response will contain zero bytes and an indication of EOF. The data server can only return zeroes 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. When the offset in the arguments to READ is less than the client's view of the file size, if the READ response indicates EOF and/or contains fewer bytes than requested, the client will interpret such a response as a hole in the file, and the NFS client will substitute zeroes for the data.

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, if necessary 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.

18.12. Layout Revocation and Fencing

As described in Section 17.7, the layout-type-specific storage protocol is responsible for handling the effects of I/Os that started before lease expiration and extend through lease expiration. The LAYOUT4_NFSV4_1_FILES layout type can prevent all I/Os to data servers from being executed after lease expiration (this prevention is called "fencing"), without relying on a precise client lease timer and without requiring data servers to maintain lease timers. The LAYOUT4_NFSV4_1_FILES pNFS server has the flexibility to revoke individual layouts, and thus fence I/O on a per-file basis.

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 sending 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 18.1, 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 ensure 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 18.3).

Before the metadata server takes any action to revoke 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 has the following implications.

* The metadata server must not restripe a file until it has contacted all of the data servers to invalidate the layouts from the previous instance.

* The metadata server must not give out mandatory locks that conflict with layouts from the previous instance without either doing a specific layout invalidation (as it would have to do anyway) or doing a global data server invalidation.

18.13. Security Issues for the File Layout Type

The NFSv4.1 file layout type MUST adhere to the security considerations outlined in Section 17.9. 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 a READ or WRITE operation on a file due to its ACL, mode attribute, open access mode, open deny mode, mandatory byte-range 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 18.10.2 for more details.

The methods for authentication, integrity, and privacy for data servers based on the LAYOUT4_NFSV4_1_FILES layout type 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. Thus, when using the LAYOUT4_NFSV4_1_FILES layout type, the impact on the RPC-based security model due to pNFS (as alluded to in Sections 2.7 and 2.8.2) is zero.

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.

19. Internationalization

Internationalization for NFSv4.1 is described in [I-D.ietf-nfsv4-internationalization], just as it is for other minor versions. The only NFSv4.1-specific element, the fs_charset_cap attribute is described in Section 19.1 below.

19.1. UTF-8 Capabilities

```
const FSCHARSET_CAP4_CONTAINS_NON_UTF8  = 0x1;
const FSCHARSET_CAP4_ALLOWS_ONLY_UTF8   = 0x2;

typedef uint32_t        fs_charset_cap4;
```

This attribute provides a simple way of determining whether a particular file system behaves as a UTF-8-only server and rejects file names which are not valid Unicode strings encoded using UTF-8. When this attribute is supported and the value returned has the FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 flag set, the error NFS4ERR_INVAL MUST be returned if any file name argument contains a string which is not a valid UTF-8-encoded string.

When this attribute is supported and the value returned has the FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 flag clear, the error NFS4ERR_INVAL will not be returned based on adherence to the rules of UTF-8. While such file systems are generally UTF-8-unaware, this cannot be assumed, since server are allowed (in some circumstances; it is a "SHOULD NOT") to accept non-UTF-8 names while being aware of the structure of UTF-8-conforming names, for the purposes of determining canonical equivalence, for example.

With regard to the flag FSCHARSET_CAP4_CONTAINS_NON_UTF8, it has proved impossible to determine, from existing treatments of this attribute, any value that might be helpful here. As a result, we are forced to assume that this flag is always a complement of FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 and that any result in which it is not is to be ignored, with the appropriate handling being the same as would apply if the attribute were not supported.

When this attribute is not supported, the client can perform a LOOKUP using a name not conforming to the rules of UTF-8 and use the error returned to determine whether names which not UTF-8-encoded Unicode are accepted.

20. 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.

20.1. Error Definitions

Error	Number	Description
NFS4_OK	0	Section 20.1.3.1
NFS4ERR_ACCESS	13	Section 20.1.6.1
NFS4ERR_ATTRNOTSUPP	10032	Section 20.1.15.1
NFS4ERR_ADMIN_REVOKED	10047	Section 20.1.5.1
NFS4ERR_BACK_CHAN_BUSY	10057	Section 20.1.12.1
NFS4ERR_BADCHAR	10040	Section 20.1.7.1
NFS4ERR_BADHANDLE	10001	Section 20.1.2.1
NFS4ERR_BADIOMODE	10049	Section 20.1.10.1
NFS4ERR_BADLAYOUT	10050	Section 20.1.10.2
NFS4ERR_BADNAME	10041	Section 20.1.7.2
NFS4ERR_BADOWNER	10039	Section 20.1.15.2
NFS4ERR_BADSESSION	10052	Section 20.1.11.1
NFS4ERR_BADSLOT	10053	Section 20.1.11.2
NFS4ERR_BADTYPE	10007	Section 20.1.4.1
NFS4ERR_BADXDR	10036	Section 20.1.1.1
NFS4ERR_BAD_COOKIE	10003	Section 20.1.1.2
NFS4ERR_BAD_HIGH_SLOT	10077	Section 20.1.11.3
NFS4ERR_BAD_RANGE	10042	Section 20.1.8.1
NFS4ERR_BAD_SEQID	10026	Section 20.1.16.1
NFS4ERR_BAD_SESSION_DIGEST	10051	Section 20.1.12.2
NFS4ERR_BAD_STATEID	10025	Section 20.1.5.2
NFS4ERR_CB_PATH_DOWN	10048	Section 20.1.11.4

NFS4ERR_CLID_INUSE	10017	Section 20.1.13.2
NFS4ERR_CLIENTID_BUSY	10074	Section 20.1.13.1
NFS4ERR_COMPLETE_ALREADY	10054	Section 20.1.9.1
NFS4ERR_CONN_NOT_BOUND_TO_SESSION	10055	Section 20.1.11.6
NFS4ERR_DEADLOCK	10045	Section 20.1.8.2
NFS4ERR_DEADSESSION	10078	Section 20.1.11.5
NFS4ERR_DELAY	10008	Section 20.1.1.3
NFS4ERR_DELEG_ALREADY_WANTED	10056	Section 20.1.14.1
NFS4ERR_DELEG_REVOKED	10087	Section 20.1.5.3
NFS4ERR_DENIED	10010	Section 20.1.8.3
NFS4ERR_DIRDELEG_UNAVAIL	10084	Section 20.1.14.2
NFS4ERR_DQUOT	69	Section 20.1.4.2
NFS4ERR_ENCR_ALG_UNSUPP	10079	Section 20.1.13.3
NFS4ERR_EXIST	17	Section 20.1.4.3
NFS4ERR_EXPIRED	10011	Section 20.1.5.4
NFS4ERR_FBIG	27	Section 20.1.4.4
NFS4ERR_FHEXPIRED	10014	Section 20.1.2.2
NFS4ERR_FILE_OPEN	10046	Section 20.1.4.5
NFS4ERR_GRACE	10013	Section 20.1.9.2
NFS4ERR_HASH_ALG_UNSUPP	10072	Section 20.1.13.4
NFS4ERR_INVAL	22	Section 20.1.1.4
NFS4ERR_IO	5	Section 20.1.4.6
NFS4ERR_ISDIR	21	Section 20.1.2.3
NFS4ERR_LAYOUTTRYLATER	10058	Section 20.1.10.3

NFS4ERR_LAYOUTUNAVAILABLE	10059	Section 20.1.10.4
NFS4ERR_LEASE_MOVED	10031	Section 20.1.16.2
NFS4ERR_LOCKED	10012	Section 20.1.8.4
NFS4ERR_LOCKS_HELD	10037	Section 20.1.8.5
NFS4ERR_LOCK_NOTSUPP	10043	Section 20.1.8.6
NFS4ERR_LOCK_RANGE	10028	Section 20.1.8.7
NFS4ERR_MINOR_VERS_MISMATCH	10021	Section 20.1.3.2
NFS4ERR_MLINK	31	Section 20.1.4.7
NFS4ERR_MOVED	10019	Section 20.1.2.4
NFS4ERR_NAMETOOLONG	63	Section 20.1.7.3
NFS4ERR_NOENT	2	Section 20.1.4.8
NFS4ERR_NOFILEHANDLE	10020	Section 20.1.2.5
NFS4ERR_NOMATCHING_LAYOUT	10060	Section 20.1.10.5
NFS4ERR_NOSPC	28	Section 20.1.4.9
NFS4ERR_NOTDIR	20	Section 20.1.2.6
NFS4ERR_NOTEMPTY	66	Section 20.1.4.10
NFS4ERR_NOTSUPP	10004	Section 20.1.1.5
NFS4ERR_NOT_ONLY_OP	10081	Section 20.1.3.3
NFS4ERR_NOT_SAME	10027	Section 20.1.15.3
NFS4ERR_NO_GRACE	10033	Section 20.1.9.3
NFS4ERR_NXIO	6	Section 20.1.16.3
NFS4ERR_OLD_STATEID	10024	Section 20.1.5.5
NFS4ERR_OPENMODE	10038	Section 20.1.8.8
NFS4ERR_OP_ILLEGAL	10044	Section 20.1.3.4

NFS4ERR_OP_NOT_IN_SESSION	10071	Section 20.1.3.5
NFS4ERR_PERM	1	Section 20.1.6.2
NFS4ERR_PNFS_IO_HOLE	10075	Section 20.1.10.6
NFS4ERR_PNFS_NO_LAYOUT	10080	Section 20.1.10.7
NFS4ERR_RECALLCONFLICT	10061	Section 20.1.14.3
NFS4ERR_RECLAIM_BAD	10034	Section 20.1.9.4
NFS4ERR_RECLAIM_CONFLICT	10035	Section 20.1.9.5
NFS4ERR_REJECT_DELEG	10085	Section 20.1.14.4
NFS4ERR_REP_TOO_BIG	10066	Section 20.1.3.6
NFS4ERR_REP_TOO_BIG_TO_CACHE	10067	Section 20.1.3.7
NFS4ERR_REQ_TOO_BIG	10065	Section 20.1.3.8
NFS4ERR_RESOURCE	10018	Section 20.1.16.4
NFS4ERR_RESTOREFH	10030	Section 20.1.16.5
NFS4ERR_RETRY_UNCACHED_REP	10068	Section 20.1.3.9
NFS4ERR_RETURNCONFLICT	10086	Section 20.1.10.8
NFS4ERR_ROFS	30	Section 20.1.4.11
NFS4ERR_SAME	10009	Section 20.1.15.4
NFS4ERR_SHARE_DENIED	10015	Section 20.1.8.9
NFS4ERR_SEQUENCE_POS	10064	Section 20.1.3.10
NFS4ERR_SEQ_FALSE_RETRY	10076	Section 20.1.11.7
NFS4ERR_SEQ_MISORDERED	10063	Section 20.1.11.8
NFS4ERR_SERVERFAULT	10006	Section 20.1.1.6
NFS4ERR_STALE	70	Section 20.1.2.7
NFS4ERR_STALE_CLIENTID	10022	Section 20.1.13.5

NFS4ERR_STALE_STATEID	10023	Section 20.1.16.6
NFS4ERR_SYMLINK	10029	Section 20.1.2.8
NFS4ERR_TOOSMALL	10005	Section 20.1.1.7
NFS4ERR_TOO_MANY_OPS	10070	Section 20.1.3.11
NFS4ERR_UNKNOWN_LAYOUTTYPE	10062	Section 20.1.10.9
NFS4ERR_UNSAFE_COMPOUND	10069	Section 20.1.3.12
NFS4ERR_WRONGSEC	10016	Section 20.1.6.3
NFS4ERR_WRONG_CRED	10082	Section 20.1.6.4
NFS4ERR_WRONG_TYPE	10083	Section 20.1.2.9
NFS4ERR_XDEV	18	Section 20.1.4.12

Table 10: Protocol Error Definitions

20.1.1. General Errors

This section deals with errors that are applicable to a broad set of different purposes.

20.1.1.1. 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 that 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.

20.1.1.2. 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.

20.1.1.3. 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 requester should wait and then try the request with a new slot and sequence value.

Some examples of situations that might lead to this error being returned:

* A server that supports hierarchical storage receives a request to process a file that had been migrated.

* An operation requires a delegation recall to proceed, but the need to wait for this delegation to be recalled and returned makes processing this request in a timely fashion impossible.

* A request is being performed on a session being migrated from another server as described in Section 16.14.3, and the lack of full information about the state of the session on the source makes it impossible to process the request immediately.

In such cases, returning the error NFS4ERR_DELAY allows necessary preparatory operations to proceed without holding up requester resources such as a session slot. After delaying for period of time, the requester can then re-send the operation in question, often as part of a nearly identical request. Because of the need to avoid spurious reissues of non-idempotent operations and to avoid acting in response to NFS4ERR_DELAY errors returned on responses returned from the replier's reply cache, integration with the session-provided reply cache is necessary. There are a number of cases to deal with, each of which requires different sorts of handling by the requester and replier:

* If NFS4ERR_DELAY is returned on a SEQUENCE operation, the request is retried in full with the SEQUENCE operation containing the same slot and sequence values. In this case, the replier MUST avoid returning a response containing NFS4ERR_DELAY as the response to SEQUENCE solely because an earlier instance of the same request returned that error and it was stored in the reply cache. If the replier did this, the retries would not be effective as there would be no opportunity for the replier to see whether the condition that generated the NFS4ERR_DELAY had been rectified during the time between the original request and the retry.

* If NFS4ERR_DELAY is returned on an operation other than SEQUENCE that validly appears as the first operation of a request, the handling is similar. The request can be retried in full without modification. In this case as well, the replier MUST avoid

returning a response containing NFS4ERR_DELAY as the response to an initial operation of a request solely on the basis of its presence in the reply cache. If the replier did this, the retries would not be effective as there would be no opportunity for the replier to see whether the condition that generated the NFS4ERR_DELAY had been rectified during the interim between the original request and the retry.

* If NFS4ERR_DELAY is returned on an operation other than the first in the request, the request when retried MUST contain a SEQUENCE operation that is different than the original one, with either the slot ID or the sequence value different from that in the original request. Because requesters do this, there is no need for the replier to take special care to avoid returning an NFS4ERR_DELAY error obtained from the reply cache. When no non-idempotent operations have been processed before the NFS4ERR_DELAY was returned, the requester should retry the request in full, with the only difference from the original request being the modification to the slot ID or sequence value in the reissued SEQUENCE operation.

* When NFS4ERR_DELAY is returned on an operation other than the first within a request and there has been a non-idempotent operation processed before the NFS4ERR_DELAY was returned, reissuing the request as is normally done would incorrectly cause the re-execution of the non-idempotent operation.

  To avoid this situation, the requester should reissue the request without the non-idempotent operation. The request still must use a SEQUENCE operation with either a different slot ID or sequence value from the SEQUENCE in the original request. Because this is done, there is no way the replier could avoid spuriously re-executing the non-idempotent operation since the different SEQUENCE parameters prevent the requester from recognizing that the non-idempotent operation is being retried.

Note that without the ability to return NFS4ERR_DELAY and the requester’s willingness to re-send when receiving it, deadlock might result. For example, 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.

20.1.1.4. NFS4ERR_INVAL (Error Code 22)

The arguments for this operation are not valid for some reason, even though they do match those specified in the XDR definition for the request.

20.1.1.5. NFS4ERR_NOTSUPP (Error Code 10004)

Operation not supported because the operation is either of the following:

* an OPTIONAL one and is not supported by this server or the file system on which it is issued.

* an operation which MUST NOT be implemented in the current minor version.

In addition, this error may be returned in certain unsupported instances of the LINK operation.

20.1.1.6. NFS4ERR_SERVERFAULT (Error Code 10006)

An error occurred on the server that 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.

20.1.1.7. 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.

20.1.2. 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.

20.1.2.1. NFS4ERR_BADHANDLE (Error Code 10001)

Illegal NFS filehandle for the current server. The current filehandle failed internal consistency checks. Once accepted as valid (by PUTFH), no subsequent status change can cause the filehandle to generate this error.

20.1.2.2. NFS4ERR_FHEXPIRED (Error Code 10014)

A current or saved filehandle that is an argument to the current operation is volatile and has expired at the server.

20.1.2.3. 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.

20.1.2.4. NFS4ERR_MOVED (Error Code 10019)

The file system that contains the current filehandle object is not present at the server or is not accessible with the network address used. It may have been made accessible on a different set of network addresses, relocated or migrated to another server, or it 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 16.3.

As with the case of NFS4ERR_DELAY, it is possible that one or more non-idempotent operations may have been successfully executed within a COMPOUND before NFS4ERR_MOVED is returned. Because of this, once the new location is determined, the original request that received the NFS4ERR_MOVED should not be re-executed in full. Instead, the client should send a new COMPOUND with any successfully executed non-idempotent operations removed. When the client uses the same session for the new COMPOUND, its SEQUENCE operation should use a different slot ID or sequence.

20.1.2.5. 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).

20.1.2.6. NFS4ERR_NOTDIR (Error Code 20)

The current (or saved) filehandle designates an object that is not a directory for an operation in which a directory is required.

20.1.2.7. 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.

20.1.2.8. 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.

20.1.2.9. NFS4ERR_WRONG_TYPE (Error Code 10083)

The current (or saved) filehandle designates an object that 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_INVAL.

20.1.3. Compound Structure Errors

This section deals with errors that relate to the 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 add to these basic constraints by requiring a Sequence operation (either SEQUENCE or CB_SEQUENCE) at the start of the Compound.

20.1.3.1. NFS_OK (Error code 0)

Indicates the operation completed successfully, in that all of the constituent operations completed without error.

20.1.3.2. 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 will specify a result count of zero.

20.1.3.3. 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 whenever the Compound does not start with a Sequence operation. This error results when that constraint is not met.

20.1.3.4. 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.

20.1.3.5. 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.

20.1.3.6. NFS4ERR_REP_TOO_BIG (Error Code 10066)

The reply to a Compound would exceed the channel's negotiated maximum response size.

20.1.3.7. 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.

20.1.3.8. NFS4ERR_REQ_TOO_BIG (Error Code 10065)

The Compound request exceeds the channel's negotiated maximum size for requests.

20.1.3.9. NFS4ERR_RETRY_UNCACHED_REP (Error Code 10068)

The requester has attempted a retry of a Compound that it previously requested not be placed in the reply cache.

20.1.3.10. NFS4ERR_SEQUENCE_POS (Error Code 10064)

A Sequence operation appeared in a position other than the first operation of a Compound request.

20.1.3.11. NFS4ERR_TOO_MANY_OPS (Error Code 10070)

The Compound request has too many operations, exceeding the count negotiated when the session was created.

20.1.3.12. 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 that changes the current filehandle and that is not followed by a GETFH.

20.1.4. File System Errors

These errors describe situations that occurred in the underlying file system implementation rather than in the protocol or any NFSv4.x feature.

20.1.4.1. 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 the type is not supported by the server, or because the type is not intended to be created by CREATE (such as a regular file or named attribute, for which OPEN is used to do the file creation).

20.1.4.2. NFS4ERR_DQUOT (Error Code 69)

Resource (quota) hard limit exceeded. The user’s resource limit on the server has been exceeded.

20.1.4.3. NFS4ERR_EXIST (Error Code 17)

A file of the specified target name (when creating, renaming, or linking) already exists.

20.1.4.4. NFS4ERR_FBIG (Error Code 27)

The file is too large. The operation would have caused the file to grow beyond the server’s limit.

20.1.4.5. 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.

20.1.4.6. NFS4ERR_IO (Error Code 5)

Indicates that an I/O error occurred for which the file system was unable to provide recovery.

20.1.4.7. 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.

20.1.4.8. NFS4ERR_NOENT (Error Code 2)

Indicates no such file or directory. The file or directory name specified does not exist.

20.1.4.9. NFS4ERR_NOSPC (Error Code 28)

Indicates there is no space left on the device. The operation would have caused the server's file system to exceed its limit.

20.1.4.10. NFS4ERR_NOTEMPTY (Error Code 66)

An attempt was made to remove a directory that was not empty.

20.1.4.11. NFS4ERR_ROFS (Error Code 30)

Indicates a read-only file system. A modifying operation was attempted on a read-only file system.

20.1.4.12. 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:

* that between file systems (where the fsids are different).
* that between different named attribute directories or between a named attribute directory and an ordinary directory.
* that between byte-ranges of a file system that the file system implementation treats as separate (for example, for space accounting purposes), and where cross-connection between the byte-ranges are not allowed.

20.1.5. 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 locking state that has been revoked. Depending on the operation, the stateid when valid may designate opens, byte-range locks, file or directory delegations, layouts, or

device maps.

20.1.5.1. 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.

20.1.5.2. NFS4ERR_BAD_STATEID (Error Code 10026)

A stateid does not properly designate any valid state. See Sections 13.2.4 and 13.2.3 for a discussion of how stateids are validated.

20.1.5.3. 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.

20.1.5.4. 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.

20.1.5.5. NFS4ERR_OLD_STATEID (Error Code 10024)

A stateid with a non-zero seqid value is not the most current seqid for the state.

20.1.6. Security Errors

These are the various permission-related errors in NFSv4.1.

20.1.6.1. 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 20.1.6.2), which restricts itself to owner or privileged-user permission failures, and NFS4ERR_WRONG_CRED (Section 20.1.6.4), which deals with appropriate permission to delete or modify transient objects based on the credentials of the user that created them.

20.1.6.2. 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.

20.1.6.3. 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.

20.1.6.4. NFS4ERR_WRONG_CRED (Error Code 10082)

An operation that manipulates state was attempted by a principal that was not allowed to modify that piece of state.

20.1.7. Name Errors

Names in NFSv4 are typically UTF-8 strings, although it is possible for servers, when accessing certain file systems, to support other encodings to support internationalization. When the strings are of length zero or are not valid UTF-8 encoding in a file system that only supports UTF-8 encodings, the error NFS4ERR_INVAL results. Besides this, there are a number of other errors to indicate specific problems with names.

20.1.7.1. NFS4ERR_BADCHAR (Error Code 10040)

A string contains a character that is not supported by the server in the context in which it being used.

20.1.7.2. NFS4ERR_BADNAME (Error Code 10041)

A name string in a request consisted of valid characters supported by the server, but the name is not supported by the server as a valid name for the 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.

20.1.7.3. NFS4ERR_NAMETOOLONG (Error Code 63)

Returned when the filename in an operation exceeds the server's implementation limit.

20.1.8. Locking Errors

This section deals 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 Section 20.1.9.

20.1.8.1. NFS4ERR_BAD_RANGE (Error Code 10042)

The byte-range of a LOCK, LOCKT, or LOCKU operation is not allowed by the server. For example, this error results when a server that only supports 32-bit ranges receives a range that cannot be handled by that server. (See Section 23.10.3.)

20.1.8.2. NFS4ERR_DEADLOCK (Error Code 10045)

The server has been able to determine a byte-range locking deadlock condition for a READW_LT or WRITEW_LT LOCK operation.

20.1.8.3. 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 14.6 for a discussion of the re-send.

20.1.8.4. 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:

* There is a share reservation inconsistent with the I/O being done.

* The range to be read or written intersects an existing mandatory byte-range lock.

20.1.8.5. NFS4ERR_LOCKS_HELD (Error Code 10037)

An operation was prevented by the unexpected presence of locks.

20.1.8.6. NFS4ERR_LOCK_NOTSUPP (Error Code 10043)

A LOCK operation was attempted that would require the upgrade or downgrade of a byte-range lock range already held by the owner, and the server does not support atomic upgrade or downgrade of locks.

20.1.8.7. NFS4ERR_LOCK_RANGE (Error Code 10028)

A LOCK operation is operating on a range that overlaps in part a currently held byte-range lock for the current lock-owner and does not precisely match a single such byte-range lock where the server does not support this type of request, and thus does not implement POSIX locking semantics [fcntl]. See Sections 23.10.4, 23.11.4, and 23.12.4 for a discussion of how this applies to LOCK, LOCKT, and LOCKU respectively.

20.1.8.8. 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 for read-only access).

20.1.8.9. NFS4ERR_SHARE_DENIED (Error Code 10015)

An attempt to OPEN a file with a share reservation has failed because of a share conflict.

20.1.9. Reclaim Errors

These errors relate to the process of reclaiming locks after a server restart.

20.1.9.1. NFS4ERR_COMPLETE_ALREADY (Error Code 10054)

The client previously sent a successful RECLAIM_COMPLETE operation specifying the same scope, whether that scope is global or for the same file system in the case of a per-fs RECLAIM_COMPLETE. An additional RECLAIM_COMPLETE operation is not necessary and results in this error.

20.1.9.2. NFS4ERR_GRACE (Error Code 10013)

This error is returned when the server is in its grace period with regard to the file system object for which the lock was requested. In this situation, a non-reclaim locking request cannot be granted. This can occur because either:

* The server does not have sufficient information about locks that might be potentially reclaimed to determine whether the lock could be granted.

* The request is made by a client responsible for reclaiming its locks that has not yet done the appropriate RECLAIM_COMPLETE operation, allowing it to proceed to obtain new locks.

In the case of a per-fs grace period, there may be clients (i.e., those currently using the destination file system) who might be unaware of the circumstances resulting in the initiation of the grace period. Such clients need to periodically retry the request until the grace period is over, just as other clients do.

20.1.9.3. 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 occurs in any of the following situations:

* There is no active grace period applying to the file system object for which the request was made.

* The client making the request has no current role in reclaiming locks.

* Previous operations have created a situation in which the server is not able to determine that a reclaim-interfering edge condition does not exist.

20.1.9.4. NFS4ERR_RECLAIM_BAD (Error Code 10034)

The server has determined that a reclaim attempted by the client is not valid, i.e., the lock specified as being reclaimed could not possibly have existed before the server restart or file system migration event. A server is not obliged to make this determination and will typically rely on the client to only reclaim locks that the client was granted prior to restart. However, when a server does have reliable information to enable it to make this determination, this error indicates that the reclaim has been rejected as invalid. This is as opposed to the error NFS4ERR_RECLAIM_CONFLICT (see Section 20.1.9.5) where the server can only determine that there has been an invalid reclaim, but cannot determine which request is invalid.

20.1.9.5. NFS4ERR_RECLAIM_CONFLICT (Error Code 10035)

The reclaim attempted by the client has encountered a conflict and cannot be satisfied. This 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. See also Section 20.1.9.4 for the related error, NFS4ERR_RECLAIM_BAD.

20.1.10. pNFS Errors

This section deals with pNFS-related errors including those that are associated with using NFSv4.1 to communicate with a data server.

20.1.10.1. NFS4ERR_BADIOMODE (Error Code 10049)

An invalid or inappropriate layout iomode was specified. For example an inappropriate layout iomode, suppose a client's LAYOUTGET operation specified an iomode of LAYOUTIOMODE4_RW, and the server is neither able nor willing to let the client send write requests to data servers; the server can reply with NFS4ERR_BADIOMODE. The client would then send another LAYOUTGET with an iomode of LAYOUTIOMODE4_READ.

20.1.10.2. 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.

20.1.10.3. 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).

20.1.10.4. NFS4ERR_LAYOUTUNAVAILABLE (Error Code 10059)

Returned when layouts are not available for the current file system or the particular specified file.

20.1.10.5. 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.

20.1.10.6. 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 18.4.4.

20.1.10.7. 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.

20.1.10.8. 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 17.5.5.3.3.

20.1.10.9. NFS4ERR_UNKNOWN_LAYOUTTYPE (Error Code 10062)

The client has specified a layout type that is not supported by the server.

20.1.11. Session Use Errors

This section deals with errors encountered when using sessions, that is, errors encountered when a request uses a Sequence (i.e., either SEQUENCE or CB_SEQUENCE) operation.

20.1.11.1. NFS4ERR_BADSESSION (Error Code 10052)

The specified session ID is unknown to the server to which the operation is addressed.

20.1.11.2. 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.

20.1.11.3. NFS4ERR_BAD_HIGH_SLOT (Error Code 10077)

The highest_slot argument in a Sequence operation exceeds the replier’s enforced highest_slotid. Also returned when the rsa_target_highest_slotid argument in a CB_RECALL_SLOT operation exceeds maximum enforced slot ID of the session’s fore channel.

20.1.11.4. 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 23.46).

20.1.11.5. NFS4ERR_DEADSESSION (Error Code 10078)

The specified session is a persistent session that 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).

20.1.11.6. 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.

20.1.11.7. 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. See Section 7.6.1.3.1.

20.1.11.8. NFS4ERR_SEQ_MISORDERED (Error Code 10063)

The requester sent a Sequence operation with an invalid sequence ID.

20.1.12. Session Management Errors

This section deals with errors associated with requests used in session management.

20.1.12.1. 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.

20.1.12.2. NFS4ERR_BAD_SESSION_DIGEST (Error Code 10051)

The digest used in a SET_SSV request is not valid.

20.1.13. Client Management Errors

This section deals with errors associated with requests used to create and manage client IDs.

20.1.13.1. 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.

20.1.13.2. NFS4ERR_CLID_INUSE (Error Code 10017)

While processing an EXCHANGE_ID operation, the server was presented with a co_ownerid field that matches an existing client with valid leased state, but the principal sending the EXCHANGE_ID operation differs from the principal that established the existing client. This indicates a collision (most likely due to chance) 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).

20.1.13.3. NFS4ERR_ENCR_ALG_UNSUPP (Error Code 10079)

An EXCHANGE_ID was sent that specified state protection via SSV, and where the set of encryption algorithms presented by the client did not include any supported by the server.

20.1.13.4. NFS4ERR_HASH_ALG_UNSUPP (Error Code 10072)

An EXCHANGE_ID was sent that specified state protection via SSV, and where the set of hashing algorithms presented by the client did not include any supported by the server.

20.1.13.5. 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.

20.1.14. Delegation Errors

This section deals with errors associated with requesting and returning delegations.

20.1.14.1. NFS4ERR_DELEG_ALREADY_WANTED (Error Code 10056)

The client has requested a delegation when it had already registered that it wants that same delegation.

20.1.14.2. NFS4ERR_DIRDELEG_UNAVAIL (Error Code 10084)

This error is returned when the server is unable or unwilling to provide a requested directory delegation.

20.1.14.3. NFS4ERR_RECALLCONFLICT (Error Code 10061)

A recallable object (i.e., a layout or delegation) is unavailable due to a conflicting recall operation that is currently in progress for that object.

20.1.14.4. NFS4ERR_REJECT_DELEG (Error Code 10085)

The callback operation invoked to deal with a new delegation has rejected it.

20.1.15. Attribute Handling Errors

This section deals with errors specific to attribute handling within NFSv4.

20.1.15.1. NFS4ERR_ATTRNOTSUPP (Error Code 10032)

An attribute specified is not supported by the server. This error MUST NOT be returned by the GETATTR operation.

20.1.15.2. NFS4ERR_BADOWNER (Error Code 10039)

This error is 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.

20.1.15.3. 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.

20.1.15.4. 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.

20.1.16. Obsoleted Errors

These errors MUST NOT be generated by any NFSv4.1 operation. This can be for a number of reasons.

* The function provided by the error has been superseded by one of the status bits returned by the SEQUENCE operation.

* The new session structure and associated change in locking have made the error unnecessary.

* There has been a restructuring of some errors for NFSv4.1 that resulted in the elimination of certain errors.

20.1.16.1. 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.

20.1.16.2. 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 23.46).

20.1.16.3. NFS4ERR_NXIO (Error Code 6)

I/O error. No such device or address. This error is for errors involving block and character device access, but because NFSv4.1 is not a device-access protocol, this error is not applicable.

20.1.16.4. NFS4ERR_RESOURCE (Error Code 10018)

For the processing of the COMPOUND procedure, the server may exhaust available resources and cannot continue processing operations within the COMPOUND procedure. This error will be returned from the server in those instances of resource exhaustion related to the processing of the COMPOUND procedure.

In NFSv4.1, the need for this general error has been eliminated because explicit limits on compound sizes are established when the session is created.

20.1.16.5. 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.

20.1.16.6. 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.

20.2. Operations and Their Valid Errors

This section contains a table that 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:

* The operations that MUST NOT be implemented: OPEN_CONFIRM, RELEASE_LOCKOWNER, RENEW, SETCLIENTID, and SETCLIENTID_CONFIRM.

* All illegal (i.e. undefined) operations.

Operation	Errors
Illegal Ops	NFS4ERR_BADXDR, NFS4ERR_OP_ILLEGAL
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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, 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_INVAL, 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,
		NFS4ERR_REP_TOO_BIG_TO_CACHE,
		NFS4ERR_REQ_TOO_BIG,
		NFS4ERR_RETRY_UNCACHED_REP,
		NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,
		NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS,
		NFS4ERR_UNSAFE_COMPOUND
CREATE_SESSION		NFS4ERR_BADXDR, NFS4ERR_CLID_INUSE,
		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_RETRY_UNCACHED_REP,
		NFS4ERR_SEQ_MISORDERED,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_STALE_CLIENTID,
		NFS4ERR_TOOSMALL,
		NFS4ERR_TOO_MANY_OPS,
		NFS4ERR_WRONG_CRED
DELEGPURGE		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_RETRY_UNCACHED_REP,
		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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
		NFS4ERR_TOO_MANY_OPS,
		NFS4ERR_WRONG_CRED
DESTROY_CLIENTID		NFS4ERR_BADXDR, NFS4ERR_CLIENTID_BUSY,
		NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
		NFS4ERR_NOT_ONLY_OP,
		NFS4ERR_REP_TOO_BIG,
		NFS4ERR_REP_TOO_BIG_TO_CACHE,
		NFS4ERR_REQ_TOO_BIG,
		NFS4ERR_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_STALE_CLIENTID,
		NFS4ERR_TOO_MANY_OPS,
		NFS4ERR_WRONG_CRED
DESTROY_SESSION		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_RETRY_UNCACHED_REP,
		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_INVAL, NFS4ERR_NOENT,
		NFS4ERR_NOT_ONLY_OP, NFS4ERR_NOT_SAME,
		NFS4ERR_REP_TOO_BIG,
		NFS4ERR_REP_TOO_BIG_TO_CACHE,
		NFS4ERR_REQ_TOO_BIG,
		NFS4ERR_RETRY_UNCACHED_REP,
		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_RETRY_UNCACHED_REP, 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_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, 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_RETRY_UNCACHED_REP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
GETATTR	NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_GRACE, 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_RETRY_UNCACHED_REP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONG_TYPE
GETDEVICEINFO	NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_INVAL, NFS4ERR_NOENT, NFS4ERR_NOTSUPP, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG,

	NFS4ERR_REP_TOO_BIG_TO_CACHE,
	NFS4ERR_REQ_TOO_BIG,
	NFS4ERR_RETRY_UNCACHED_REP,
	NFS4ERR_SERVERFAULT, NFS4ERR_TOOSMALL,
	NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNKNOWN_LAYOUTTYPE
GETDEVICELIST	NFS4ERR_BADXDR, NFS4ERR_BAD_COOKIE,
	NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
	NFS4ERR_FHEXPIRED, NFS4ERR_INVAL,
	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_RETRY_UNCACHED_REP,
	NFS4ERR_SERVERFAULT,
	NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNKNOWN_LAYOUTTYPE
GETFH	NFS4ERR_FHEXPIRED, NFS4ERR_MOVED,
	NFS4ERR_NOFILEHANDLE,
	NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_STALE
LAYOUTCOMMIT	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED,
	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_RETRY_UNCACHED_REP,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
	NFS4ERR_SYMLINK, NFS4ERR_TOO_MANY_OPS,
	NFS4ERR_UNKNOWN_LAYOUTTYPE,
	NFS4ERR_WRONG_CRED

LAYOUTGET	NFS4ERR_ACCESS, NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADIOMODE, NFS4ERR_BADLAYOUT, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_DELEG_REVOKED, NFS4ERR_DQUOT, NFS4ERR_EXPIRED, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, NFS4ERR_ROFS, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_SYMLINK, NFS4ERR_TOO_MANY_OPS, NFS4ERR_WRONGSEC, NFS4ERR_WRONG_TYPE, NFS4ERR_XDEV
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_INVAL, 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_RETRY_UNCACHED_REP, 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_INVAL,

	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_RETRY_UNCACHED_REP, 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_INVAL, 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_RETRY_UNCACHED_REP, 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_INVAL, NFS4ERR_IO, NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP,
		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_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_RETRY_UNCACHED_REP,
		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_INVAL,
		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_RETRY_UNCACHED_REP,

	NFS4ERR_ROFS, NFS4ERR_SERVERFAULT, NFS4ERR_SHARE_DENIED, NFS4ERR_STALE, NFS4ERR_SYMLINK, NFS4ERR_TOO_MANY_OPS, NFS4ERR_UNSAFE_COMPOUND, NFS4ERR_WRONGSEC, NFS4ERR_WRONG_TYPE
OPEN_CONFIRM	NFS4ERR_NOTSUPP
OPEN_DOWNGRADE	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_EXPIRED, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP,

	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_RETRY_UNCACHED_REP, 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_RETRY_UNCACHED_REP, 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_INVAL, 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_RETRY_UNCACHED_REP, 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_INVAL, 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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT, NFS4ERR_STALE,
		NFS4ERR_TOOSMALL, NFS4ERR_TOO_MANY_OPS
READLINK		NFS4ERR_ACCESS, 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_RETRY_UNCACHED_REP,
		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_INVAL,
		NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE,
		NFS4ERR_OP_NOT_IN_SESSION,
		NFS4ERR_REP_TOO_BIG,
		NFS4ERR_REP_TOO_BIG_TO_CACHE,
		NFS4ERR_REQ_TOO_BIG,
		NFS4ERR_RETRY_UNCACHED_REP,
		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_INVAL,
		NFS4ERR_IO, NFS4ERR_MOVED,
		NFS4ERR_NAMETOOLONG, 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_RETRY_UNCACHED_REP,
		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_INVAL,
		NFS4ERR_IO, NFS4ERR_MLINK,
		NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG,
		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_RETRY_UNCACHED_REP,
		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_RETRY_UNCACHED_REP,
		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_RETRY_UNCACHED_REP,

	NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
SECINFO	NFS4ERR_ACCESS, NFS4ERR_BADCHAR, NFS4ERR_BADNAME, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_MOVED, NFS4ERR_NAMETOOLONG, 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_RETRY_UNCACHED_REP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
SECINFO_NO_NAME	NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, 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_RETRY_UNCACHED_REP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE, NFS4ERR_TOO_MANY_OPS
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_RETRY_UNCACHED_REP,
		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_RETRY_UNCACHED_REP,
		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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_TOO_MANY_OPS
VERIFY		NFS4ERR_ACCESS, NFS4ERR_ATTRNOTSUPP,
		NFS4ERR_BADCHAR, NFS4ERR_BADXDR,
		NFS4ERR_DEADSESSION, NFS4ERR_DELAY,
		NFS4ERR_FHEXPIRED, NFS4ERR_GRACE,

	NFS4ERR_INVAL, 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_RETRY_UNCACHED_REP,
	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_INVAL, 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_RETRY_UNCACHED_REP,
	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_INVAL, 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_RETRY_UNCACHED_REP,
		NFS4ERR_ROFS, NFS4ERR_SERVERFAULT,
		NFS4ERR_STALE, NFS4ERR_SYMLINK,
		NFS4ERR_TOO_MANY_OPS,
		NFS4ERR_WRONG_TYPE

Table 11: Valid Error Returns for Each Protocol Operation

20.3. Callback Operations and Their Valid Errors

This section contains a table that 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.

Callback Operation	Errors
CB_GETATTR	NFS4ERR_BADHANDLE, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_INVAL, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_REP_TOO_BIG, NFS4ERR_REP_TOO_BIG_TO_CACHE, NFS4ERR_REQ_TOO_BIG, NFS4ERR_RETRY_UNCACHED_REP, NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS,
CB_ILLEGAL	NFS4ERR_BADXDR, NFS4ERR_OP_ILLEGAL
CB_LAYOUTRECALL	NFS4ERR_BADHANDLE, NFS4ERR_BADIOMODE, 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_RETRY_UNCACHED_REP, NFS4ERR_TOO_MANY_OPS, NFS4ERR_UNKNOWN_LAYOUTTYPE, NFS4ERR_WRONG_TYPE
CB_NOTIFY	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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_TOO_MANY_OPS
CB_NOTIFY_DEVICEID		NFS4ERR_BADXDR, 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_RETRY_UNCACHED_REP,
		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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_TOO_MANY_OPS
CB_PUSH_DELEG		NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,
		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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_TOO_MANY_OPS,
		NFS4ERR_WRONG_TYPE
CB_RECALL		NFS4ERR_BADHANDLE, NFS4ERR_BADXDR,
		NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,
		NFS4ERR_OP_NOT_IN_SESSION,
		NFS4ERR_REP_TOO_BIG,
		NFS4ERR_REP_TOO_BIG_TO_CACHE,

		NFS4ERR_REQ_TOO_BIG,
		NFS4ERR_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_TOO_MANY_OPS
CB_RECALL_ANY		NFS4ERR_BADXDR, NFS4ERR_DELAY,
		NFS4ERR_INVAL,
		NFS4ERR_OP_NOT_IN_SESSION,
		NFS4ERR_REP_TOO_BIG,
		NFS4ERR_REP_TOO_BIG_TO_CACHE,
		NFS4ERR_REQ_TOO_BIG,
		NFS4ERR_RETRY_UNCACHED_REP,
		NFS4ERR_TOO_MANY_OPS
CB_RECALLABLE_OBJ_AVAIL		NFS4ERR_BADXDR, 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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_TOO_MANY_OPS
CB_RECALL_SLOT		NFS4ERR_BADXDR,
		NFS4ERR_BAD_HIGH_SLOT, NFS4ERR_DELAY,
		NFS4ERR_OP_NOT_IN_SESSION,
		NFS4ERR_REP_TOO_BIG,
		NFS4ERR_REP_TOO_BIG_TO_CACHE,
		NFS4ERR_REQ_TOO_BIG,
		NFS4ERR_RETRY_UNCACHED_REP,
		NFS4ERR_TOO_MANY_OPS
CB_SEQUENCE		NFS4ERR_BADSESSION, NFS4ERR_BADSLOT,
		NFS4ERR_BADXDR,
		NFS4ERR_BAD_HIGH_SLOT,
		NFS4ERR_CONN_NOT_BOUND_TO_SESSION,
		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
CB_WANTS_CANCELLED		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_RETRY_UNCACHED_REP,
		NFS4ERR_SERVERFAULT,
		NFS4ERR_TOO_MANY_OPS

Table 12: Valid Error Returns for Each Protocol Callback Operation

20.4. 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_DEVICEID, 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		CREATE_SESSION, 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
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
NFS4ERR_DELAY	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_DEVICEID, 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, LAYOUTGET, 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_NOTIFY_DEVICEID, 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, RENAME
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, WANT_DELEGATION, WRITE
NFS4ERR_NOMATCHING_LAYOUT	CB_LAYOUTRECALL
NFS4ERR_NOSPC	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_DEVICEID, 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_DEVICEID, 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_DEVICEID, 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_REP_TOO_BIG_TO_CACHE	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY,

	CB_NOTIFY_DEVICEID, 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_REQ_TOO_BIG	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_DEVICEID, 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	ACCESS, BACKCHANNEL_CTL, BIND_CONN_TO_SESSION, CB_GETATTR, CB_LAYOUTRECALL, CB_NOTIFY, CB_NOTIFY_DEVICEID, 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_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_DEVICEID, 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_DEVICEID, 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 13: Errors and the Operations That Use Them

21. NFSv4.1 Procedures

Both procedures, NULL and COMPOUND, MUST be implemented.

21.1. Procedure 0: NULL - No Operation

21.1.1. ARGUMENTS

```
void;
```

21.1.2. RESULTS

```
void;
```

21.1.3. 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.

21.1.4. ERRORS

None.

21.2. Procedure 1: COMPOUND - Compound Operations

21.2.1. 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<>;
   };
```

21.2.2. 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<>;
   };
```

21.2.3. DESCRIPTION

The COMPOUND procedure is used to combine one or more NFSv4 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 until 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 7.6.4 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. If it does, 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 zero and 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 an operation array that contains operation 2 and the minorversion field has a value of 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 be set to NFS4ERR_OP_ILLEGAL. The COMPOUND procedure's return results will also be NFS4ERR_OP_ILLEGAL.

The definition of the "tag" in the request is left to the implementer. It may be used to summarize the content of the Compound request for the benefit of packet-sniffers and engineers debugging implementations. However, the value of "tag" in the response SHOULD be the same value as provided in the request. This applies to the tag field of the CB_COMPOUND procedure as well.

21.2.3.1. 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.

21.2.3.1.1. Current Filehandle

The current and saved filehandles are used throughout the protocol. Most operations implicitly use the current filehandle as an 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 23.19) 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 23.21) and PUTPUBFH (Section 23.20) 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 operation 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.

21.2.3.1.2. Current Stateid

With NFSv4.1, additions of a current stateid and a saved stateid have been made to the COMPOUND processing environment; this allows for the passing of stateids between operations. There are no changes to the syntax of the protocol, only changes to the semantics of a few operations.

A "current stateid" is the stateid that is associated with the current filehandle. The current stateid may only be changed by an operation that modifies the current filehandle or returns a stateid. If an operation returns a stateid, it MUST set the current stateid to the returned value. If an operation sets the current filehandle but does not return a stateid, the current stateid MUST be set to the

all-zeros special stateid, i.e., (seqid, other) = (0, 0). If an operation uses a stateid as an argument but does not return a stateid, the current stateid MUST NOT be changed. For example, PUTFH, PUTROOTFH, and PUTPUBFH will change the current server state from {ocfh, (osid)} to {cfh, (0, 0)}, while LOCK will change the current state from {cfh, (osid} to {cfh, (nsid)}. Operations like LOOKUP that transform a current filehandle and component name into a new current filehandle will also change the current state to {0, 0}. The SAVEFH and RESTOREFH operations will save and restore both the current filehandle and the current stateid as a set.

The following example is the common case of a simple READ operation with a normal stateid showing that the PUTFH initializes the current stateid to (0, 0). The subsequent READ with stateid (sid1) leaves the current stateid unchanged.

```
PUTFH fh1                              - -> {fh1, (0, 0)}
READ (sid1), 0, 1024      {fh1, (0, 0)} -> {fh1, (0, 0)}
```

Figure 3

This next example performs an OPEN with the root filehandle and, as a result, generates stateid (sid1). The next operation specifies the READ with the argument stateid set such that (seqid, other) are equal to (1, 0), but the current stateid set by the previous operation is actually used when the operation is evaluated. This allows correct interaction with any existing, potentially conflicting, locks.

```
PUTROOTFH                              - -> {fh1, (0, 0)}
OPEN "compA"              {fh1, (0, 0)} -> {fh2, (sid1)}
READ (1, 0), 0, 1024      {fh2, (sid1)} -> {fh2, (sid1)}
CLOSE (1, 0)              {fh2, (sid1)} -> {fh2, (sid2)}
```

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 reuse 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 an 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

21.2.4. ERRORS

COMPOUND will of course return every error that each operation on the fore channel can return (see Table 11). 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:

Error	Notes
NFS4ERR_BADCHAR	The tag argument has a character the replier does not support.
NFS4ERR_BADXDR	
NFS4ERR_DELAY	
NFS4ERR_INVAL	The tag argument is not in UTF-8 encoding.
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 14: COMPOUND Error Returns

22. Operations: REQUIRED, RECOMMENDED, or OPTIONAL

The following tables summarize the operations of the NFSv4.1 protocol and the corresponding designation of REQUIRED, RECOMMENDED, and 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

Operation	REQ, REC, OPT, or MNI	Feature (REQ, REC, or OPT)	Definition
ACCESS	REQ		Section 23.1
BACKCHANNEL_CTL	REQ		Section 23.33
BIND_CONN_TO_SESSION	REQ		Section 23.34
CLOSE	REQ		Section 23.2
COMMIT	REQ		Section 23.3
CREATE	REQ		Section 23.4
CREATE_SESSION	REQ		Section 23.36
DELEGPURGE	OPT	FDELG (REQ)	Section 23.5
DELEGRETURN	OPT	FDELG, DDELG, pNFS (REQ)	Section 23.6
DESTROY_CLIENTID	REQ		Section 23.50
DESTROY_SESSION	REQ		Section 23.37
EXCHANGE_ID	REQ		Section 23.35
FREE_STATEID	REQ		Section 23.38
GETATTR	REQ		Section 23.7
GETDEVICEINFO	OPT	pNFS (REQ)	Section 23.40
GETDEVICELIST	OPT	pNFS (OPT)	Section 23.41
GETFH	REQ		Section 23.8
GET_DIR_DELEGATION	OPT	DDELG (REQ)	Section 23.39
LAYOUTCOMMIT	OPT	pNFS (REQ)	Section 23.42
LAYOUTGET	OPT	pNFS (REQ)	Section 23.43

LAYOUTRETURN	OPT	pNFS (REQ)	Section 23.44
LINK	OPT		Section 23.9
LOCK	REQ		Section 23.10
LOCKT	REQ		Section 23.11
LOCKU	REQ		Section 23.12
LOOKUP	REQ		Section 23.13
LOOKUPP	REQ		Section 23.14
NVERIFY	REQ		Section 23.15
OPEN	REQ		Section 23.16
OPENATTR	OPT		Section 23.17
OPEN_CONFIRM	MNI		N/A
OPEN_DOWNGRADE	REQ		Section 23.18
PUTFH	REQ		Section 23.19
PUTPUBFH	REQ		Section 23.20
PUTROOTFH	REQ		Section 23.21
READ	REQ		Section 23.22
READDIR	REQ		Section 23.23
READLINK	OPT		Section 23.24
RECLAIM_COMPLETE	REQ		Section 23.51
RELEASE_LOCKOWNER	MNI		N/A
REMOVE	REQ		Section 23.25
RENAME	REQ		Section 23.26
RENEW	MNI		N/A
RESTOREFH	REQ		Section 23.27

SAVEFH	REQ		Section 23.28
SECINFO	REQ		Section 23.29
SECINFO_NO_NAME	REC	pNFS file layout (REQ)	Section 23.45, Section 18.13
SEQUENCE	REQ		Section 23.46
SETATTR	REQ		Section 23.30
SETCLIENTID	MNI		N/A
SETCLIENTID_CONFIRM	MNI		N/A
SET_SSV	REQ		Section 23.47
TEST_STATEID	REQ		Section 23.48
VERIFY	REQ		Section 23.31
WANT_DELEGATION	OPT	FDELG (OPT)	Section 23.49
WRITE	REQ		Section 23.32

Table 15: Operations

Operation	REQ, REC, OPT, or MNI	Feature (REQ, REC, or OPT)	Definition
CB_GETATTR	OPT	FDELG (REQ)	Section 25.1
CB_LAYOUTRECALL	OPT	pNFS (REQ)	Section 25.3
CB_NOTIFY	OPT	DDELG (REQ)	Section 25.4
CB_NOTIFY_DEVICEID	OPT	pNFS (OPT)	Section 25.12
CB_NOTIFY_LOCK	OPT		Section 25.11
CB_PUSH_DELEG	OPT	FDELG (OPT)	Section 25.5
CB_RECALL	OPT	FDELG, DDELG, pNFS (REQ)	Section 25.2
CB_RECALL_ANY	OPT	FDELG, DDELG, pNFS (REQ)	Section 25.6
CB_RECALL_SLOT	REQ		Section 25.8
CB_RECALLABLE_OBJ_AVAIL	OPT	DDELG, pNFS (REQ)	Section 25.7
CB_SEQUENCE	REQ		Section 25.9
CB_WANTS_CANCELLED	OPT	FDELG, DDELG, pNFS (REQ)	Section 25.10

Table 16: Callback Operations

23. NFSv4.1 Operations

23.1. Operation 3: ACCESS - Check Access Rights

23.1.1. 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;
};
```

23.1.2. RESULTS

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

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

23.1.3. 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 do 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:

* Whether or not a regular file is executable 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.

* 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 explicit read permissions on executable files in order to access them over NFS is not going to be acceptable to some users and storage administrators. Historically, NFS servers have allowed a user to READ a file if the user has execute access to the file.

As a practical example, the UNIX specification [access_api] 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:

* 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.

* 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. For example, 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 with 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:

* 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.

* 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.

* 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.

* 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 or not the reply will have ACCESS4_EXECUTE set in the access field. The server MUST NOT also examine read permission bits when determining whether or not the reply will have ACCESS4_EXECUTE set in the access field. 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.

* 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.

* 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 in 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 23.16) or READ (Section 23.22) the file. In other words, if the 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.

23.1.4. 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 cannot reliably perform an access check with only current file attributes.

In the NFSv2 protocol, the only reliable way to determine whether an operation was allowed was to try it and see if it succeeded or failed. Using the ACCESS operation in the NFSv4.1 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 that will result in an access failure. The OPEN operation provides a point where the server can verify access to the file object and a method to return that information to the client. The ACCESS operation is still useful for directory operations or for use in the case that 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.

23.2. Operation 4: CLOSE - Close File

23.2.1. ARGUMENTS

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

23.2.2. RESULTS

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

23.2.3. 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 are only those 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 sending 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 combination of principal, security flavor, and, if applicable, GSS mechanism 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 23.35) to send CLOSE.

23.2.4. 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 stateid (the "other" value is zero and the "seqid" field is NFS4_UINT32_MAX, see Section 13.2.3).

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.

23.3. Operation 5: COMMIT - Commit Cached Data

23.3.1. ARGUMENTS

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

23.3.2. RESULTS

```
struct COMMIT4resok {
        verifier4       writeverf;
};

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

23.3.3. 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 one or more WRITE operations that had the "committed" field of their results field set to UNSTABLE4.

The offset specifies the position within the file where the flush is to begin. An offset value of zero means to flush data starting at the beginning of the file. The count specifies the number of bytes of data to flush. If the count is zero, a flush from the 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 operations and the COMMIT. The client does this by comparing the write verifier returned from the initial WRITE operations 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.

23.3.4. IMPLEMENTATION

The COMMIT operation is similar in operation and semantics to the POSIX fsync() [fsync] system interface 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. When using pNFS, if a WRITE returned UNSTABLE4 and NFL4_UFLG_COMMIT_THRU_MDS is not set, then the client MUST COMMIT to the data server. The COMMIT

may result in flushing the data but not the metadata. In this case, the metadata MUST be flushed with a subsequent LAYOUTCOMMIT to the metadata server. A complete set of pNFS rules for flushing data and metadata is described in Section 18.9 As in the case of fsync(), it may be that there is some modified data or no modified data to synchronize. The data may have been synchronized by the server's normal periodic buffer synchronization activity. COMMIT should return NFS4_OK, unless there has been an unexpected error.

COMMIT differs from fsync() in that it is possible for the client to flush a range of the file (most likely triggered by a buffer-reclamation scheme on the client before the 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 zero and count zero, it should do the equivalent of applying fsync() to the entire 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 second 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 zero and count of zero, 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 (via the WRITE operation) by the client with the "committed" field in the result of WRITE 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 the "committed" field in the result 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 differs from that 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 implementer. If there is only one buffer of 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 operations 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 repeatably 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 the former systems, the virtual memory systems, the virtual memory system will need to be modified instead of the buffer cache.

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

23.4.1. 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;
};
```

23.4.2. 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;
};
```

23.4.3. 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 filehandle 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 zero, or if objname does not obey the UTF-8 definition, the error NFS4ERR_INVAL will be returned.

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

The createattrs specifies the initial set of attributes for the object. The set of attributes may include any writable attribute valid for the object type. When the operation is successful, the server will return to the client an attribute mask signifying which attributes were successfully set for the object.

If createattrs includes neither the owner attribute nor an ACL with an ACE for the owner, and if the server's file system both supports and requires an owner attribute (or an owner ACE), then the server MUST derive the owner (or the owner ACE). This would typically be from the principal indicated in the RPC credentials of the call, but the server's operating environment or file system semantics may dictate other methods of derivation. Similarly, if createattrs includes neither the group attribute nor a group ACE, and if the server's file system both supports and requires the notion of a group attribute (or group ACE), the server MUST derive the group attribute (or the corresponding owner ACE) for the file. This could be from the RPC call's credentials, such as the group principal if the credentials include it (such as with AUTH_SYS), from the group identifier associated with the principal in the credentials (e.g., POSIX systems have a user database [passwd] that has a group identifier for every user identifier), inherited from the directory in which the object is created, or whatever else the server's operating environment or file system semantics dictate. This applies to the OPEN operation too.

Conversely, it is possible that 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.

23.4.4. 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.

23.5. Operation 7: DELEGPURGE - Purge Delegations Awaiting Recovery

23.5.1. ARGUMENTS

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

23.5.2. RESULTS

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

23.5.3. DESCRIPTION

This operation purges all of the delegations awaiting recovery for a given client. This is useful for clients that 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.

The DELEGPURGE operation should be used by clients that record delegation information on stable storage on the client. In this case, after the client recovers all delegations it knows of, it should immediately send a DELEGPURGE operation. 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 might be different. The reason for this is that after sending a request that resulted in a delegation, the client might experience a failure before it both received the delegation and committed the delegation to the client’s stable storage.

The server MAY support DELEGPURGE, but if it does not, it MUST NOT support CLAIM_DELEGATE_PREV and MUST NOT support CLAIM_DELEG_PREV_FH.

23.6. Operation 8: DELEGRETURN - Return Delegation

23.6.1. ARGUMENTS

```
struct DELEGRETURN4args {
        /* CURRENT_FH: delegated object */
        stateid4        deleg_stateid;
};
```

23.6.2. RESULTS

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

23.6.3. DESCRIPTION

The DELEGRETURN operation returns the delegation represented by the current filehandle and stateid.

Delegations may be returned voluntarily (i.e., before the server has recalled them) or when recalled. 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 23.35) to send DELEGRETURN.

23.7. Operation 9: GETATTR - Get Attributes

23.7.1. ARGUMENTS

```
struct GETATTR4args {
        /* CURRENT_FH: object */
        bitmap4         attr_request;
};
```

23.7.2. RESULTS

```
struct GETATTR4resok {
        fattr4          obj_attributes;
};

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

23.7.3. 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 that it was able to return, which will include all attributes requested by the client that 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 that are absent should be treated as having support for a very small set of attributes as described in Section 16.4.1, even if previously, when the file system was present, more attributes were supported.

All servers MUST support the REQUIRED attributes as specified in Section 11.10, for all file systems, with the exception of absent file systems.

On success, the current filehandle retains its value.

23.7.4. IMPLEMENTATION

Suppose there is an OPEN_DELEGATE_WRITE delegation held by another client for the 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:

* The requested attribute values are returned in the response to CB_GETATTR.

* The OPEN_DELEGATE_WRITE delegation is returned.

* The OPEN_DELEGATE_WRITE delegation is revoked.

Unless one of the above happens very quickly, one or more NFS4ERR_DELAY errors will be returned while a delegation is outstanding.

23.8. Operation 10: GETFH - Get Current Filehandle

23.8.1. ARGUMENTS

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

23.8.2. RESULTS

```
struct GETFH4resok {
        nfs_fh4         object;
};

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

23.8.3. DESCRIPTION

This operation returns the current filehandle value.

On success, the current filehandle retains its value.

As described in Section 7.6.4, GETFH is REQUIRED or RECOMMENDED to immediately follow certain operations, and servers are free to reject such operations if the client fails to insert GETFH in the request as REQUIRED or RECOMMENDED. Section 23.16.4.1 provides additional justification for why GETFH MUST follow OPEN.

23.8.4. 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 look up a directory entry and obtain its filehandle, then the following request is needed.

```
PUTFH (directory filehandle)

LOOKUP (entry name)

GETFH
```

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

23.9.1. ARGUMENTS

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

23.9.2. RESULTS

```
struct LINK4resok {
        change_info4    cinfo;
};

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

23.9.3. 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 zero, or if newname does not obey the UTF-8 definition, the error NFS4ERR_INVAL will be returned.

23.9.4. 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 that 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. For example, 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 servers when there is an internal partitioning of a file system that the LINK operation would violate.

On some servers, "." and ".." are illegal values for newname and the error NFS4ERR_BADNAME will be returned if they are specified.

When the current filehandle designates a named attribute directory and the object to be linked (the saved filehandle) is not a named attribute for the same object, the error NFS4ERR_XDEV MUST be returned. When the saved filehandle designates a named attribute and the current filehandle is not the appropriate named attribute directory, the error NFS4ERR_XDEV MUST also be returned.

When the current filehandle designates a named attribute directory and the object to be linked (the saved filehandle) is a named attribute within that directory, the server may return the error NFS4ERR_NOTSUPP.

In the case that newname is already linked to the file represented by the saved filehandle, the server will return NFS4ERR_EXIST.

Note that symbolic links are created with the CREATE operation.

23.10. Operation 12: LOCK - Create Lock

23.10.1. 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;
};
```

23.10.2. 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;
};
```

23.10.3. 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:

* Length is equal to zero.

* 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, the owner, offset, and length of a conflicting lock are returned.

The locker argument specifies the lock-owner that is associated with the LOCK operation. 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 that 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:

* 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.

* The open_seqid and lock_seqid fields of the open_owner field (locker.open_owner.open_seqid and locker.open_owner.lock_seqid).

* 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.

23.10.4. IMPLEMENTATION

If the server is unable to determine the exact offset and length of the conflicting byte-range 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 types 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 access, while a POSIX-compliant system does not.

When the client sends a LOCK operation that corresponds to a range that the lock-owner has locked already (with the same or different lock type), or to a sub-range of such a range, or to a byte-range that 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 sends a LOCK operation that amounts to upgrading (changing from a READ_LT lock to a WRITE_LT lock) or downgrading (changing from WRITE_LT lock to a READ_LT 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 operations from other clients.

When a client holds an OPEN_DELEGATE_WRITE delegation, the client holding that delegation is assured that there are no opens by other clients. Thus, there can be no conflicting LOCK operations from such clients. Therefore, the client may be handling locking requests locally, without doing LOCK operations on the server. If it does that, it must be prepared to update the lock status on the server, by sending appropriate LOCK and LOCKU operations before returning the delegation.

When one or more clients hold OPEN_DELEGATE_READ delegations, any LOCK operation where the server is implementing mandatory locking semantics MUST result in the recall of all such delegations. The LOCK operation may not be granted until all such delegations are returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while the delegation remains outstanding.

23.11. Operation 13: LOCKT - Test for Lock

23.11.1. ARGUMENTS

```
struct LOCKT4args {
        /* CURRENT_FH: file */
        nfs_lock_type4  locktype;
        offset4         offset;
        length4         length;
        lock_owner4     owner;
};
```

23.11.2. RESULTS

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

23.11.3. 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 the 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.

23.11.4. 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 23.10.4, some servers may return NFS4ERR_LOCK_RANGE to certain (otherwise non-conflicting) LOCK operations 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 operation for the same range and lock-owner would fail with NFS4ERR_LOCK_RANGE.

When a client holds an OPEN_DELEGATE_WRITE delegation, it may choose (see Section 23.10.4) to handle LOCK requests locally. In such a case, LOCKT requests will similarly be handled locally.

23.12. Operation 14: LOCKU - Unlock File

23.12.1. ARGUMENTS

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

23.12.2. RESULTS

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

23.12.3. 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 if applicable, the GSS mechanism, combination that sent a LOCK operation 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 23.35) to send LOCKU.

23.12.4. 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 [fcntl] semantics, a client receiving this error should, if it desires support for such operations, simulate the operation using LOCKU on ranges corresponding to locks it actually holds, possibly followed by LOCK operations for the sub-ranges not being unlocked.

When a client holds an OPEN_DELEGATE_WRITE delegation, it may choose (see Section 23.10.4) to handle LOCK requests locally. In such a case, LOCKU operations will similarly be handled locally.

23.13. Operation 15: LOOKUP - Lookup Filename

23.13.1. ARGUMENTS

```
struct LOOKUP4args {
        /* CURRENT_FH: directory */
        component4      objname;
};
```

23.13.2. RESULTS

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

23.13.3. DESCRIPTION

The LOOKUP operation looks up 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.

23.13.4. IMPLEMENTATION

If the client wants to achieve the effect of a multi-component look up, it may construct a COMPOUND request such as (and obtain each filehandle):

```
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 "shared" 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 namespace. 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 look up 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 look up 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.

23.14. Operation 16: LOOKUPP - Lookup Parent Directory

23.14.1. ARGUMENTS

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

23.14.2. RESULTS

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

23.14.3. 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 23.45), 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.

23.14.4. 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 is typically done for the

file system root directory in UNIX).

23.15. Operation 17: NVERIFY - Verify Difference in Attributes

23.15.1. ARGUMENTS

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

23.15.2. RESULTS

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

23.15.3. 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.

23.15.4. 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 an OPTIONAL 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 an attribute is a supported one but is one that is not supported for use in NVERIFY (e.g rdattr_error or any set-only attribute (such as time_modify_set)) is specified, the error NFS4ERR_INVAL is returned to the client.

23.16. Operation 18: OPEN - Open a Regular File

23.16.1. 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:
            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 */
   };

   /*
    * Includes multiple types of operations:
    *
    *    - Non-reclaim operations valid independent of grace period
    *      status.
    *    - Reclaim operations only used during a grace period.
    *    - Reclaim operations only used during a special delegation
    *      recovery period.
    */

   enum open_claim_type4 {

           CLAIM_NULL              = 0,    /* Non-reclaim operation, */
           CLAIM_PREVIOUS          = 1,    /* Reclaim operation --
                                              grace period only. */
           CLAIM_DELEGATE_CUR      = 2,    /* Non-reclaim operation. */
           CLAIM_DELEGATE_PREV     = 3,    /* Reclaim operation --
                                              special delegation
                                              recovery period only. */
           /*
            * Beyond this point, all values are new to v4.1.
            */

           /*
            * Like CLAIM_NULL, but object identified
            * by the current filehandle.
            */
           CLAIM_FH                = 4,    /* Non-reclaim operation. */

           /*
            * Like CLAIM_DELEGATE_CUR, but object identified
            * by current filehandle.
            */
           CLAIM_DELEG_CUR_FH      = 5,    /* Non-reclaim operation. */
```

```
            /*
             * Like CLAIM_DELEGATE_PREV, but object identified
             * by current filehandle.
             */
             CLAIM_DELEG_PREV_FH     = 6     /* Reclaim operation --
                                                special delegation
                                                recovery period
                                                only. */
   };

   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 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;
   };
```

23.16.2. 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_CANCELLED          = 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;
   };
```

23.16.3. 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 of whether or not a file is to be created, 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 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 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 11.12.1.14) 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 23.16.4, 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:

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 17: Required Methods for Exclusive Create

If CREATE_SESSION4_FLAG_PERSIST is set in the results of CREATE_SESSION, the reply cache is persistent (see Section 23.36). If the EXCHGID4_FLAG_USE_PNFS_MDS flag is set in the results from EXCHANGE_ID, the server is a pNFS server (see Section 23.35). If the client attempts to use EXCLUSIVE4 on a persistent session, or a session derived from an EXCHGID4_FLAG_USE_PNFS_MDS client ID, the server MUST return NFS4ERR_INVAL.

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 OPEN4_SHARE_DENY_NONE. In the case that there is an 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 14.7.

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 that 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 associated 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 MAY support CLAIM_DELEGATE_PREV and/or CLAIM_DELEG_PREV_FH (new to NFSv4.1). If it does support either claim type, CREATE_SESSION MUST NOT remove the client's delegation state, and the server MUST support the DELEGPURGE operation.

Table 18

For OPEN requests that reach the server during the grace period, the server returns an error of NFS4ERR_GRACE. The following claim types are exceptions:

* OPEN requests specifying the claim type CLAIM_PREVIOUS are devoted to reclaiming opens after a server restart and are typically only valid during the grace period.

* OPEN requests specifying the claim types CLAIM_DELEGATE_CUR and CLAIM_DELEG_CUR_FH are valid both during and after the grace period. Since the granting of the delegation that they are subordinate to assures that there is no conflict with locks to be reclaimed by other clients, the server need not return NFS4ERR_GRACE when these are received during the grace period.

* OPEN requests specifying the claim types CLAIM_DELEGATE_PREV and CLAIM_DELEG_PREV_FH are valid both during and after the grace period. They must be don during the special delegation recovery period which can overlap a grace period.

For any OPEN request, the server may return an OPEN delegation, which allows further opens and closes to be handled locally on the client as described in Section 15.4. Note that delegation is up to the server to decide. The client should never assume that delegation will or will not be granted in a particular instance. It should always be prepared for either case. A partial exception is the reclaim (CLAIM_PREVIOUS) case, in which a delegation type is claimed. In this case, delegation will always be granted, although the server may specify an immediate recall in the delegation structure.

The rflags returned by a successful OPEN allow the server to return information governing how the open file is to be handled.

* OPEN4_RESULT_CONFIRM is deprecated and MUST NOT be returned by an NFSv4.1 server.

* OPEN4_RESULT_LOCKTYPE_POSIX indicates that the server’s byte-range locking behavior supports the complete set of POSIX locking techniques [fcntl]. From this, the client can choose to manage byte-range locking state in a way to handle a mismatch of byte-range locking management.

* OPEN4_RESULT_PRESERVE_UNLINKED indicates that the NFSv4.1 server will preserve the open file if the client (or any other client) removes the file as long as it remains open. Since the server cannot be aware of files opened using NFSv3 and the client has no information regarding this bit for NFSv4.0 opens, the client, in situations which such opens might exist could find it necessary to do a silly-rename even when this bit is set for all NFSv4.1 OPENs.

  In addition to the basic guarantee above, the server,by returning this bit. promises to preserve the file through any necessary grace period after server restart or file systen migration, thereby giving the client the opportunity to reclaim its open.

  In cases in which a client knows that this flag is returned for all NFSv4.1 opens of a particular file, it can avoid the need for a possible "silly rename" of the file to assure its preservation. It should be noted the possibility of files being open by NFSv3 and NFSv4.0 clients may make the use of silly-rename if necessary. See Section 23.25.4 for further details.

* OPEN4_RESULT_MAY_NOTIFY_LOCK indicates that the server may attempt CB_NOTIFY_LOCK callbacks for locks on this file. This flag is a hint only, and may be safely ignored by the client.

If the component is of zero length, NFS4ERR_INVAL will be returned. The component may also be subject to UTF-8, character support, or other name validity checks. See Section 20.1.7 for further discussion.

When an OPEN is done and the specified open-owner already has the resulting filehandle open, the result is to "OR" together the new share and deny status together with the existing status. In this case, only a single CLOSE need be done, even though multiple OPENs were completed. When such an OPEN is done, checking of share reservations for the new OPEN proceeds normally, with no exception for the existing OPEN held by the same 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, 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 that 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 neither indicating a desire nor a non-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 ond_why 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 OPEN_DELEGATE_WRITE delegations on this file type.

WND4_NOT_SUPP_UPGRADE
: The server does not support atomic upgrade of an OPEN_DELEGATE_READ delegation to an OPEN_DELEGATE_WRITE delegation.

WND4_NOT_SUPP_DOWNGRADE
: The server does not support atomic downgrade of an OPEN_DELEGATE_WRITE delegation to an OPEN_DELEGATE_READ delegation.

WND4_CANCELED
: The client specified OPEN4_SHARE_ACCESS_WANT_CANCEL and now any "want" for this file object is cancelled.

WND4_IS_DIR
: The specified file object is a directory, and the operation is OPEN or WANT_DELEGATION, which do not support delegations on directories.

OPEN4_SHARE_ACCESS_WANT_READ_DELEG, OPEN_SHARE_ACCESS_WANT_WRITE_DELEG, or OPEN_SHARE_ACCESS_WANT_ANY_DELEG mean, respectively, the client wants an OPEN_DELEGATE_READ, OPEN_DELEGATE_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 an OPEN_DELEGATE_READ delegation on a file and requests an OPEN_DELEGATE_WRITE delegation, then the client is requesting atomic upgrade of its OPEN_DELEGATE_READ delegation to an OPEN_DELEGATE_WRITE delegation. If the client has an OPEN_DELEGATE_WRITE delegation on a file and requests an OPEN_DELEGATE_READ delegation, then the client is requesting atomic downgrade to an OPEN_DELEGATE_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 from the delegation type the client currently has, indicates successful upgrade or downgrade. If the server does not support atomic delegation upgrade or downgrade, then ond_why will be set to WND4_NOT_SUPP_UPGRADE or WND4_NOT_SUPP_DOWNGRADE.

OPEN4_SHARE_ACCESS_WANT_NO_DELEG means that the client wants no delegation.

OPEN4_SHARE_ACCESS_WANT_CANCEL means that 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 is set:

* OPEN4_SHARE_ACCESS_WANT_READ_DELEG

* OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG

* OPEN4_SHARE_ACCESS_WANT_ANY_DELEG

If the client specifies OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL, then it wishes to register a "want" for a delegation, in the event the OPEN results do not include a delegation. If so and the server denies the delegation due to insufficient resources, the server MAY later inform the client, via the CB_RECALLABLE_OBJ_AVAIL operation, that the resource limitation condition has eased. The server will tell the client that it intends to send a future CB_RECALLABLE_OBJ_AVAIL operation by setting delegation_type in the results to OPEN_DELEGATE_NONE_EXT, ond_why to WND4_RESOURCE, and ond_server_will_signal_avail set to TRUE. If ond_server_will_signal_avail is set to TRUE, the server MUST later send a CB_RECALLABLE_OBJ_AVAIL operation.

If the client specifies OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_UNCONTENDED, then it wishes to register a "want" for a delegation, in the event the OPEN results do not include a delegation. If so and the server denies the delegation due to contention, the server MAY later inform the client, via the CB_PUSH_DELEG operation, that the contention 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_PUSH_DELEG 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.

23.16.4. 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’s 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 OPTIONAL 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

* SHOULD set the owner of the file to that corresponding to the credential of request’s RPC header.

* 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 value of the expression (share_access & ~OPEN4_SHARE_ACCESS_WANT_DELEG_MASK) MUST be one of OPEN4_SHARE_ACCESS_READ, OPEN4_SHARE_ACCESS_WRITE, or OPEN4_SHARE_ACCESS_BOTH. If not, the server MUST return NFS4ERR_INVAL. The value of share_deny MUST be one of OPEN4_SHARE_DENY_NONE, OPEN4_SHARE_DENY_READ, OPEN4_SHARE_DENY_WRITE, or OPEN4_SHARE_DENY_BOTH. If not, the server MUST return NFS4ERR_INVAL.

Based on the share_access value (OPEN4_SHARE_ACCESS_READ, OPEN4_SHARE_ACCESS_WRITE, or OPEN4_SHARE_ACCESS_BOTH), the client should check that the requester has the proper access rights to perform the specified operation. This would generally be the results of applying the ACL access rules to the file for the current requester. However, just as with the ACCESS operation, the client

should not attempt to second-guess the server's decisions, as access rights may change and may be subject to server administrative controls outside the ACL framework. If the requester's READ or WRITE operation is not authorized (depending on the share_access value), the server MUST return NFS4ERR_ACCESS.

Note that if the client ID was not created with the EXCHGID4_FLAG_BIND_PRINC_STATEID capability 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 send READ and/or WRITE. See Section 23.35.

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 will be returned. 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 to avoid the common implementation practice of renaming an open file to ".nfs<unique value>" instead of removing the file. After the server returns OPEN4_RESULT_PRESERVE_UNLINKED for all NFsv4.1 OPENs and there are no OPENs issued by other protocols to deal with, if a client sends a REMOVE operation that would reduce the file's link count to zero, the server will report a value of zero for the numlinks attribute on the file, when GETATTR can be done on this file.

If another client has a delegation of the file being opened that conflicts with open being done (sometimes depending on 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 an OPEN_DELEGATE_WRITE delegation, any open by a different client will conflict, while for an OPEN_DELEGATE_READ delegation, only opens with one of the following characteristics will be considered conflicting:

* The value of share_access includes the bit OPEN4_SHARE_ACCESS_WRITE.

* The value of share_deny specifies OPEN4_SHARE_DENY_READ or OPEN4_SHARE_DENY_BOTH.

* 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.

23.16.4.1. Warning to Client implementers

OPEN resembles LOOKUP in that it generates a filehandle for the client to use. Unlike LOOKUP though, OPEN creates server state on the filehandle. In normal circumstances, the client can only release this state with a CLOSE operation. CLOSE uses the current filehandle to determine which file to close. Therefore, the client MUST follow every OPEN operation with a GETFH operation in the same COMPOUND procedure. This will supply the client with the filehandle such that CLOSE can be used appropriately.

Simply waiting for the lease on the file to expire is insufficient because the server may maintain the state indefinitely as long as another client does not attempt to make a conflicting access to the same file.

See also Section 7.6.4.

23.17. Operation 19: OPENATTR - Open Named Attribute Directory

23.17.1. ARGUMENTS

```
struct OPENATTR4args {
        /* CURRENT_FH: object */
        bool    createdir;
};
```

23.17.2. RESULTS

```
struct OPENATTR4res {
        /*
         * If status is NFS4_OK,
         *   new CURRENT_FH: named attribute
         *                   directory
         */
        nfsstat4        status;
};
```

23.17.3. 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 filehandle 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. Named attributes or a named attribute directory MUST NOT have their own named attributes.

23.17.4. IMPLEMENTATION

If the server does not support named attributes for file system objects on the file system associated with the current filehandle, an error of NFS4ERR_NOTSUPP will be returned to the client.

23.18. Operation 21: OPEN_DOWNGRADE - Reduce Open File Access

23.18.1. ARGUMENTS

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

23.18.2. RESULTS

```
struct OPEN_DOWNGRADE4resok {
        stateid4        open_stateid;
};

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

23.18.3. 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 expression (share_access & ~OPEN4_SHARE_ACCESS_WANT_DELEG_MASK) 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_INVAL.

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_INVAL.

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:

* 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.

* 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_INVAL. 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.

23.18.4. IMPLEMENTATION

An OPEN_DOWNGRADE operation may make OPEN_DELEGATE_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.

23.19. Operation 22: PUTFH - Set Current Filehandle

23.19.1. ARGUMENTS

```
struct PUTFH4args {
        nfs_fh4         object;
};
```

23.19.2. RESULTS

```
struct PUTFH4res {
        /*
         * If status is NFS4_OK,
         *    new CURRENT_FH: argument to PUTFH
         */
        nfsstat4        status;
};
```

23.19.3. DESCRIPTION

This operation replaces the current filehandle with the filehandle provided as an argument. It 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 21.2.3.1.1 for more details on the current filehandle.

See Section 21.2.3.1.2 for more details on the current stateid.

23.19.4. IMPLEMENTATION

This operation is used in an NFS request to set the context for file accessing operations that follow in the same COMPOUND request.

23.20. Operation 23: PUTPUBFH - Set Public Filehandle

23.20.1. ARGUMENT

```
void;
```

23.20.2. RESULT

```
struct PUTPUBFH4res {
        /*
         * If status is NFS4_OK,
         *   new CURRENT_FH: public fh
         */
        nfsstat4        status;
};
```

23.20.3. DESCRIPTION

This operation replaces the current filehandle with the filehandle that represents the public filehandle of the server's namespace. This filehandle may be different from the "root" filehandle that 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], [RFC2055], and [RFC2224]. 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 directory corresponding to the public filehandle MUST be a descendant of the directory corresponding to the root filehandle.

See Section 21.2.3.1.1 for more details on the current filehandle.

See Section 21.2.3.1.2 for more details on the current stateid.

23.20.4. IMPLEMENTATION

This operation is used 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 pathname provided in the LOOKUP should be evaluated as either an absolute path relative to the server's root or relative to

the public filehandle. [RFC2224]contains further discussion of the functionality. With NFSv4.1, 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 signifies absolute or relative evaluation of an NFS URL, respectively.

Note that there are warnings mentioned in [RFC2224] with respect to the use of absolute evaluation and the restrictions the server may place on that evaluation with respect to how much of its namespace has been made available. These same warnings apply to NFSv4.1. It is likely, therefore, that because of server implementation details, an NFSv3 absolute public filehandle look up may behave differently than an NFSv4.1 absolute resolution.

There is a form of security negotiation as described in [RFC2755]. 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 12 [To be Updated] of the NFSv4-wide security document, currently

23.21. Operation 24: PUTROOTFH - Set Root Filehandle

23.21.1. ARGUMENTS

```
void;
```

23.21.2. RESULTS

```
struct PUTROOTFH4res {
        /*
         * If status is NFS4_OK,
         *   new CURRENT_FH: root fh
         */
        nfsstat4        status;
};
```

23.21.3. DESCRIPTION

This operation replaces the current filehandle with the filehandle that represents the root of the server's namespace. From this filehandle, a LOOKUP operation can locate any other filehandle on the server. This filehandle may be different from the "public" filehandle that may be associated with some other directory on the server.

PUTROOTFH also clears the current stateid.

See Section 21.2.3.1.1 for more details on the current filehandle.

See Section 21.2.3.1.2 for more details on the current stateid.

23.21.4. IMPLEMENTATION

This operation is used in an NFS request to set the context for file accessing operations that follow in the same COMPOUND request.

23.22. Operation 25: READ - Read from File

23.22.1. ARGUMENTS

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

23.22.2. RESULTS

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

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

23.22.3. 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 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 zero and eof is set to TRUE. The READ is subject to access permissions checking.

If the client specifies a count value of zero, the READ succeeds and returns 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 operation, if offset + count is equal to the size of the file), or the READ operation 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 equal to zero, the server MAY allow the READ to be serviced subject to mandatory byte-range locks or the current share deny modes for the file. For a READ with a stateid value of all bits equal to one, the server MAY allow READ operations to bypass locking checks at the server.

On success, the current filehandle retains its value.

23.22.4. IMPLEMENTATION

If the server returns a "short read" (i.e., fewer data than requested and eof is set to FALSE), the client should send another READ to get the remaining data. A server may return less data than requested under several circumstances. The file may have been truncated by another client or perhaps on the server itself, changing the file size from what the requesting client believes to be the case. This would reduce the actual amount of data available to the client. It is possible that the server reduce the transfer size and so return a short read result. Server resource exhaustion may also occur in a short read.

If mandatory byte-range locking is in effect for the file, and if the byte-range corresponding to the data to be read from the file is WRITE_LT locked by an owner not associated with the stateid, the server will return the NFS4ERR_LOCKED error. The client should try to get the appropriate READ_LT via the LOCK operation before re-attempting the READ. When the READ completes, the client should release the byte-range lock via LOCKU.

If another client has an OPEN_DELEGATE_WRITE delegation for the file being read, the delegation must be recalled, and the operation cannot proceed until that delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while the delegation remains outstanding. Normally, delegations will not be recalled as a result of a READ operation since the recall will occur as a result of an earlier OPEN. However, since it is possible for a READ to be done with a special stateid, the server needs to check for this case even though the client should have done an OPEN previously.

23.23. Operation 26: READDIR - Read Directory

23.23.1. ARGUMENTS

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

23.23.2. 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;
   };
```

23.23.3. 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 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 zero (first read of the directory). 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 set of attributes to be returned for each directory entry supplied by the server. As in the case of GETATTR, if this set includes unsupported attributes, they are not included in the returned data and no error results. Because of the possibility of mount points with a directorry, different sets of attributes might be supported for different entries since they might be parts of distinct file systems.

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 11.12.1.12). 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 NOT be returned to the client as part of the READDIR response. To

enable some client environments, the cookie values of zero, 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 one and two SHOULD NOT be used, and for READDIR results, cookie values of zero, one, and two SHOULD NOT be returned.

On success, the current filehandle retains its value.

23.23.4. 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 the 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.

23.24. Operation 27: READLINK - Read Symbolic Link

23.24.1. ARGUMENTS

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

23.24.2. RESULTS

```
struct READLINK4resok {
        linktext4       link;
};

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

23.24.3. DESCRIPTION

READLINK reads the data associated with a symbolic link. Depending on the value of the UTF-8 capability attribute (Section 19.1), 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.

23.24.4. 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 pathname 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.

23.25. Operation 28: REMOVE - Remove File System Object

23.25.1. ARGUMENTS

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

23.25.2. RESULTS

```
struct REMOVE4resok {
        change_info4    cinfo;
};

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

23.25.3. DESCRIPTION

The REMOVE operation removes (deletes) a directory entry which names a file system object from the directory corresponding to the current filehandle. If the entry in the directory was the last reference to the (i.e., there are no other links to that object), the specified object may be destroyed. In addition, as discussed below, the destruction of the object can be delayed by its use as an open file. 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 zero, or if the target does not obey the UTF-8 definition (and the server is enforcing UTF-8 encoding; see Section 19.1), the error NFS4ERR_INVAL will be returned.

On success, the current filehandle retains its value.

23.25.4. IMPLEMENTATION

In two important respects, the REMOVE operation within NFSv4.1 differs from remove operations for earlier versions of NFS:

* NFSv3 required a different operator RMDIR for directory removal and together with REMOVE for non-directory removal. This allowed clients to skip checking the file type when being passed a non-directory delete system call (e.g., unlink() [unlink] in POSIX) to remove a directory, as well as the converse (e.g., a rmdir() on a non-directory) because they knew the server would check the file type. NFSv4.1 REMOVE can be used to delete any directory entry irrespective of its file type.

  The implementer 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 sending a REMOVE operation. Alternatively, the implementer can produce a COMPOUND call that includes a LOOKUP/VERIFY sequence of operations to verify the file type before a REMOVE operation in the same COMPOUND call.

* In order to deal with removal of open files in a manner consistent with local file system semantics, the server has the option of returning the flag OPEN4_RESULT_PRESERVE_UNLINKED, to indicate to the client that the file will be preserved as long has it has an outstanding NFSv4.1 open (see Section 23.16) that returned this flag.

  Regardless of the state of OPEN4_RESULT_PRESERVE_UNLINKED, which controls the continued existence of the object to be deleted, it is unwise for the client to rely on the availability of disk space due to the REMOVE. This is because server file space allocation policies may differ.

If there is an OPEN preventing writing the file (i.e one specifying OPEN4_SHARE_DENY_WRITE or OPEN4_SHARE_DENY_BOTH) the server SHOULD should reject the removal operation (i.e a REMOVE or a removal that occurs as part of RENAME when a file is renamed-over).

In addition, the server MAY implement its own restrictions on removal of a file while it is open. The server might disallow such a removal operation. The conditions that influence the restrictions on removal of a file while it is still open include:

* Whether certain access protocols (i.e., those other than NFS) are holding the file open.

* Whether particular options or policies on the server are enabled.

If a file has an outstanding OPEN and this prevents the removal of the file's directory entry, the error NFS4ERR_FILE_OPEN is returned. This applies to both rejection because of an OPEN preventing witing the file and to that due to the server's own policies.

The case of a removal operation being done by a client that holds a write delegation presents important issues that allows the server to proceed to process the remove without the delegation being recalled, since the client is presumed aware of the existece of OPENs that were sent by it previously or neer sent to the server because the delegtion holder can process OPENs on its own. Altough the server has considrable freedom in determining the OPENs that might prevent a successful removal operation, the server may omit the recall in this case, unless the server is aware of OPENs performed by other access protocols or because options or policies are in effect that might prevent the removal operation independent of the existence of OPENs with deny modes specified.

To deal efficiently with the common case of servers whose policies are such that only the potential existence of OPENs denying READ and/ or WRITE, such situations can be dealt with as follows:

* The client doing the removal operation and holding the delegation needs to make the server aware of the existence of such OPENs without necessarily returning the delegations.

* The server, using the knowledge that he has about any such OPENs proceeds to make its decision using only the OPENs it is aware of, without recalling the delegation.

In cases in which the recall can be dispensed with:

* If the removal operation succeeds, the server may treat the associate write delegation as effectively canceled just as if it had recalled and returned.

* If the removal operation returns successfully, the client can simply forget about the delegation, as if it had been returned.

* If the removal operation returns successfully, the client can client can avoid flushing pending data to be written to the file being removed.

* If the removal operation returns successfully, the client can simply forget about opens the server is unaware of/ These open cannot have any active deny modes specified.

Where the determination above cannot be made definitively because delegations are being held by other clients they MUST be recalled to allow processing of the removal operation to continue. When such a delegation is held, the server has no reliable knowledge of the status of OPENs for that client, so unless there are files opened with the particular deny modes by clients without delegations other than the client doing the removal operation, the determination cannot be made until all such delegations are recalled, and the operation cannot proceed until each sufficient delegation has been returned or revoked to allow the server to make a correct determination.

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 25.4.

In all cases in which delegations are recalled, the server is likely to return one or more NFS4ERR_DELAY errors while delegations remain outstanding.

The handling of the REMOVE operation, when it not rejected as described above involves the following activities:

A): The elimination of the directory entry identified by the REMOVE parameters.

B): The reduction of the link count associated with the file being removed.

C): The deletion of the file data and its eventual re-use to accommodate newly-writtten data.

This last item need only be done when the link count decremented as part of item B reaches the value zero. In addition, this last state transition, will, under certain circumstances. be deferred as long as the file is known by the server to be open. See below for details.

Items A and B are done in sequence but there is no atomicity requirement so that other request may see A having been done without B occurring. The execution of C is often delayed until the link count reaches zero and,in many cases,until the file is no longer open.

* If the reply from the OPEN had the flag OPEN4_RESULT_PRESERVE_UNLINKED set, the server is obligated to maintain access to the removed object (using a filehandle) until the last OPEN is closed.

  This obligation continues across reboots and grace periods, so the file is preserved through the grace period and only considered closed, if it is not reclaimed during the grace period.

  When all of the directory entries within a directory are deleted, it is subject to deletion itself, despite the fact that there still might be files actively used by their filehandles, even though they were once referred by directory entries since removed.

* If a client does not have support for the OPEN4_RESULT_PRESERVE_UNLINKED flag, it will ignore the value and behave as if it were not set.

* If the reply from the OPEN did not have the flag OPEN4_RESULT_PRESERVE_UNLINKED set, the client has the option, as it did in NFSv3 and NFSv4.0, of renaming the file instead of removing it (referred to as "silly rename")

23.26. Operation 29: RENAME - Rename Directory Entry

23.26.1. ARGUMENTS

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

23.26.2. RESULTS

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

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

23.26.3. 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, the target is removed atomically as part of the rename. See Section 23.25.4 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 23.26.4), 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 has a length of zero, or if oldname or newname does not obey the UTF-8 definition in the case of a Unicode- aware file system, the error NFS4ERR_INVAL will be returned.

23.26.4. 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 either of the above 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.

When the restrictions regsarding open files apply to an open file with the target name and the client has a write delegation for that file, delegation recalls can be avoided in the same fashion as described in Section 23.25.4.

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 that 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 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 designates 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 23.26.3 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:

* 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.

* 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 errors while the delegation(s) remains outstanding, although it might not do that if the delegations are returned quickly.

The direct modification resulting from the RENAME operation must be atomic to the client. However, any necessary changes to the link count attributes for the file involved and the eventual deletion of the file’s data in the cas of a file deleted because it is renamed

over need not be atomic. In addition, the delay of deletion until last close functions in the renamed over case behaves as indicated in Section 23.25.4.

The statement "source and target directories MUST reside on the same file system on the server" means that the fsid fields in the attributes for the directories are the same. If they reside on different file systems, the error NFS4ERR_XDEV is returned.

Based on the value of the fh_expire_type attribute for the object, the filehandle may or may not expire on a RENAME. However, server implementers are strongly encouraged to attempt to keep filehandles from expiring in this fashion.

On some servers, the file names "." and ".." are illegal as either oldname or newname, and will result in the error NFS4ERR_BADNAME. In addition, on many servers the case of oldname or newname being an alias for the source directory will be checked for. Such servers will return the error NFS4ERR_INVAL in these cases.

If either of the source or target filehandles are not directories, the server will return NFS4ERR_NOTDIR.

23.27. Operation 31: RESTOREFH - Restore Saved Filehandle

23.27.1. ARGUMENTS

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

23.27.2. RESULTS

```
struct RESTOREFH4res {
        /*
         * If status is NFS4_OK,
         *     new CURRENT_FH: value of saved fh
         */
        nfsstat4        status;
};
```

23.27.3. DESCRIPTION

The RESTOREFH operation sets 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.

See Section 21.2.3.1.1 for more details on the current filehandle.

See Section 21.2.3.1.2 for more details on the current stateid.

23.27.4. IMPLEMENTATION

Operations like OPEN and LOOKUP use the current filehandle to represent a directory and replace it with a new filehandle. Assuming that 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)
```

23.28. Operation 32: SAVEFH - Save Current Filehandle

23.28.1. ARGUMENTS

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

23.28.2. RESULTS

```
struct SAVEFH4res {
        /*
         * If status is NFS4_OK,
         *    new SAVED_FH: value of current fh
         */
        nfsstat4        status;
};
```

23.28.3. DESCRIPTION

The SAVEFH operation saves 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 21.2.3.1.1 for more details on the current filehandle.

See Section 21.2.3.1.2 for more details on the current stateid.

23.28.4. IMPLEMENTATION

23.29. Operation 33: SECINFO - Obtain Available Security

Although this is an existing NFSv4.1 operation and appropriately described in this document, much of the detail regarding the values returned and their role in security negotiation is described in Section 16 of the NFSv4-wide security document, currently [I-D.dnoveck-nfsv4-security]. This adaptation has been necessary since connection characteristics are now an appropriate subject of negotiation, where previously negotiation only concerned the choice of appropriate auth flavors on existing connection.

23.29.1. ARGUMENTS

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

23.29.2. 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;
   };
```

23.29.3. 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 that 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 RFC 5531 [RFC5531]), or RPCSEC_GSS (as defined in RFC 2203 [RFC2203]). 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 RFC 2743 [RFC2743]), the quality of protection (as defined in RFC 2743 [RFC2743]), and the service type (as defined in RFC 2203 [RFC2203]). It is possible for SECINFO to return multiple entries with flavor equal to RPCSEC_GSS with different security triple values.

On success, the current filehandle is consumed (see Section 6.2.1.8), 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 zero, or if the name does not obey the UTF-8 definition (assuming UTF-8 capabilities are enabled; see Section 19.1), the error NFS4ERR_INVAL will be returned.

See Section 16 of [I-D.dnoveck-nfsv4-security] for additional information on the use of SECINFO.

23.29.4. 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 13 for a list of operations that can return NFS4ERR_WRONGSEC. In addition, when READDIR returns attributes, the rdattr_error (Section 11.12.1.12) 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 a 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 6.2.2). 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 represents 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 6.2.1.1 states, a put filehandle operation followed by SAVEFH MUST NOT return NFS4ERR_WRONGSEC. Thus, the RESTOREFH operation, under certain conditions (see Section 6.2.1), 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 rdattr_error attribute, the server will return the NFS4ERR_WRONGSEC error in rdattr_error for the entry.

To resolve an error return of NFS4ERR_WRONGSEC, the client does the following:

* For LOOKUP and OPEN, the client will use SECINFO with the same current filehandle and name as provided in the original LOOKUP or OPEN to enumerate the available security triples.

* For the rdattr_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 rdattr_error attribute.

* 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.

* For LOOKUPP, the client will use SECINFO_NO_NAME { style = SECINFO_STYLE4_PARENT } and provide the filehandle that equals the filehandle originally provided to LOOKUPP.

See Section 26 for a discussion on the recommendations for the security flavor used by SECINFO and SECINFO_NO_NAME.

23.30. Operation 34: SETATTR - Set Attributes

23.30.1. ARGUMENTS

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

23.30.2. RESULTS

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

23.30.3. 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 byte-range locking context that is necessary for SETATTR requests that set the size attribute. Since setting the size attribute modifies the file's data, it has the same locking requirements as a corresponding WRITE. Any SETATTR that sets the size attribute is incompatible with a share

reservation that specifies OPEN4_SHARE_DENY_WRITE. The area between the old end-of-file and the new end-of-file is considered to be modified just as would have been the case had the area in question been specified as the target of WRITE, for the purpose of checking conflicts with byte-range locks, for those cases in which a server is implementing mandatory byte-range locking behavior. A valid stateid SHOULD always be specified. When the file size attribute is not set, the special stateid consisting of all bits equal to zero MAY be passed.

On either success or failure of the operation, the server will return the attrsset bitmask to represent what (if any) attributes were successfully set. The attrsset in the response is a subset of the attrmask field of the obj_attributes field in the argument.

On success, the current filehandle retains its value.

23.30.4. 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 the affected byte-range 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 that is held by a client other than the one doing the SETATTR, the delegation(s) must be recalled, and the operation cannot proceed to actually change an attribute until each such delegation is

returned or revoked. In all cases in which delegations are recalled, the server is likely to return one or more NFS4ERR_DELAY errors while the delegation(s) remains outstanding, although it might not do that if the delegations are returned quickly.

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 the 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 attribute bits not requested to be set by the client. If the attribute masks in the request and reply are equal, the status field in the reply MUST be NFS4_OK.

23.31. Operation 37: VERIFY - Verify Same Attributes

23.31.1. ARGUMENTS

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

23.31.2. RESULTS

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

23.31.3. DESCRIPTION

The VERIFY operation is used to verify that attributes have the value assumed by the client before proceeding with the 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.

23.31.4. 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 OPTIONAL 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 specified is supported but is one fpr which use of VERIFY is inappropriate (e.g. rdattr_error or any set-only attribute (such as time_modify_set)), the error NFS4ERR_INVAL is returned to the client.

23.32. Operation 38: WRITE - Write to File

23.32.1. 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<>;
};
```

23.32.2. RESULTS

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

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

23.32.3. 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 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 zero, the WRITE will succeed and return a count of 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 implementer is free to implement DATA_SYNC4 in the same fashion as FILE_SYNC4, but with a possible performance drop. If stable is UNSTABLE4, the server is free to commit any part of the data and the metadata to stable storage, including all or none, before returning a reply to the client. There is no guarantee whether or when any uncommitted data will subsequently be committed to stable storage. The only guarantees made by the server are that it will not destroy any data without changing the value of 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 19,

* The server MAY commit the data at a stronger level than requested.

* The server MUST commit the data at a level at least as strong as that requested.

stable	committed
UNSTABLE4	FILE_SYNC4, DATA_SYNC4, UNSTABLE4
DATA_SYNC4	FILE_SYNC4, DATA_SYNC4
FILE_SYNC4	FILE_SYNC4

Table 19: Valid Combinations of the Fields Stable in the Request and Committed in the Reply

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 equal to zero, the server MAY allow the WRITE to be serviced subject to mandatory byte-range locks or the current share deny modes for the file. For a WRITE with a stateid value of all bits equal to 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 equal to zero were used.

On success, the current filehandle retains its value.

23.32.4. 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 zero 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.

Some implementations may return NFS4ERR_NOSPC instead of NFS4ERR_DQUOT when a user's quota is exceeded.

In the case that the current filehandle is 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 byte-range locking is in effect for the file, and the corresponding byte-range of the data to be written to the file is READ_LT or WRITE_LT 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_LT lock that overlaps with the byte-range that was to be written. If the stateid's owner has no conflicting READ_LT 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_LT 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_LT lock, either the server attempted to temporarily effectively upgrade this READ_LT lock to a WRITE_LT lock or the server has no upgrade capability. If the server attempted to upgrade the READ_LT 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 to 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.

23.33. Operation 40: BACKCHANNEL_CTL - Backchannel Control

23.33.1. ARGUMENT

```
typedef opaque gsshandle4_t<>;

struct gss_cb_handles4 {
        rpc_gss_service_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 5531 */
case RPCSEC_GSS:
        gss_cb_handles4 cbsp_gss_handles;
};

struct BACKCHANNEL_CTL4args {
        uint32_t                bca_cb_program;
        callback_sec_parms4     bca_sec_parms<>;
};
```

23.33.2. RESULT

```
struct BACKCHANNEL_CTL4res {
        nfsstat4                bcr_status;
};
```

23.33.3. DESCRIPTION

The BACKCHANNEL_CTL operation replaces the backchannel's callback program number and adds (not replaces) RPCSEC_GSS handles 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 23.36).

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.

If an RPCSEC_GSS handle is using the SSV context (see Section 7.9), then because each SSV RPCSEC_GSS handle shares a common SSV GSS context, there are security considerations specific to this situation discussed in Section 7.10.

23.34. Operation 41: BIND_CONN_TO_SESSION - Associate Connection with Session

23.34.1. 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;
};
```

23.34.2. 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;
   };
```

23.34.3. 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 23.35) 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 7.9) 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 that 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 that the client wants to associate with both the fore channel 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 CDFC4_BACK_OR_BOTH, the server MUST return CDFS4_BACK or CDFS4_BOTH.

See the CREATE_SESSION operation (Section 23.36), 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_INVAL.

23.34.4. 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 23.35). 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.

* The client reconnects.

* The client assumes that 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.

* If the request succeeds, this means that 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.

* If the server returns an RPC authentication error, this means that 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.

* 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.

23.35. Operation 42: EXCHANGE_ID - Instantiate Client ID

The EXCHANGE_ID operation exchanges long-hand client and server identifiers (owners) and provides access to a client ID, creating one if necessary. This client ID becomes associated with the connection on which the operation is done, so that it is available when a CREATE_SESSION is done or when the connection is used to issue a request on an existing session associated with the current client.

23.35.1. 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>;
};
```

23.35.2. 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;
   };
```

23.35.3. DESCRIPTION

The client uses the EXCHANGE_ID operation to register a particular instance of that client with the server, as represented by a client_owner4. However, when the client_owner4 has already been registered by other means (e.g., Transparent State Migration), the client may still use EXCHANGE_ID to obtain the client ID assigned previously.

The client ID returned from this operation will be associated with the connection on which the EXCHANGE_ID is received and will serve as a parent object for sessions created by the client on this connection

or to which the connection is bound. As a result of using those sessions to make requests involving the creation of state, that state will become associated with the client ID returned.

In situations in which the registration of the client_owner has not occurred previously, the client ID must first be used, along with the returned eir_sequenceid, in creating an associated session using CREATE_SESSION.

If the flag EXCHGID4_FLAG_CONFIRMED_R is set in the result, eir_flags, then it is an indication that the registration of the client_owner has already occurred and that a further CREATE_SESSION is not needed to confirm it. Of course, subsequent CREATE_SESSION operations may be needed for other reasons.

The value eir_sequenceid is used to establish an initial sequence value associated with the client ID returned. In cases in which a CREATE_SESSION has already been done, there is no need for this value, since sequencing of such request has already been established, and the client has no need for this value and will ignore it.

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 it 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 5.5, the co_ownerid identifies the client, and the co_verifier specifies a particular incarnation of that client. An EXCHANGE_ID sent with a new incarnation of the client will lead to the server removing lock state of the old incarnation. On the other hand, when an EXCHANGE_ID sent with the current incarnation and co_ownerid does not result in an unrelated error, it will potentially update an existing client ID's properties or simply return information about the existing client_id. The latter would happen when this operation is done to the same server using different network addresses as part of creating trunked connections.

A server MUST NOT provide the same client ID to two different incarnations of an eia_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 in connection with network trunking as described in Section 7.5. 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 16.11.9 of the current document).

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 servers claim matching server_owner.so_major_id and eir_server_scope, the client can use the verification techniques discussed in Section 7.5.1 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:

* The capabilities expressed by the following bits, which come from the results of EXCHANGE_ID:

  - EXCHGID4_FLAG_SUPP_MOVED_REFER

  - 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 operations on confirmed client IDs though the server MAY refuse to change them.

* 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 operations.

* For SP4_MACH_CRED or SP4_SSV state protection:

- The list of operations (spo_must_enforce) that MUST use the specified state protection. This list comes from the results of EXCHANGE_ID.

- The list of operations (spo_must_allow) that MAY use the specified state protection. This list comes from the results of EXCHANGE_ID.

Once the client ID is confirmed, these properties cannot be updated by subsequent EXCHANGE_ID requests.

* 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 operations.

  There are REQUIRED and RECOMMENDED relationships among the length of the key of the encryption algorithm ("key length"), the length of the output of hash algorithm ("hash length"), and the length of the SSV ("SSV length").

  o key length MUST be <= hash length. This is because the keys used for the encryption algorithm are actually subkeys derived from the SSV, and the derivation is via the hash algorithm. The selection of an encryption algorithm with a key length that exceeded the length of the output of the hash algorithm would require padding, and thus weaken the use of the encryption algorithm.

  o hash length SHOULD be <= SSV length. This is because the SSV is a key used to derive subkeys via an HMAC, and it is recommended that the key used as input to an HMAC be at least as long as the length of the HMAC’s hash algorithm’s output (see Section 3 of [RFC2104]).

        o key length SHOULD be <= SSV length. This is a transitive result of the above two invariants.

        o key length SHOULD be >= hash length / 2. This is because the subkey derivation is via an HMAC and it is recommended that if the HMAC has to be truncated, it should not be truncated to less than half the hash length (see Section 4 of [RFC2104]).

      - Number of concurrent versions of the SSV the client and server will support (see Section 7.9). This property is represented by spi_window in the EXCHANGE_ID results. The property may be updated by subsequent EXCHANGE_ID operations.

   * 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.

   * 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_INVAL.

   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 that 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 that 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 used when sending the EXCHANGE_ID operation. The situations described in items 6, 7, 8, or 9 of the second numbered list of Section 23.35.4 below 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 that the client is trying to establish a new client ID; it is attempting to trunk data communication to the server (See Section 7.5); or it is attempting to update properties of an unconfirmed client ID. The situations described in items 1, 2, 3, 4, or 5 of the second numbered list of Section 23.35.4 below 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 that 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 the 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 18.1 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 that section. 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 states from unauthorized changes (Section 7.8.3):

* 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.

* SP4_MACH_CRED. If spa_how is SP4_MACH_CRED, then the client MUST send the EXCHANGE_ID operation with RPCSEC_GSS as the security flavor, and with a service of RPC_GSS_SVC_INTEGRITY or RPC_GSS_SVC_PRIVACY. If SP4_MACH_CRED is specified, then 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.

* SP4_SSV. If spa_how is SP4_SSV, then the client MUST send the EXCHANGE_ID operation with RPCSEC_GSS as the security flavor, and with a service of RPC_GSS_SVC_INTEGRITY or RPC_GSS_SVC_PRIVACY. If SP4_SSV is specified, then the client wants to use the SSV to protect its 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 bitmap). 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 for which the server will require SP4_MACH_CRED or SP4_SSV protection. 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 sending with the machine credential or the SSV-based credential, even if the object the 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 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 [RFC4055].

  Due to known weaknesses in id-sha1, it is RECOMMENDED that the client specify at least one algorithm within ssp_hash_algs other than id-sha1.

  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. Because of known the weaknesses in id-sha1, it is RECOMMENDED that it not be selected by the server as long as ssp_hash_algs contains any other supported algorithm.

  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_INVAL.

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 [CSOR_AES]. 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_INVAL. Note that due to previously stated requirements and recommendations on the relationships between key length and hash length, some combinations of RECOMMENDED and REQUIRED encryption algorithm and hash algorithm either SHOULD NOT or MUST NOT be used. Table 20 summarizes the illegal and discouraged combinations.

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_INVAL. The server responds with spi_window, which MUST NOT exceed ssp_window and MUST be at least one. 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 (see Section 7.9). 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; the client can create more handles with another EXCHANGE_ID call.

Because each SSV RPCSEC_GSS handle shares a common SSV GSS context, there are security considerations specific to this situation discussed in Section 7.10.

The seq_window (see Section 5.2.3.1 of [RFC2203]) of each RPCSEC_GSS handle in spi_handle MUST be the same as the seq_window of the RPCSEC_GSS handle used for the credential of the RPC request of which the EXCHANGE_ID operation was sent as a part.

Encryption Algorithm	MUST NOT be combined with	SHOULD NOT be combined with
id-aes128-CBC		id-sha384, id-sha512
id-aes192-CBC	id-sha1	id-sha512
id-aes256-CBC	id-sha1, id-sha224	

Table 20

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.

A possible use for implementation identifiers would be in diagnostic software that extracts 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 how the implementation interacts with its peer. The client and server are not allowed to depend on the peer's manifesting a particular allowed behavior based on an implementation identifier but are required to interoperate as specified elsewhere in the protocol specification.

Because it is possible that some implementations might violate the protocol specification and interpret the identity information, implementations MUST provide facilities to allow the NFSv4 client and server to be configured to set the contents of the nfs_impl_id structures sent to any specified value.

23.35.4. 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 an 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 that 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 the 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 Section 23.35.4, Paragraph 7, Item 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 that 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, 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 that the properties of old_clientid_ret are different than those specified in the current EXCHANGE_ID. Whether or not the properties are being updated, 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 that 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, byte-range locks 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 supports neither the CLAIM_DELEGATE_PREV nor CLAIM_DELEG_PREV_FH claim types, associated delegations should be purged as well; otherwise, delegations are retained and recovery proceeds according to Section 15.2.1.

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 23.36.3.

* 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 23.35.4.

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 a 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 }

   The server returns NFS4ERR_PERM and leaves the client record intact.

23.36. Operation 43: CREATE_SESSION - Create New Session and Confirm Client ID

23.36.1. 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<>;
   };
```

23.36.2. RESULT

```
   struct CREATE_SESSION4resok {
           sessionid4              csr_sessionid;
           sequenceid4             csr_sequence;

           uint32_t                csr_flags;

           channel_attrs4          csr_fore_chan_attrs;
           channel_attrs4          csr_back_chan_attrs;
   };

   union CREATE_SESSION4res switch (nfsstat4 csr_status) {
   case NFS4_OK:
           CREATE_SESSION4resok    csr_resok4;
   default:
           void;
   };
```

23.36.3. 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 it is not, the server MUST return NFS4ERR_NOT_ONLY_OP.

In addition to creating a session, CREATE_SESSION has the following effects:

* 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.

* The connection CREATE_SESSION that 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 with which the new session will be associated. 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 23.36.4). 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. Except where stated otherwise, 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) to the client.

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 17.5.2 it SHOULD support CREATE_SESSION4_FLAG_PERSIST if it supports the layout_hint (Section 11.16.4) 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 connection over which the CREATE_SESSION operation arrived be associated with the session's backchannel in addition to its fore channel. If the server agrees, it sets CREATE_SESSION4_FLAG_CONN_BACK_CHAN in the result field csr_flags. 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 over which the CREATE_SESSION operation arrived is currently in non-RDMA mode but has the capability to operate in RDMA mode, then the client is requesting that the server "step up" to RDMA mode on the connection. If the server agrees, it sets CREATE_SESSION4_FLAG_CONN_RDMA in the result field csr_flags. 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 ([RFC8166]).

csa_fore_chan_attrs, csa_back_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. For each channel, the server

* will reply in ca_headerpadsize with its preferred value, or zero if padding is not in use, and

* 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 7.6.4. For each channel, the server MAY decrease this value but MUST NOT increase it.

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). For each channel, the server MAY decrease this value, but MUST NOT increase it. 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. After the session is created, 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 7.6.4.

ca_maxresponsesize_cached:
Like ca_maxresponsesize, but the maximum size of a reply that will be stored in the reply cache (Section 7.6.1). For each channel, the server MAY decrease this value, but MUST NOT increase it. If, in the reply to CREATE_SESSION, the value of ca_maxresponsesize_cached of a channel is less than the value of ca_maxresponsesize of the same channel, then this is an indication to the requester that it needs to be selective about which replies it directs the replier to cache; for example, large replies from non-idempotent 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 23.46) or CB_SEQUENCE (the csa_cachethis field, see Section 25.9) operations. After the session is created, if a requester sends a request for which the size of the reply would exceed ca_maxresponsesize_cached, the replier will return NFS4ERR_REP_TOO_BIG_TO_CACHE, per the description in Section 7.6.4.

ca_maxoperations:
The maximum number of operations the replier will accept in a COMPOUND or CB_COMPOUND. For the backchannel, the server MUST NOT change the value the client offers. For the fore channel, the server MAY change the requested value. After the session is created, if a 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 zero to ca_maxrequests - 1 inclusive. For the backchannel, the server MUST NOT change the value the client offers. For the fore channel, the server MAY change the requested value.

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). For each channel, the server MAY decrease this value, but MUST NOT increase it.

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 "Context Creation Response - Successful Acceptance", Section 5.2.3.1 of [RFC2203].

The first RPCSEC_GSS handle, gcbp_handle_from_server, is the fore handle the server returned to the client (either in the handle field of data type rpc_gss_init_res or as 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 to which the client will map the RPCSEC_GSS context. The server can immediately use the value of gcbp_handle_from_client in the RPCSEC_GSS credential in callback RPCs. That is, 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 "Elements of the RPCSEC_GSS Security Protocol", Section 5 of [RFC2203]) 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 "RPC Request Header", Section 5.3.1 of [RFC2203]).

If the RPCSEC_GSS handle identified by gcbp_handle_from_server does not exist on the server, the server will return NFS4ERR_NOENT.

Within each element of csa_sec_parms, the fore and back RPCSEC_GSS contexts MUST share the same GSS context and MUST have the same seq_window (see Section 5.2.3.1 of RFC 2203 [RFC2203]). 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 Sections 5 and 5.3.1 of [RFC2203]).

If an RPCSEC_GSS handle is using the SSV context (see Section 7.9), then because each SSV RPCSEC_GSS handle shares a common SSV GSS context, there are security considerations specific to this situation discussed in Section 7.10.

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 replier MUST return NFS4ERR_SEQ_MISORDERED.

23.36.4. 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 need to 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.

* First, this is a reply cache just for detecting and processing CREATE_SESSION requests for a given client ID.

* 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 a 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 a 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 a SEQUENCE. If CREATE_SESSION was originally sent in a COMPOUND that started with a SEQUENCE, then the client SHOULD send a retry in a COMPOUND that starts with a 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.

After the client received a reply to an EXCHANGE_ID operation that contains a new, unconfirmed client ID, the server expects the client to follow with a CREATE_SESSION operation to confirm the client ID. The server expects value of csa_sequenceid in the arguments to that CREATE_SESSION to be to equal the value of the field eir_sequenceid that was returned in results of the EXCHANGE_ID that returned the unconfirmed client ID. Before the server replies to that EXCHANGE_ID operation, it 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 client ID slot thus initialized, the processing of the CREATE_SESSION operation is divided into four phases:

1. Client record look up. 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 23.35.4, 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 is 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.

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, 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.

If the session state is created successfully, the server associates the session with the client ID provided by the client.

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 need to 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 that the fore channel, in order 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 client.

Implementing RPCSEC_GSS callback support requires changes to both the client and server implementations of RPCSEC_GSS. One possible set of changes includes:

* Adding a data structure that wraps the GSS-API context with a reference count.

* New functions to increment and decrement the reference count. If the reference count is decremented to zero, the wrapper data structure and the GSS-API context it refers to would be freed.

* Change RPCSEC_GSS to create the wrapper data structure upon receiving GSS-API context from gss_accept_sec_context() and gss_init_sec_context(). The reference count would be initialized to 1.

* Adding a function to map an existing RPCSEC_GSS handle to a pointer to the wrapper data structure. The reference count would be incremented.

* Adding a function to create a new RPCSEC_GSS handle from a pointer to the wrapper data structure. The reference count would be incremented.

* Replacing calls from RPCSEC_GSS that free GSS-API contexts, with calls to decrement the reference count on the wrapper data structure.

23.37. Operation 44: DESTROY_SESSION - Destroy a Session

23.37.1. ARGUMENT

```
struct DESTROY_SESSION4args {
        sessionid4      dsa_sessionid;
};
```

23.37.2. RESULT

```
struct DESTROY_SESSION4res {
        nfsstat4        dsr_status;
};
```

23.37.3. 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 7.9) 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

* DESTROY_SESSION MUST be the final operation in the COMPOUND request.

* It is advisable to avoid placing DESTROY_SESSION in a COMPOUND request with other state-modifying operations, because the DESTROY_SESSION will destroy the reply cache.

* Because the session and its reply cache are destroyed, a client that retries the request may receive an error in reply to the retry, even though the original request was successful.

If the COMPOUND request starts with SEQUENCE, and if the sessionids specified in SEQUENCE and DESTROY_SESSION are different, then DESTROY_SESSION can appear in any position of the COMPOUND request (except for the first position). The two sessionids can belong to different client IDs.

If the COMPOUND request does not start with SEQUENCE, and if DESTROY_SESSION is not the sole operation, then server MUST return NFS4ERR_NOT_ONLY_OP.

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. If so, then 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 NFSERR_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.

23.38. Operation 45: FREE_STATEID - Free Stateid with No Locks

23.38.1. ARGUMENT

```
struct FREE_STATEID4args {
        stateid4        fsa_stateid;
};
```

23.38.2. RESULT

```
struct FREE_STATEID4res {
        nfsstat4        fsr_status;
};
```

23.38.3. DESCRIPTION

The FREE_STATEID operation is used to free a stateid that no longer has any associated locks (including opens, byte-range locks, delegations, and layouts). This may be because of client LOCKU 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 that had been associated with revoked locks, by sending the FREE_STATEID operation, the client 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 13.4.2.

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.

23.39. Operation 46: GET_DIR_DELEGATION - Get a Directory Delegation

23.39.1. 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;
};
```

23.39.2. 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;
   };
```

23.39.3. 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 refuse 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.

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.

When a client makes a request for a directory delegation while it already holds a directory delegation for that directory (including the case where it has been recalled but not yet returned by the client or revoked by the server), the server MUST reply with the value of gddr_status set to NFS4_OK, the value of gddrnf_status set to GDD4_UNAVAIL, and the value of gddrnf_will_signal_deleg_avail set to FALSE. The delegation the client held before the request remains intact, and its state is unchanged. The current stateid is not changed (see Section 21.2.3.1.2 for a description of the current stateid).

23.39.4. 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, notifications, which can be either synchronous or asynchronous, provide a way to reduce network traffic as well as improve client performance under certain conditions.

The bitmap gdda_notification_types allows the client to request sending of particular notification types and to inform the server of other information relevant to the provision of notifications. For detailed description of the notification, see the appropriate subsection of Section 25.4. The bits, which are defined in Section 15.9.3, can be classified as follows:

* Content update notifications can be requested to allow the client to maintain directory information in accord with that on the server, despite ongoing changes on the server.

  The client can ask for notifications on addition of entries to a directory (by setting the bit NOTIFY4_ADD_ENTRY), notifications on entry removal (NOTIFY4_REMOVE_ENTRY), and renames (NOTIFY4_RENAME_ENTRY).

  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.

  In addition, the client can ask for notification of other sorts of directory change by setting NOTIFY4_CHANGE_COOKIE_VERIFIER. These include changes to cookie verifiers, cookies within the directory, or the order of directory entries.

* The client can ask for notification of attribute changes by setting either NOTIFY4_CHANGE_DIR_ATTRIBUTE (for changes to directory attributes) or NOTIFY4_CHANGE_CHILD_ATTRIBUTE (for change to attributes of objects associated with the entries within the directory)

  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.

* Authorization notifications are use to inform the client of information useful to determine when the local equivalents of LOOKUP, READDIR, and GETATTR can be considered authorized with the use of ACCESS to check for authorization. These notifications are discussed in Section 15.9.9.

  NOTIFY4_CHANGE_AUTH is used to inform the client of a necessary change in the handling of authorization for the local equivalents of LOOKUP and READDIR operations. The structure of this notification is described in Section 25.4.10.

  NOTIFY4_CHANGE_AUTH is used to inform the client of a necessary change in the handling of authorization for the local equivalents of GETATTR operations. The structure of this notification is described in Section 25.4.11.

* In addition to requesting particular types of notifications some of the bits in gdda_notification_types are used as flags to inform the server of notification-related choices that the client can make. These include NOTIFY4_GFLAG_EXTEND and NOTIFY4_CFLAG_ORDER.

The bitmap gddr_notification_types allows the server to indicate that particular notification types will be sent when necessary and to inform the client of other information useful in connection with the provision of notifications. The bits, which are defined in Section 15.9.3, can be classified as follows:

* For bits that have associated notifications, the bit is zero if that notification was not requested and only set to one if that notification was requested and the server undertook to send it when necessary.

These notifications are discussed in Sections 15.9.7 through 15.9.9.

The client MUST use security policy that the directory or its applicable ancestor (Section 6.2) 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 6.2).

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.

23.40. Operation 47: GETDEVICEINFO - Get Device Information

23.40.1. ARGUMENT

```
struct GETDEVICEINFO4args {
        deviceid4       gdia_device_id;
        layouttype4     gdia_layout_type;
        count4          gdia_maxcount;
        bitmap4         gdia_notify_types;
};
```

23.40.2. 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;
};
```

23.40.3. DESCRIPTION

The GETDEVICEINFO operation 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 9.3.7). The numbers of bit positions are listed in the notify_device_type4 enumeration type (Section 25.12). Only two enumerated values of notify_device_type4 currently apply to GETDEVICEINFO: NOTIFY_DEVICEID4_CHANGE and NOTIFY_DEVICEID4_DELETE (see Section 25.12).

The notification bitmap applies only to the specified device ID. If a client sends a GETDEVICEINFO operation 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 send a GETDEVICEINFO operation 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 25.12).

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.

23.40.4. 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.

* CB_NOTIFY_DEVICEID deletes a device ID. If the client believes it has layouts that refer to the device ID, then it is possible that layouts referring to the deleted device ID 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 that any layouts have been revoked, the client must recover from layout revocation as described in Section 17.5.6. If TEST_STATEID indicates that at least one layout has not been revoked, the client should send a GETDEVICEINFO operation on the supposedly deleted device ID to verify that the device ID has been deleted.

  If GETDEVICEINFO indicates that the device ID does not exist, then the client assumes the server is faulty and recovers by sending an EXCHANGE_ID operation. If GETDEVICEINFO indicates that the device ID does exist, then while the server is faulty for sending an erroneous device ID deletion notification, the degree to which it is faulty does not require the client to create a new client 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 GETDEVICEINFO or GETDEVICELIST results are received that indicate that the device ID has been in fact deleted, the device ID should be removed from the client’s cache.

* CB_NOTIFY_DEVICEID indicates that 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.

23.41. Operation 48: GETDEVICELIST - Get All Device Mappings for a File System

23.41.1. 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;
};
```

23.41.2. 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;
};
```

23.41.3. DESCRIPTION

This operation is used by the client to enumerate all of the device IDs that 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 gdia_layout_type. Since this operation might require multiple calls to enumerate all the device IDs (and is thus similar to the READDIR

(Section 23.23) operation), the client also provides gdia_cookie and gdia_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 gdla_cookie to zero. The field gdla_cookieverf MUST be ignored by the server when gdla_cookie is zero. The client provides gdla_maxdevices to limit the number of device IDs in the result. If gdla_maxdevices is zero, the server MUST return NFS4ERR_INVAL. The server MAY return fewer device IDs.

The successful response to the operation will contain the cookie, gdlr_cookie, and the 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.

23.41.4. 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.

23.42. Operation 49: LAYOUTCOMMIT - Commit Writes Made Using a Layout

23.42.1. 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;    /* Unused */
           length4                 loca_length;    /* Unused */
           bool                    loca_reclaim;
           stateid4                loca_stateid;
           newoffset4              loca_last_write_offset;
           newtime4                loca_time_modify;
           layoutupdate4           loca_layoutupdate;
   };
```

23.42.2. 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;
   };
```

23.42.3. DESCRIPTION

The LAYOUTCOMMIT operation commits changes in the layout represented by the current filehandle, client ID (derived from the session ID in the preceding SEQUENCE operation), and stateid. As a layout-independent operation, LAYOUTCOMMIT commits the entire layout; layout type-specific data (loca_layoutupdate) may specify a smaller scope of data that is to be committed (e.g., for the block layout, see [RFC5663]).

The loca_offset and loca_length arguments are no longer used. The client should set both loca_offset and loca_length to 0. The server is to ignore the loca_offset and loca_length arguments. The client MUST hold one or more existing layouts previously granted via LAYOUTGET (Section 23.43), with an iomode of LAYOUTIOMODE4_RW. If layout type-specific data (loca_layoutupdate) restricts the scope of the LAYOUTCOMMIT to less than the entire layout, the client MUST hold one or more existing layouts with an iomode of LAYOUTIOMODE4_RW fully covering the committed byte ranges. For the case where the client does not hold any previously granted layout, the server MUST return the error NFS4ERR_BAD_LAYOUT. Otherwise, where no previously granted layout has an iomode of LAYOUTIOMODE4_RW, the server MUST return the error NFS4ERR_BAD_IOMODE.

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 17.7.4). This type of request may be necessary when the client has uncommitted writes to provisionally allocated byte-ranges of a file that 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 value of the field loca_stateid MUST be a value that the metadata server returned before it restarted. 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 17.5.3).

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. 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 that 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 17.5.4 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 byte-ranges of the original layout have been used and what byte-ranges 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.

23.42.4. IMPLEMENTATION

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 LAYOUTIOMODE4_RW 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.

If loca_reclaim is TRUE, the metadata server is free to either examine or ignore the value in the field loca_stateid. The metadata server implementation might or might not encode in its layout stateid information that allows the metadata server to perform a consistency check on the LAYOUTCOMMIT request.

23.43. Operation 50: LAYOUTGET - Get Layout Information

23.43.1. 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;
   };
```

23.43.2. 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;
   };
```

23.43.3. DESCRIPTION

The LAYOUTGET operation 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 13.4.2.1).

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.

* If loga_length is less than loga_minlength, the metadata server MUST return NFS4ERR_INVAL.

* 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 that it desires a layout of any length with the ending offset of the range no less than the value 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.

* If the sum of loga_offset and loga_minlength exceeds NFS4_UINT64_MAX, and loga_minlength is not NFS4_UINT64_MAX, the error NFS4ERR_INVAL MUST result.

* If the sum of loga_offset and loga_length exceeds NFS4_UINT64_MAX, and loga_length is not NFS4_UINT64_MAX, the error NFS4ERR_INVAL 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 21.

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 21

If loga_minlength is not zero and the metadata server cannot return a layout according to the rules in Table 21, 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 21, then the metadata server MUST return the error NFS4ERR_LAYOUTTRYLATER. Assuming that loga_length is greater than loga_minlength or equal to zero, the metadata server SHOULD return a layout according to the rules in Table 22.

Desired layouts based on loga_length. The rules of Table 21 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
_READ	u64m	MAY be _READ	MUST be <= a_off	SHOULD be u64m
_READ	u64m	MAY be _RW	MUST be <= a_off	SHOULD be u64m
_READ	> 0 and < u64m	MAY be _READ	MUST be <= a_off	SHOULD be >= a_off - layout offset + a_len
_READ	> 0 and < u64m	MAY be _RW	MUST be <= a_off	SHOULD be >= a_off - layout offset + a_len
_READ	0	MAY be _READ	MUST be <= a_off	SHOULD be > a_off - layout offset
_READ	0	MAY be _READ	MUST be <= a_off	SHOULD be > a_off - layout offset
_RW	u64m	MUST be _RW	MUST be <= a_off	SHOULD be u64m
_RW	> 0 and < u64m	MUST be _RW	MUST be <= a_off	SHOULD be >= a_off - layout offset + a_len
_RW	0	MUST be _RW	MUST be <= a_off	SHOULD be > a_off - layout offset

Table 22

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 17.5.3).

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 21 and Table 22 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 21 and Table 22 is equal to the value of the lo_iomode field in each element of logr_layout. As shown in Table 21 and Table 22, 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 LAYOUTIOMODE4_READ requests to LAYOUTIOMODE4_RW 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 21 and Table 22 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 21 states that the metadata server MUST return a layout of length NFS4_UINT64_MAX, then the lo_length field of the last element of logr_layout MUST be NFS4_UINT64_MAX.

* If an applicable rule of Table 21 states that the metadata server MUST NOT return a layout of length NFS4_UINT64_MAX, then the lo_length field of the last element of logr_layout MUST NOT be NFS4_UINT64_MAX.

* If an applicable rule of Table 22 states that the metadata server SHOULD return a layout of length NFS4_UINT64_MAX, then the lo_length field of the last element of logr_layout SHOULD be NFS4_UINT64_MAX.

* When the value of the returned layout length of Table 21 and Table 22 is not NFS4_UINT64_MAX, then the returned layout length is equal to the sum of the lo_length fields of each element of logr_layout.

Once a LAYOUTGET operation returns with logr_return_on_close set to TRUE for a given file, then all subsequent LAYOUTGET requests by that client for the same file and layout type, MUST reply with logr_return_on_close set to TRUE until the client returns all its open state for that file using CLOSE and DELEGRETURN. Note that return_on_close also applies retroactively to all layout segments retrieved by the client for that file and layout type.

After the client has closed all open stateids and returned the delegation stateids for a file for which logr_return_on_close was set to TRUE, the server MUST invalidate all layout segments that were issued to the client for that file. The client MUST NOT attempt to use that layout or the layout stateid.

If the server needs to revoke all open stateids and delegation stateids owned by the client for a file for which logr_return_on_close was set to TRUE, then it MUST also revoke all layout segments of type loga_layout_type that were issued for that file to that client, and take action to fence the access to the DSes

The logr_stateid stateid is returned to the client for use in subsequent layout related operations. See Sections 13.2, 17.5.3, and 17.5.5.2 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 (logr_layout) MUST be equal to the loga_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 neither the requested file nor its containing file system support layouts, 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 17.5.5.2 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.

23.43.4. 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 21.2.3.1.2). 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 21 and Table 22, 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.

* If LAYOUTGET is preceded by OPEN in the same COMPOUND request and the OPEN requests OPEN4_SHARE_ACCESS_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 18.2 for a formal definition), the metadata server might round up the returned layout's length.

* If LAYOUTGET is preceded by OPEN in the same COMPOUND request, and the OPEN requests OPEN4_SHARE_ACCESS_WRITE access and does not truncate the file, 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.

* This strategy is as above, but the OPEN truncates the file. In this case, the 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.

* 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 zero, 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.

* This strategy is 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.

* Again, a process on the client invokes a request to read from offset 10000 for length 50000 (i.e. a range with a starting offset of 10000 and an ending offset of 69999), 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. The client intends to map the request of the process into a series of thirteen READ requests starting at offset 8192, each with length 4096, with a total length of 53248 (which equals 13 * 4096), which fully contains the range that client's process wants to read. Because the value of threshold4_read_iosize is equal to 4096, it is practical and reasonable for the client to use several LAYOUTGET operations 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 zero, 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 sending more LAYOUTGET operations for the remainder of the range.

* An NFS client detects a sequential read pattern, and so sends a LAYOUTGET operation 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. Once the client reads from an offset of the file that represents 50% of the way through the range of the last layout it received, in order to avoid stalling I/O that would wait for a layout, the client sends more operations from an offset of the file that represents 50% of the way through the last layout it received. The client continues to request layouts with byte-ranges that are well in advance of the byte-ranges of recent and/or read requests of processes running on the client.

* This strategy is 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.

* 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 sends 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 send LAYOUTGET operations 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 for which the client does not have device address mappings, 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 23.40.4. 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.

23.44. Operation 51: LAYOUTRETURN - Release Layout Information

23.44.1. 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;
   };
```

23.44.2. 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;
};
```

23.44.3. 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 17.5.5.3.5 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, the 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 Sections 13.2, 17.5.3, and 17.5.5.2 for a further discussion and requirements.

Return of a layout or all layouts does not invalidate the mapping of storage device ID to a storage device address. The mapping remains in effect until specifically changed or deleted via device ID notification callbacks. Of course if there are no remaining layouts that refer to a previously used device ID, the server is free to delete a device ID without a notification callback, which will be the case when notifications are not in effect.

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 23.42) 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. For example, the client uses the seqid on slot 1 of the session, receives 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 17.5.5 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_INVAL 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 the all-zeroes special stateid (see Section 21.2.3.1.2). 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 23.35), 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 23.35) to send LAYOUTRETURN if LAYOUTRETURN's operation code was set in the spo_must_allow result of EXCHANGE_ID.

23.44.4. 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 17.5.3.1 for more details.

Once the client has returned all layouts referring to a particular device ID, the server MAY delete the device ID.

23.45. Operation 52: SECINFO_NO_NAME - Get Security on Unnamed Object

Although this is a new NFSv4.1 operation and appropriately described in this document, much of the detail regarding the values returned and their role in security negotiation is described in Section 16 of the NFSv4-wide security document, currently [I-D.dnoveck-nfsv4-security]. This adaptation has been necessary since connection characteristics are now an appropriate subject of negotiation, where previously negotiation only concerned the choice of appropriate auth flavors on existing connection.

23.45.1. 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;
```

23.45.2. RESULT

```
/* CURRENTFH: consumed if status is NFS4_OK */
typedef SECINFO4res SECINFO_NO_NAME4res;
```

23.45.3. DESCRIPTION

Like the SECINFO operation, SECINFO_NO_NAME is used by the client to obtain a list of valid RPC authentication flavors and transport characteristics 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, where the current filehandle MUST be that of directory (an object of type NF4DIR). 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 returns 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 6.2 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 6.2.1.8), 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 of SECINFO in Section 12.5 of the NFSv4-wide security document.

23.45.4. IMPLEMENTATION

See the discussion on SECINFO in Section 12.5.4.2 of the NFSv4-wide security document, currently [I-D.dnoveck-nfsv4-security].

23.46. Operation 53: SEQUENCE - Supply Per-Procedure Sequencing and Control

23.46.1. ARGUMENT

```
struct SEQUENCE4args {
        sessionid4     sa_sessionid;
        sequenceid4    sa_sequenceid;
        slotid4        sa_slotid;
        slotid4        sa_highest_slotid;
        bool           sa_cachethis;
};
```

23.46.2. 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;
};
```

23.46.3. DESCRIPTION

The SEQUENCE operation is used by the server to implement session request control and the reply cache semantics.

SEQUENCE 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, DESTROY_CLIENTID, 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 23.35), then the server returns NFS4ERR_CONN_NOT_BOUND_TO_SESSION.

The sa_sessionid argument identifies the session to which this request applies. 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 for which the client has a request outstanding; 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 7.6.1 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 7.6.1.3). 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 25.8) 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 or not the client received the callback operation, 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 of the session until a connection is associated with the session's a backchannel. 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 or RPCSEC_GSS handles 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 at least one of the following are true:

* All SSV RPCSEC_GSS handles on the session's backchannel have been destroyed and all non-SSV GSS contexts have expired.

* At least one more SSV RPCSEC_GSS handle has been added to the backchannel.

* The expiration time of at least one non-SSV GSS context of an RPCSEC_GSS handle is beyond the lease period from the current time (relative to the time of when a SEQUENCE response was sent)

SEQ4_STATUS_CB_GSS_CONTEXTS_EXPIRED
When set, indicates all non-SSV GSS contexts and all SSV RPCSEC_GSS handles assigned to the session's backchannel have expired or have been destroyed. This bit remains set on all SEQUENCE replies until at least one non-expired non-SSV GSS context for the session's backchannel has been established or at least one SSV RPCSEC_GSS handle has been assigned to the backchannel.

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 23.38), 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 operation. 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 re-establish 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 16.11.9.2.

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 23.51), every SEQUENCE operation will 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.

* 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.

* If sa_sequenceid and the cached sequence ID are the same, this is a retry, and the server replies with what is recorded in the reply cache. The lease is possibly renewed as described below.

* 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 13.3), except if SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED is returned in sr_status_flags.

23.46.4. 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 received after a server restart might encounter requests performed and recorded in a persistent reply cache before the server restart. In this case, SEQUENCE will be processed successfully, while requests that were not previously performed and recorded 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.

23.47. Operation 54: SET_SSV - Update SSV for a Client ID

23.47.1. ARGUMENT

```
struct ssa_digest_input4 {
        SEQUENCE4args sdi_seqargs;
};

struct SET_SSV4args {
        opaque          ssa_ssv<>;
        opaque          ssa_digest<>;
};
```

23.47.2. 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;
};
```

23.47.3. 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. The SSV is the key used for the SSV GSS mechanism (Section 7.9)

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 23.35); the server returns NFS4ERR_INVAL in that case.

The field ssa_digest is computed as the output of the HMAC [RFC2104] using the subkey derived from the SSV4_SUBKEY_MIC_I2T and current SSV as the key (see Section 7.9 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 23.35, 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 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. 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 7.9).

23.47.4. 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 sending a SET_SSV operation 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 23.34.4.

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 7.8.3). 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.

23.48. Operation 55: TEST_STATEID - Test Stateids for Validity

23.48.1. ARGUMENT

```
struct TEST_STATEID4args {
        stateid4        ts_stateids<>;
};
```

23.48.2. RESULT

```
struct TEST_STATEID4resok {
        nfsstat4        tsr_status_codes<>;
};

union TEST_STATEID4res switch (nfsstat4 tsr_status) {
    case NFS4_OK:
        TEST_STATEID4resok tsr_resok4;
    default:
        void;
};
```

23.48.3. 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:

* SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED

* SEQ4_STATUS_EXPIRED_ADMIN_STATE_REVOKED

* 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 avoids problems with earlier stateids in a COMPOUND request from interfering with the checking of subsequent stateids, as would happen if individual stateids were tested by a series of corresponding by operations in a COMPOUND request.

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:

* There is no check for the type of stateid object, as would be the case for normal use of a stateid.

* There is no reference to the current filehandle.

* 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 valid status values in the returned status_code array are NFS4ERR_OK, NFS4ERR_BAD_STATEID, NFS4ERR_OLD_STATEID, NFS4ERR_EXPIRED, NFS4ERR_ADMIN_REVOKED, and NFS4ERR_DELEG_REVOKED.

23.48.4. IMPLEMENTATION

See Sections 13.2.2 and 13.2.4 for a discussion of stateid structure, lifetime, and validation.

23.49. Operation 56: WANT_DELEGATION - Request Delegation

23.49.1. 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;
};
```

23.49.2. RESULT

```
union WANT_DELEGATION4res switch (nfsstat4 wdr_status) {
case NFS4_OK:
        open_delegation4 wdr_resok4;
default:
        void;
};
```

23.49.3. 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:

* Get a delegation on all types of files except directories.

* 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.

* 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, except as noted below.

* OPEN4_SHARE_ACCESS_WANT_READ_DELEG

* OPEN4_SHARE_ACCESS_WANT_WRITE_DELEG

* OPEN4_SHARE_ACCESS_WANT_ANY_DELEG

* OPEN4_SHARE_ACCESS_WANT_NO_DELEG. Unlike the OPEN operation, this flag SHOULD NOT be set by the client in the arguments to WANT_DELEGATION, and MUST be ignored by the server.

* OPEN4_SHARE_ACCESS_WANT_CANCEL

* OPEN4_SHARE_ACCESS_WANT_SIGNAL_DELEG_WHEN_RESRC_AVAIL

* OPEN4_SHARE_ACCESS_WANT_PUSH_DELEG_WHEN_UNCONTENDED

The handling of the above flags in WANT_DELEGATION is the same as in OPEN. 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_DELEGATION are of data type open_delegation4, which is the same data type as the "delegation" field in the results of the OPEN operation (see Section 23.16.3). 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) & ~OPEN4_SHARE_ACCESS_WANT_NO_DELEG) is zero, then the client is indicating no explicit desire or non-desire for a delegation and the server MUST return NFS4ERR_INVAL.

The client uses the OPEN4_SHARE_ACCESS_WANT_CANCEL 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 23.16. 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 wdr_resok4.od_whynone.ond_why to WND4_WRITE_DELEG_NOT_SUPP_FTYPE.

23.49.4. 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.

23.50. Operation 57: DESTROY_CLIENTID - Destroy a Client ID

23.50.1. ARGUMENT

```
struct DESTROY_CLIENTID4args {
        clientid4       dca_clientid;
};
```

23.50.2. RESULT

```
struct DESTROY_CLIENTID4res {
        nfsstat4        dcr_status;
};
```

23.50.3. DESCRIPTION

The DESTROY_CLIENTID operation destroys the client ID. If there are sessions (both idle and non-idle), opens, locks, delegations, and/or wants (Section 23.49) 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.

23.50.4. 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 that it ever generated the client ID, the server can safely reuse the client ID on a future EXCHANGE_ID operation.

23.51. Operation 58: RECLAIM_COMPLETE - Indicates Reclaims Finished

23.51.1. ARGUMENT

```
struct RECLAIM_COMPLETE4args {
        /*
         * If rca_one_fs TRUE,
         *
         *    CURRENT_FH: object in
         *    file system reclaim is
         *    complete for.
         */
        bool            rca_one_fs;
};
```

23.51.2. RESULTS

```
struct RECLAIM_COMPLETE4res {
        nfsstat4        rcr_status;
};
```

23.51.3. DESCRIPTION

A RECLAIM_COMPLETE operation is used to indicate that the client has reclaimed all of the locking state that it will recover using reclaim-type operation used to re-establish locking state during a server grace period. It is not used in connection with the special delegation recovery period used after client restart.

It does so when it is recovering state due to either a server restart or the migration of a file system to another server. There are two types of RECLAIM_COMPLETE operations:

* 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 has been completed. The current filehandle need not be set in this case.

* 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 the migration of the file system has been completed. Presence of a current filehandle is required when rca_one_fs is set to TRUE. When the current filehandle designates a filehandle in a file system not in the process of migration, the operation returns NFS4_OK and is otherwise ignored.

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 as a result of one of server restart and before it does the first non-reclaim operation that obtains a lock, it MUST send a RECLAIM_COMPLETE with rca_one_fs set to FALSE, 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 migrated file system on a new server, before it sends the first non-reclaim operation that obtains a lock on this new server, it MUST send 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.

It should be noted that there are situations in which a client needs to issue both forms of RECLAIM_COMPLETE. An example is an instance of file system migration in which the file system is migrated to a server for which the client has no clientid. As a result, the client needs to obtain a clientid from the server (incurring the responsibility to do RECLAIM_COMPLETE with rca_one_fs set to FALSE) as well as RECLAIM_COMPLETE with rca_one_fs set to TRUE to complete the per-fs grace period associated with the file system migration. These two may be done in any order as long as all necessary lock reclaims have been done before issuing either of them.

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 of [RFC5661] 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.

23.51.4. 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 able 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 sent, the client effectively acknowledges any locks not yet reclaimed as lost. This allows the server to re-enable the client to recover locks if the occurrence of edge conditions, as described in Section 13.4.3, had caused the server to disable the client's ability to recover locks.

Because previous descriptions of RECLAIM_COMPLETE were not sufficiently explicit about the circumstances in which use of RECLAIM_COMPLETE with rca_one_fs set to TRUE was appropriate, there have been cases in which it has been misused by clients who have issued RECLAIM_COMPLETE with rca_one_fs set to TRUE when it should have not been. There have also been cases in which servers have, in various ways, not responded to such misuse as described above, either ignoring the rca_one_fs setting (treating the operation as a global RECLAIM_COMPLETE) or ignoring the entire operation.

While clients SHOULD NOT misuse this feature, and servers SHOULD respond to such misuse as described above, implementers need to be aware of the following considerations as they make necessary trade-offs between interoperability with existing implementations and proper support for facilities to allow lock recovery in the event of file system migration.

* When servers have no support for becoming the destination server of a file system subject to migration, there is no possibility of a per-fs RECLAIM_COMPLETE being done legitimately, and occurrences of it SHOULD be ignored. However, the negative consequences of accepting such mistaken use are quite limited as long as the client does not issue it before all necessary reclaims are done.

* When a server might become the destination for a file system being migrated, inappropriate use of per-fs RECLAIM_COMPLETE is more concerning. In the case in which the file system designated is not within a per-fs grace period, the per-fs RECLAIM_COMPLETE SHOULD be ignored, with the negative consequences of accepting it being limited, as in the case in which migration is not supported. However, if the server encounters a file system undergoing migration, the operation cannot be accepted as if it were a global RECLAIM_COMPLETE without invalidating its intended use.

23.52. Operation 10044: ILLEGAL - Illegal Operation

23.52.1. ARGUMENTS

```
void;
```

23.52.2. RESULTS

```
struct ILLEGAL4res {
        nfsstat4        status;
};
```

23.52.3. 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.

23.52.4. 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.

24. 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.

24.1. Procedure 0: CB_NULL - No Operation

24.1.1. ARGUMENTS

```
void;
```

24.1.2. RESULTS

```
void;
```

24.1.3. 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.

24.1.4. ERRORS

None.

24.2. Procedure 1: CB_COMPOUND - Compound Operations

24.2.1. 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<>;
   };
```

24.2.2. 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<>;
   };
```

24.2.3. 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 7.6.4 for details.

The minorversion field of the arguments MUST be the same as the minorversion of the COMPOUND procedure used to create 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 equal 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 the COMPOUND procedure (see Section 21.2.3).

Illegal operation codes are handled in the same way as they are handled for the COMPOUND procedure.

24.2.4. 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.

24.2.5. ERRORS

CB_COMPOUND will of course return every error that each operation on the backchannel can return (see Table 12). 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 includes:

Error	Notes
NFS4ERR_BADCHAR	The tag argument has a character the replier does not support.
NFS4ERR_BADXDR	
NFS4ERR_DELAY	
NFS4ERR_INVAL	The tag argument is not in UTF-8 encoding.
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 23: CB_COMPOUND Error Returns

25. NFSv4.1 Callback Operations

25.1. Operation 3: CB_GETATTR - Get Attributes

25.1.1. ARGUMENT

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

25.1.2. RESULT

```
struct CB_GETATTR4resok {
        fattr4  obj_attributes;
};

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

25.1.3. DESCRIPTION

The CB_GETATTR operation is used by the server to obtain the current modified state of a file that has been OPEN_DELEGATE_WRITE delegated. The size and change attributes are the only ones guaranteed to be serviced by the client. See Section 15.4.3 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 an OPEN_DELEGATE_WRITE delegation, an NFS4ERR_BADHANDLE error is returned.

25.1.4. 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).

25.2. Operation 4: CB_RECALL - Recall a Delegation

25.2.1. ARGUMENT

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

25.2.2. RESULT

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

25.2.3. 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 that 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.

25.2.4. 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.

25.3. Operation 5: CB_LAYOUTRECALL - Recall Layout from Client

25.3.1. 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;
   };
```

25.3.2. RESULT

```
   struct CB_LAYOUTRECALL4res {
           nfsstat4        clorr_status;
   };
```

25.3.3. 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 for one of the following: a specific layout of a specific file (LAYOUTRECALL4_FILE), an entire file system ID (LAYOUTRECALL4_FSID), or all file systems

(LAYOUTRECALL4_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 23.44). 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 17.5.5.3.5 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 following: 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 17.5.3 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 Sections 13.2, 17.5.3, and 17.5.5.2 for a further discussion and requirements.

25.3.4. 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 that 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 byte-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 17.5.3.1 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 the 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.

25.4. Operation 6: CB_NOTIFY - Notify Client Using Directory Delegations

25.4.1. ARGUMENT

```
/* Changed entry information.  */
struct notify_entry4 {
        component4      ne_file;
        fattr4          ne_attrs;
};

/* Previous entry information */
struct prev_entry4 {
        notify_entry4   pe_prev_entry;
        /* what READDIR returned for this entry */
        nfs_cookie4     pe_prev_entry_cookie;
};

struct notify_remove4 {
        notify_entry4   nrm_old_entry;
        nfs_cookie4     nrm_old_entry_cookie;
};


/*
 * 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<>;
};
```

cna_stateid designates the associated directory delegation while cna_fh designates the directory for which the delegation is held.

Each element of cna_changes provides a relevant notification with type based on notify_mask and associated data, with associated information in a format that depends on the type, within a nominally opaque array. The elements are processed in sequence.

The bitmap notify_mask contains bits whose indices are derived from the enum notify_type4 defined in Section 15.9.3. The following issues need to be noted:

* Many of bits defined in notify_type4 are flags which do not have an associated notification message, as explained in Section 15.9.3.

  If any such bits are set in notify mask, the callback is invalid and NFS4ERR_INVAL is to be returned.

* If the mask contains and bits in positions not defined as valid elements of the enum notify_type4, the callback is invalid and NFS4ERR_INVAL is to be returned.

* If the bitmask contains no bits set or more that one bit set, the callback is invalid and NFS4ERR_INVAL is to be returned.

When an element is found to be invalid, there is no processing of further elements.

25.4.2. RESULT

```
struct CB_NOTIFY4res {
        nfsstat4    cnr_status;
};
```

25.4.3. DESCRIPTION

The CB_NOTIFY operation is used by the server to send notifications to clients about events related to the maintenance of cached information regarding delegated directories that is used to locally satisfy needs for information normally provided by the use of LOOKUP, READDIR, and GETATTR requests.

The registration of notifications occurs when the delegation is established using GET_DIR_DELEGATION. As a result, these notifications are sent over the backchannel, when certain events occur that affect the directory, the files within it, or the delegation itself. Most notifications are sent asynchronously but the sending of some pf these is initiated promptly, as part of operations making changes, while others are subject to delay and might be sent periodically, some time after the motivating change occurs, as shown in the Modes column in Table 24.

These notifications are used in providing the following functions:

* Notifications relating to the updating of directory contents, as discussed in Section 15.9.7.

* Notifications relating to the updating of attributes for directories and objects within them, as discussed in Section 15.9.8.

* Notifications relating to authorization for use of cached information in locally satisfying requests, as discussed in Section 15.9.9.

The notifications are sent as list of pairs of bitmaps and values with each bitmap consisting of a single bit selected from the enum notify_type, defined in Section 15.9.3 which identifies the specific type of notification being sent. Although the description in Section 9.3.7 is relevant, these bitmaps each have only a single bit set so that the contents of the accompanying opaque array is described by the notification structure associated with that notification type. The individual types are shown in Table 24 with the Modes used in that table defined as follows:

Synch:
: Notifications are sent synchronously in the context of the operation causing the change that the client needs to be informed about.

  When there is a situation in which the notification is to be sent but the client has not requested that type of notification, the delegation is recalled and needs to be returned or revoked before the operation proceeds.

Ordered:
: Notification are sent promptly in the context of the operation causing the change that the client needs to be informed about.

  There is a potential need to order such notifications since processing some notifications in an order different from that in which events occurred can confuse the client. Normally this ordering is provided by putting a number of notifications in the same CB_NOTIFY so that they are processed in order

  In case in which a server has more notifications than can fit in a single CB_COMPOUND request, enforcing appropriate ordering will involve serializing multiple CB_COMPOUND requests. This can involve waiting for responses before sending new callbacks or sending all callbacks associated with a given delegation using the same slot of the session.

When there is a situation in which the notification is to be sent but the client has not requested that type of notification, the delegation is recalled but processing of the operation proceeds without waiting for a client response.

Prompt:
Requests sent promptly as in the case of Ordered notifications but without need for ordering support outside of the context of particular notification types.

The functions of such notifications are either inherently order-independent (e.g., two request to purge a cache are effectively the same as one, independent of the order) or where state is updated protected by an ascending sequence value to prevent difficulties with out-of-order updates.

When there is a situation in which the notification is to be sent but the client has not requested that type of notification, the delegation is recalled but processing of the operation proceeds without waiting for a client response.

Batched:
Sent outside the context of the change, with substantial delays and with no commitment to deliver changes in the order made. Such updates can be sent periodically with sufficient delays between updates to eliminate misordering issues.

When there is a situation in which the notification is to be sent but the client has not requested that type of notification, processing of the operation proceeds normally

Note that for many notifications normally sent batched and described that way in the table below, there are situations in which the client can ask for them be sent using the Ordered approach and the server can undertake to do so. This apples to attribute notifications when the delay of zero is chosen by the client and agreed to by the server,

Name	Function	Mode	Desc.	Disc.
NOTIFY4_ADD_ENTRY	Add dir. entry	Ordered	S. 25.4.4	S. 15.9.7
NOTIFY4_REMOVE_ENTRY	Remove dir. entry	Ordered	S. 25.4.5	S. 15.9.7
NOTIFY4_RENAME_ENTRY	Rename dir. entry	Ordered	S. 25.4.6	S. 15.9.7
NOTIFY4_CHANGE_CHILD_ATTR	Update dir entry attr.	Batched	S. 25.4.7	S. 15.9.8
NOTIFY4_CHANGE_DIR_ATTR	Update dir attr.	Batched	S. 25.4.7	S. 15.9.8
NOTIFY4_CHANGE_COOKIE_VERIFIER	Update dir. contents	Prompt	S. 25.4.8	S. 15.9.7
NOTIFY4_		Prompt	S. 25.4.9	S. 15.9.8
NOTIFY4_		Prompt	S. 25.4.10	S. 15.9.9
NOTIFY4_		Prompt	S. 25.4.11	S. 15.9.9

Table 24: Notification Types

25.4.4. NOTIFY4_ADD_ENTRY

```
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;
};
```

When this notification is sent, the associated data will be in the form of a notify_add4, as defined above.

The server will send information about the new directory entry being created. If the client is known to be interested in the order of the entries, the cookie for that entry in also. 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 the client is known to be interested in the order of the entries, additional information to place the new entry as provided as described in the rest of this paragraph. 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.

25.4.5. NOTIFY4_REMOVE_ENTRY

When this notification is sent, the associated data will be in the form of a notify_remove4, as defined in Section 25.4.1

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.

25.4.6. NOTIFY4_RENAME_ENTRY

```
struct notify_rename4 {
        notify_remove4  nrn_old_entry;
        notify_add4     nrn_new_entry;
};
```

When this notification is sent, the associated data will be in the form of a notify_rename4, as defined above.

The server will send information about both the old entry and the new entry. This includes the 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.

25.4.7. Attribute update Notifications

```
struct notify_attr4 {
        notify_entry4   na_changed_entry;
};
```

When this notification is sent, the associated data will be in the form of a notify_attr44, as defined above.

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 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.

25.4.8. NOTIFY4_CHANGE_COOKIE_VERIFIER

```
struct notify_verifier4 {
        verifier4       nv_old_cookieverf;
        verifier4       nv_new_cookieverf;
};
```

When this notification is sent, the associated data will be in the form of a notify_verifier4, as defined above.

The holder is informed via this notification of a number of potential events:

* When the cookie verifier changes, the client is informed of the new value.

* When there is any change in the cookie assigned to an existing directory entry, the client is informed of the change even if the verifier has remained the same.

  This is necessary because servers are free to not change cookie verifiers in many cases in which a cookie is changed.

* If there is a change in the order of directory entries and the client has previously indicated concern with keeping its order in sync with that of the server by using the NOTIFY4_CFLAG_ORDER flag. The notification is sent even if there is no corresponding change in directory entry cookies.

  In this case as well, the message can be sent without a verifier change.

Upon receiving this notification, the client can invalidate its cookies and re-send a READDIR to get the new set of entries presented in the server's order together with up-to-date cookies.

25.4.9. Attribute Mask Change Notifications

```
struct notify_changeam4 {
        uint32_t        ncam_order;
        bitmap4         ncam_damask;
        bitmap4         ncam_chmask;
};
```

When this notification is sent, the associated data will be in the form of a notify_changem4, as defined above.

This notification is sent whenever the server wishes to change the set of attributes for which updates are to be sent. This includes the case in which one or both the masks is set to indicate an empty attribute mask. This enables to respond to excessive attribute notification traffic without recalling the delegation.

The fields in the notification are used as follows;

* ncam_order is used to protect against the potential effects of notification misordering. The responder need to compare the ncam_order value received to the last such value received and only modify the attribute masks if the new value is greater than the last one received.

* ncam_damask is a bit mask identifying the set of attributes of the delegated directory that will be included in subsequent NOTIFY4_CHANGE_DIR_ATTR notifications.

* ncam_chmask is a bit mask identifying the set of attributes for objects identified in entries within the delegated directory that will be included in subsequent NOTIFY4_CHANGE_CHILD_ATTR notifications.

25.4.10. Change of Authorization Notifications

```
const NCAU_OWNER                = 1;
const NCAU_GROUP                = 2;
const NCAU_OTHERS               = 4;

typedef uint32_t usetmask4;

struct notify_changeau4 {
        uint32_t        ncau_order;
        utf8str_mixed   ncau_owner:
        utf8str_mixed   ncau_group:
        usetmask4       ncau_lookup;
        usetmask4       ncau_readdir;
        usetmask4       ncau_flush;
};
```

When this notification is sent, the associated data will be in the form of a notify_changeu4, as defined above.

This notification is used to inform the client of necessary changes in the authorization of the local equivalents of LOOKUP and READDIR operations.

The fields in the notification are used as described below. Many of the fields are in the form of a usetmask4 which defines the handling of a set of users by including or excluding the directory owner, set of users in the owning group but excluding the directory owner, and all other users, with one bit used for each of those sets.

* ncau_order is a numeric value used to avoid mistakes when notifications are processed in an unexpected order. The value incremented each time such a notification is sent for a given directory delegation and the client can check for ascending values as discussed below.

  For the fields ncau_owner, ncau_group, ncau_lookup, and ncau_readdir, the specified changes are to be used to update the client's state only if the ncau_order is greater than the last one received.

  The field ncau_flush is to be acted on unconditionally, regardless of the value of ncau_order. Such action are not qualified by ordering flushing a cache is an i idempotent operation.

* ncau_owner and ncau_group provide the updated values of the directory owner and directory owning group to be used in classifying requests for authorization and in the caching of results from those authorization checks.

* ncau_lookup provides, for each of the three group of users specified in a usetmask4, whether requests to lookup a file by users in that group can be granted without an explicit ACCESS check.

* ncau_readdir provides, for each of the three group of users specified in a usetmask4, whether requests to read the directory by users in that group can be granted without an explicit ACCESS check.

* ncau_flush indicates, for each of the three group of users specified in a usetmask4, whether the cache of ACCESS check results for users of that class is to be flushed.

  Such cache flushing is necessary when changes in the mode or ACL-related attributes make previous results unreliable and when changes in the owning use or group affect the categorization of users.

25.4.11. Change of GETATTR Authorization Notifications

```
enum ncga_state4 {
        NCGAS_NO                = 1,
        NCGAS_ALLOK             = 2,
        NCGAS_MOSTOK            = 3
};

enum ncga_type4 {
        NCGAT_RESET             = 0,
        NCGAT_SET               = 1,
        NCGAT_ADD               = 2
};

struct notify_changega4 {
        enum ncga_type4 ncga_type;
        uint32_t        ncga_order;
        uint64_t        ncga_addid<>;
};
```

When this notification is sent, the associated data will be in the form of a notify_changega4, as defined above.

The purpose of this notification is to inform the client of the need to make changes in the handling of authorization for local equivalents of the GETATTR operation using cached data.

At any time, the handling is as directed by the client’s current value of its GETATTR authorization state that is represented by one of the three values below.

* NCGAS_NO indicates that explicit ACCESS checks are always necessary.

* NCGAS_ALLOK indicates that explicit ACCESS checks are never necessary.

* NCGAS_SOMEOK that explicit ACCESS checks are necessary only for use of a specific set of files identified by fileid.

Transition between these states are effected depending on the value of the ncga_type field of the notification, as described below.

* NCGAT_RESET causes the state to be set to NCGAS_NO and resets the list of exceptions to empty.

* NCGAT_SET causes the state to be set to NCGAS_ALLOK or NCGAS_SOMEOK, depending on whether the list of fileids to be set as exceptions is empty or not.

* NCGAT_ADD adds to the list fileids to be used as exceptions and set the state to NCGAS_SOMEOK if it is currently NCGAS_ALLOK.

The notify_changega4 contains the following fields:

* ncga_type defines the type of state transition, as described above.

* ncga_order is a numeric value used to prevent problems when notifications are received by the client in an order different from the one in which they are sent.

  Every notification carries the current value maintained on the server. The value is incremented for every notification of type NCGAT_RESET that is sent.

  For notifications of type NCGAT_RESET, the notification is only acted upon if the order value sent is greater than the last one received of that type that is acted upon.

  For notifications of other types, the notification is only acted upon if the order value sent is equal the last one received of type NCGAT_RESET that is acted upon.

* ncga_addid is a set of fileids of object within the directory which are to be added to the list of objects for which the local equivalent of GETATTR requires explicit ACCESS checks

25.5. Operation 7: CB_PUSH_DELEG - Offer Previously Requested Delegation to Client

25.5.1. ARGUMENT

```
struct CB_PUSH_DELEG4args {
        nfs_fh4          cpda_fh;
        open_delegation4 cpda_delegation;

};
```

25.5.2. RESULT

```
struct CB_PUSH_DELEG4res {
        nfsstat4 cpdr_status;
};
```

25.5.3. DESCRIPTION

CB_PUSH_DELEG is used by the server both to 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.

25.5.4. 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.

25.6. Operation 8: CB_RECALL_ANY - Keep Any N Recallable Objects

25.6.1. 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;
};
```

25.6.2. RESULT

```
struct CB_RECALL_ANY4res {
        nfsstat4        crar_status;
};
```

25.6.3. 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, while not optimal, the server is free to recall individual objects to reduce the load.

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 is consistent with a delegation keeping potential operations from being sent to the server. In the case of layouts, while it is true that the usefulness of a layout is indicated by the use of the layout when storage devices receive I/O requests, because there is no mandate that a storage device indicate to the metadata server any past or present use of a layout, the metadata server is not likely to know which layouts are good candidates to recall in response to low resources.

In order to implement an effective reclaim scheme for such objects, the server's knowledge of available resources must be used to determine when objects must be recalled with the clients selecting the actual objects to be returned.

Server implementations may differ in their resource allocation requirements. For example, one server may share resources among all classes of recallable objects, whereas another may use separate resource pools for layouts and for delegations, or further separate resources by types of delegations.

When a given resource pool is over-utilized, the server can 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 OPEN_DELEGATE_READ delegations on non-directory file objects.

RCA4_TYPE_MASK_WDATA_DLG
: The client is to return OPEN_DELEGATE_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.

RCA4_TYPE_MASK_BLK_LAYOUT
: See [RFC5663] for a description.

RCA4_TYPE_MASK_OBJ_LAYOUT_MIN to RCA4_TYPE_MASK_OBJ_LAYOUT_MAX
: See [RFC5664] 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 9.3.13).

When a bit is set in the type mask that corresponds to an undefined type of recallable object, NFS4ERR_INVAL 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_INVAL 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 a 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 if it has not yet received an acknowledgment 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 an 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 cancelled or modified by any subsequent CB_RECALL_ANY for the same type mask. Thus, if the server sends two CB_RECALL_ANYs, 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 to deny 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 sending 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.

25.6.4. IMPLEMENTATION

The client can choose to return any type of object specified by the mask. If a server wishes to limit the use of objects of a specific type, it should only specify that type in the mask it sends. Should the client fail to return requested objects, it is up to the server to handle this situation, typically by sending specific recalls (i.e., sending CB_RECALL operations) 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.

25.7. Operation 9: CB_RECALLABLE_OBJ_AVAIL - Signal Resources for Recallable Objects

25.7.1. ARGUMENT

```
typedef CB_RECALL_ANY4args CB_RECALLABLE_OBJ_AVAIL4args;
```

25.7.2. RESULT

```
struct CB_RECALLABLE_OBJ_AVAIL4res {
        nfsstat4        croa_status;
};
```

25.7.3. 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.

25.8. Operation 10: CB_RECALL_SLOT - Change Flow Control Limits

25.8.1. ARGUMENT

```
struct CB_RECALL_SLOT4args {
        slotid4       rsa_target_highest_slotid;
};
```

25.8.2. RESULT

```
struct CB_RECALL_SLOT4res {
        nfsstat4   rsr_status;
};
```

25.8.3. 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 Send" messages to take the total RDMA credit count to rsa_target_highest_slotid + 1 or below.

25.8.4. 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_BADSLOT.

25.9. Operation 11: CB_SEQUENCE - Supply Backchannel Sequencing and Control

25.9.1. 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<>;
   };
```

25.9.2. 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;
   };
```

25.9.3. 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 that are used to adjust the size of the reply cache. In each CB_COMPOUND request, CB_SEQUENCE MUST appear once and MUST be the first operation. The error NFS4ERR_SEQUENCE_POS MUST be returned when CB_SEQUENCE is found in any position in a CB_COMPOUND beyond the first. If any other operation is in the first position of

CB_COMPOUND, NFS4ERR_OP_NOT_IN_SESSION MUST be returned.

See Section 23.46.3 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 7.6.1.3).

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 7.6.3.

The value of the csa_sequenceid argument relative to the cached sequence ID on the slot falls into one of three cases.

* 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.

* 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.

* 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 a 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. See Section 7.6.1.3 for the conditions when the error NFS4ERR_RETRY_UNCACHED_REP might be returned.

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 7.6.1 for an exception). The latter is the highest slot ID the client would prefer the server use on a future CB_SEQUENCE operation.

25.10. Operation 12: CB_WANTS_CANCELLED - Cancel Pending Delegation Wants

25.10.1. ARGUMENT

```
struct CB_WANTS_CANCELLED4args {
        bool cwca_contended_wants_cancelled;
        bool cwca_resourced_wants_cancelled;
};
```

25.10.2. RESULT

```
struct CB_WANTS_CANCELLED4res {
        nfsstat4        cwcr_status;
};
```

25.10.3. DESCRIPTION

The CB_WANTS_CANCELLED operation is used to notify the client that some or all of the wants it registered for recallable delegations and layouts have been cancelled.

If cwca_contended_wants_cancelled is TRUE, this indicates that 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 that 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.

25.10.4. 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.

25.11. Operation 13: CB_NOTIFY_LOCK - Notify Client of Possible Lock Availability

25.11.1. ARGUMENT

```
struct CB_NOTIFY_LOCK4args {
    nfs_fh4     cnla_fh;
    lock_owner4 cnla_lock_owner;
};
```

25.11.2. RESULT

```
struct CB_NOTIFY_LOCK4res {
        nfsstat4        cnlr_status;
};
```

25.11.3. 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 operation, 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 that has been polling for a blocking byte-range 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" fields of the returned stateids (subject to wraparound) are to be used to select the controlling value of the OPEN4_RESULT_MAY_NOTIFY_LOCK flag.

25.11.4. IMPLEMENTATION

The server MUST NOT grant the byte-range lock to the client unless and until it receives a LOCK operation from the client. Similarly, the client receiving this callback cannot assume that it now has the lock or that a subsequent LOCK operation 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 14.6.

Similarly, the client is not required to implement this callback, and even it does, is still free to ignore it. Therefore, the server MUST NOT assume that the client will act based on the callback.

25.12. Operation 14: CB_NOTIFY_DEVICEID - Notify Client of Device ID Changes

25.12.1. 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;      /* Unused */
   };

   struct CB_NOTIFY_DEVICEID4args {
           notify4 cnda_changes<>;
   };
```

25.12.2. RESULT

```
   struct CB_NOTIFY_DEVICEID4res {
           nfsstat4        cndr_status;
   };
```

25.12.3. 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 9.3.7 for a description of how NFSv4.1 bitmaps work.

As with CB_NOTIFY (Section 25.4.3), 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 17.2.10).

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 contains an unused boolean field, ndc_immediate, which provides no useful information to the client. The client is permitted to finish outstanding I/O that references the previously provided device-ID-to-device-address mapping. Before requesting new layouts, the client needs to replace the previously provided device-ID-to-device-address mapping using a GETDEVINFO operation. All outstanding layouts remain valid after a notification of type NOTIFY_DEVICEID4_CHANGE. If the device-ID-to-device-address mapping changed in an incompatible way, that would invalidate outstanding layouts, the server MUST recall all outstanding layouts and send a NOTIFY_DEVICEID4_DELETE notification instead

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.

25.13. Operation 10044: CB_ILLEGAL - Illegal Callback Operation

25.13.1. ARGUMENT

```
           void;
```

25.13.2. RESULT

```
   /*
    * CB_ILLEGAL: Response for illegal operation numbers
    */
   struct CB_ILLEGAL4res {
           nfsstat4        status;
   };
```

25.13.3. 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 24.2.3 for more details.

The status field of CB_ILLEGAL4res MUST be set to NFS4ERR_OP_ILLEGAL.

25.13.4. 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.

26. Security Considerations

The majority of the Security Considerations relevant to NFS Version 4.1 will appear in the corresponding section of the document devoted to the security of NFS Version 4 as a whole [I-D.dnoveck-nfsv4-security]. Some Security considerations relating to the use of pNFS and other NFSv4.1-specific features will be dealt with in subsections of this section.

26.1. Issues with Inherited Security Considerations Section

The following significant issues need to be addressed in a new Security Considerations section for NFSv4 in whtever document it appears:

* The absence of a threat analysis.

* The lack of attention to the security consequences of transmission of user data in the clear.

* The treatment of AUTH_SYS as OPTIONAL without any discussion of the security consequences of using it.

It is anticipated that such a revised Security Considerations section will appear in the NFSV4-wide security document and that the corresponding section will deal with Security Considerations (including a threat analysis) for NFSv4.1-specific features such as parallel NFS.

26.2. Threat Analysis

Possible additional threats raised by new features in NFSv4.1 will be dealt as follows:

* There do not appear to be additional threats arising from the use of sessions per se. State protection, originally discussed, as part NFSv4.1, in now dealt with in NFSv4-wide security document, rather than in this one.

  Threats related to the persistent storage of session state and locking state are dealt with in Section 26.2.1.

* Threats related to the use of pNFS will be dealt with in Section 26.2.2 and its subsections.

26.2.1. Threat Analysis for Use of Persistent Sessions and Locking State

Locking state can be transferred between two different client-server associations as a result of server restart. This raises the possibility that the transfer might be made inappropriately if a hostile client presents itself as the owner of existing persistently-preserved locking state created before the server restart.

In order to prevent any such misuse, servers that implement persistent sessions or locking state MUST do the following:

* Limit the persistent storage of state to situations in which the client peer owning that state is identifiable (e.g. by the use of client-peer identification together with use of RPC-with-tls).

* Persistently store the client identification together with the locking or reply state being maintained across potential server restart.

* Only restore persisted state when the successor client peer securely identifies itself with identification matching that stored when the state was created.

* Mark such persisted state as having been restored to prevent it being used by a second client peer instance.

It should be noted that there are similar situations in which state created by one client peer might be incorrectly accessed by another with current specifications taking a laxer approach as described in [I-D.dnoveck-nfsv4-security]. For various reasons, it was not possible for these older features to require the level of strictness applied above to persisted locking state:

26.2.2. Threat Analysis for Use of pNFS

The threat analysis is divided based on layout type and coupling mode. Although most layout types only support a single coupling mode, the flexible files layout is designed to support multiple coupling modes with the result that its use with tight and loose coupling need to be dealt with separately.

* For layout types that use RPC for data access and rely on the support of a separate control protocol (i.e. The files layout type described in Section 18 and the flexible files layout described in [RFC8435] when used in tight coupling mode.), the material in Section 17.9.2 provides a general picture of the security issues while the corresponding threat analysis appears in Section 26.2.2.3.

* For layout types that use RPC for data access and have no separate control protocol (i.e. The flexible files layout described in [RFC8435] when used in loose coupling mode.), the material in Section 17.9.3 provides a general picture of the security issues while the corresponding threat analysis appears in Section 26.2.2.2.

* For layout types that do not use RPC for data access and have a separate control protocol (e.g. the blocks layout [RFC5663], the SCSI layout [RFC8154], and the objects layout [RFC5664]), the material in Section 17.9.1 provides a general picture of the security issues while the corresponding threat analysis appears in Section 26.2.2.1.

26.2.2.1. Threat Analysis for pNFS Layout Types Involving non-RPC Data Access

Because data is accessed using non-RPC protocols, there might be no provision for authenticating or even identifying the requester asking for data to be read or written. In such situations, every client whose data access requests are executed has to be trusted to make such requests only in cases in which they would be authorized if pNFS were not used.

Clients that make IO requests in such environments have the ability to access or modify data when they are not authorized to do so. This provides a means whereby hostile clients can compromise data security.

Servers can address such threats by limiting use of non-RPC data access to clients who can be trusted to only make data access requests that would be authorized in the non-pNFS case. The client-peer authentication provided by RPC-with-TLS can be helpful in this regard while the use of such data access without such authentication gives rise likely compromise of necessary data security.

26.2.2.2. Threat Analysis for Layout Types Using NFS versions as Data Access Protocols without Control Protocol Assistance

Although RPC is used for data access in this case, it is not used identify or authenticate the principal making the data access request. Instead, AUTH_SYS is used and the requesting principal is specified by the layout. As a result, the situation, in terms of threats to data integrity, is similar to that described in Section 26.2.2.1.

[Author Aside]: TH Needs to review this section.

26.2.2.3. Threat Analysis for pNFS Layout Types using NFSv4.1 as a Data Access Protocol Together with Control Protocol Assistance

Here we need to look at each of the possible pairs of communicating entities individually:

* For communication between the clients and either the metadata server to data storage devices (aka data servers), RPC should be used and the threat analysis in the NFSv4-wide security applies just as it does in the non-pNFS case. There will be no separate threat analysis in this document.

* For communication between the data storage devices and the metadata server or between two data storage devices, it is necessary that the communicating peers mutually authenticate and there needs to be a trust relationship between the peers. See Section 26.2.2.4 for an analysis of inter-component trust relationships for the various mapping types.

26.2.2.4. pNFS and Inter-component Trust Relationships.

When pNFS is not involved, there are only two actors involved in file access: the server and the client. There are two possibilities regarding trust relationships:

* With authenticated principals (e.g. RPCSEC_GSS), there is no need for the server to trust the client or the user.

  The client does need to trust the server to obey the protocol and not provide information about its requests to others

* When AUTH_SYS is used, the server has to trust the client to correctly authenticate user principals. To deal with the possibility that clients might take advantage of this situation to cause the server to execute unauthenticated requests, RPC-with-tls can be used together with client peer authentication to limit the set of clients whose unauthenticated requests are accepted or to limit the set of principals that can be identified in this way. At a minimum, acceptance of requests identified as made by root would be limited.

When pNFS is involved, the situation is potentially more complicated in that there are three sorts of actors and six potential trust relationships.

* The interactions between the client and the metadata server are as described above.

* The interactions between client and data server will vary depending on the mapping type.

  For the cases described in Section 26.2.2.3, interactions between client and data server follows the same model as that between the client and metadata server. This case, when principals are authenticated on both the MDS and the data server (i.e., when RPC_SECGSS is used) is the only one in which the client does not have to be specially trusted

  Other mapping types will be discussed below.

* Trust relationships between metadata server and data servers need to exist, although, depending on the details of the mapping type, the client might not be aware of this distinction, and consider both servers together as a unified entity.

  How this issue is addressed for various mapping types are discussed below.

Unlike the cases in which tight coupling is provided, for other mapping types we have no way of authenticating IO requesters and need to trust the client to determine when IO operations are authorized. The resulting situation is similar to that in which AUTH_SYS is used. As is the case when AUTH_SYS is used, the client peer needs to identify itself so that the server can use this identity to avoid a case in which hostile node represents itself as having the ability to individually authorize IO requests given a layout obtained when file was opened.

Because layouts are per-client, rather than per-principal, objects, IO authorization needs to checked for each request, as would be the case if pNFS were not used. Of particular concern is the case in which the principal performing the IO is not the same as the one opening the file.

* In the case of such layout types as block or object, RPC is not used so thee is no opportunity to identify the principal making the request to the data server.

  In the case in which the principal making the IO request is not the opener, the client need to use ACCESS to ascertain the authorization for the IO request.

* In the case of the flexible files layout in the loose coupling mode, the issues are similar, even though RPC is used. Although, the principal identification could be sent, this mapping type specifies that a different unrelated principal identified in the layout is to be passed. As a result, the principal issuing the request is not identified in the RPC request.

  Just as with mapping types that do not use RPC, in the case in which the principal making the IO request is not the opener, the client need to use ACCESS to ascertain the authorization for the IO request. The principal can be authenticated at this point if the metadata server is accessed using RPC_SECGSS.

Issues regarding trust relationships between the metadata server and data servers are discussed below:

* In implementing the file layout and, one assumes, in the case of the flexible files layout in tight coupling mode, there is necessarily a (two-way) trust relationship between metadata server and data server. However, because the nature of their relationship requires adherence to an undocumented control protocol, those outside of the two servers are not in a position to verify or understand the mutual requirements of the two servers.

  For all practical purposes, the client considers the two server interfaces as a unified whole and has a trust relationship with those two server interfaces considered together.

  In particular, authorization decisions for IO requests are to have the same results whether the data server or metadata server and is the responsibility of the two servers to appropriately coordinate to ensure that, just as it needs to coordinate locking globally.

* For layout types such as block and object there is no special control protocol but the data access protocol is a subset of the data device's protocol while the control protocol has access to a different (usually larger) subset of that same data device protocol.

  For the clients to be assured that their data is safe, there need to be restrictions on the use of the data device's protocol to prevent hostile clients getting access to data without authorization by the metadata sever.

* When the flexible files layout type is used in the loose coupling mode, the situation is similar to those in which block or object protocols are used. Just as in those cases, the data device's protocol is NFSv3 but the data access protocol uses a restricted subset with special constraints about how authorization is determined. The metadata server has access to essentially the full NFSv3 protocol.

  Although clients are unlikely to be aware of the details, the safety of their data depends on limiting access by additional clients to data servers that are not restricted to the NFSv3 subset to be used by data servers. In order to effect the necessary limiting, the use of client peer authentication is needed to prevent use by hostile clients that are not prepared to implement the equivalent of the metadata server's authorization decisions. The potential damage is expanded if such hostile clients have access to the full NFSv3 protocol.

27. IANA Considerations

This section uses terms that are defined in [RFC8126].

27.1. IANA Actions

This update does not require any modification of, or additions to, registry entries or registry rules associated with NFSv4.1. However, since this document obsoletes RFC 8881, IANA is presumed to have updated all registry entries and registry rules references that point to RFC 8881 to point to this document instead.

Previous actions by IANA related to NFSv4.1 are listed in the remaining subsections of Section 27.

27.2. Named Attribute Definitions

IANA created 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 namespace identifiers for these attributes are defined as string names. The protocol does not define the specific assignment of the namespace for these file attributes. The IANA registry promotes interoperability where common interests exist. While application developers are allowed to define and use attributes as needed, they are encouraged to register the attributes with IANA.

Such registered named attributes are presumed to apply to all minor versions of NFSv4, including those defined subsequently to the registration. If the named attribute is intended to be limited to specific minor versions, this will be clearly stated in the registry’s assignment.

The registry is to be maintained using the Specification Required policy as defined in Section 4.6 of [RFC8126].

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 the IESG the flexibility to set up bases of assignment of Experimental Use and Standards Action, the prefixes of "EXPE" and "STDS" are Reserved. The named attribute with a zero-length name is Reserved.

The prefix "PRIV" is designated 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).

27.2.1. Initial Registry

There is no initial registry.

27.2.2. Updating Registrations

The registrant is always permitted to update the point of contact field. Any other change will require Expert Review or IESG Approval.

27.3. Device ID Notifications

IANA created a registry called the "NFSv4 Device ID Notifications Registry".

The potential exists for new notification types to be added to the CB_NOTIFY_DEVICEID operation (see Section 25.12). 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 the IETF. Another way to add a notification is to specify a new layout type (see Section 27.5).

Hence, all assignments to the registry are made on a Standards Action basis per Section 4.6 of [RFC8126], 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 that 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 [RFC5662], 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 RFCTBD20, RFCTBD21, 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 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 a new minor version of NFSv4 approved, a Designated Expert should review the registry to make recommended updates as needed.

27.3.1. Initial Registry

The initial registry is in Table 25. Note that the next available value is zero.

Notification Name	Value	RFC	How	Minor Versions
NOTIFY_DEVICEID4_CHANGE	1	RFC 8881	N	1
NOTIFY_DEVICEID4_DELETE	2	RFC 8881	N	1

Table 25: Initial Device ID Notification Assignments

27.3.2. Updating Registrations

The update of a registration will require IESG Approval on the advice of a Designated Expert.

27.4. Object Recall Types

IANA created a registry called the "NFSv4 Recallable Object Types Registry".

The potential exists for new object types to be added to the CB_RECALL_ANY operation (see Section 25.6). 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 27.5).

All assignments to the registry are made on a Standards Action basis per Section 4.9 of [RFC8126], 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 designated 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 that 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 [RFC5662], 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 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 a new minor version of NFSv4 approved, a Designated Expert should review the registry to make recommended updates as needed.

27.4.1. Initial Registry

The initial registry is in Table 26. Note that the next available value is five.

Recallable Object Type Name	Value	RFC	How	Minor Versions
RCA4_TYPE_MASK_RDATA_DLG	0	RFC 8881	N	1
RCA4_TYPE_MASK_WDATA_DLG	1	RFC 8881	N	1
RCA4_TYPE_MASK_DIR_DLG	2	RFC 8881	N	1
RCA4_TYPE_MASK_FILE_LAYOUT	3	RFC 8881	N	1
RCA4_TYPE_MASK_BLK_LAYOUT	4	RFC 8881	L	1
RCA4_TYPE_MASK_OBJ_LAYOUT_MIN	8	RFC 8881	L	1
RCA4_TYPE_MASK_OBJ_LAYOUT_MAX	9	RFC 8881	L	1

Table 26: Initial Recallable Object Type Assignments

27.4.2. Updating Registrations

The update of a registration will require IESG Approval on the advice of a Designated Expert.

27.5. Layout Types

IANA created 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 designated 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 [RFC5662], 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 27.5.3.

4. How the RFC introduces the layout 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 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 a new minor version of NFSv4 approved, a Designated Expert should review the registry to make recommended updates as needed.

27.5.1. Initial Registry

The initial registry is in Table 27.

Layout Type Name	Value	RFC	How	Minor Versions
LAYOUT4_NFSV4_1_FILES	0x1	RFC 8881	N	1
LAYOUT4_OSD2_OBJECTS	0x2	RFC 5664	L	1
LAYOUT4_BLOCK_VOLUME	0x3	RFC 5663	L	1

Table 27: Initial Layout Type Assignments

27.5.2. Updating Registrations

The update of a registration will require IESG Approval on the advice of a Designated Expert.

27.5.3. 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 17.7.3).

   * Include an IANA considerations section, which will in turn include:

     - A request to IANA for a new layout type per Section 27.5.

- A list of requests to IANA for any new recallable object types for CB_RECALL_ANY; each entry is to be presented in the form described in Section 27.4.

- A list of requests to IANA for any new notification values for CB_NOTIFY_DEVICEID; each entry is to be presented in the form described in Section 27.3.

* Include a security considerations section. This section MUST explain how the NFSv4.1 authentication, authorization, and access-control models are preserved. That is, if a metadata server would restrict a READ or WRITE operation, how would pNFS via the layout similarly restrict a corresponding input or output operation?

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 "The Internet Standards Process--Revision 3" (BCP 9 [BCP09]). 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.

27.6. Path Variable Definitions

This section deals with the IANA considerations associated with the variable substitution feature for location names as described in Section 16.17.3. As described there, variables subject to substitution consist of a domain name and a specific name within that domain, with the 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.

27.6.1. Path Variables Registry

IANA created a registry called the "NFSv4 Path Variables Registry".

27.6.1.1. 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 an 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.

27.6.1.1.1. Initial Registry

The initial registry is in Table 28.

Variable Name	RFC	Point of Contact
${ietf.org:CPU_ARCH}	RFC 8881	IESG
${ietf.org:OS_TYPE}	RFC 8881	IESG
${ietf.org:OS_VERSION}	RFC 8881	IESG

Table 28: Initial List of Path Variables

IANA has created registries for the values of the variable names ${ietf.org:CPU_ARCH} and ${ietf.org:OS_TYPE}. See Sections 27.6.2 and 27.6.3.

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.

27.6.1.1.2. 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 update the point of contact of an assignment made on a First Come First Serve basis. Any other update will require Expert Review.

27.6.2. Values for the ${ietf.org:CPU_ARCH} Variable

IANA created 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 designated 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).

27.6.2.1. Initial Registry

There is no initial registry.

27.6.2.2. Updating Registrations

The registrant is free to update the assignment, i.e., change the explanation and/or point-of-contact fields.

27.6.3. Values for the ${ietf.org:OS_TYPE} Variable

IANA created 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 designated for Private Use.

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).

27.6.3.1. Initial Registry

There is no initial registry.

27.6.3.2. Updating Registrations

The registrant is free to update the assignment, i.e., change the explanation and/or point of contact fields.

28. References

28.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, DOI 10.17487/RFC5531, May 2009, <https://www.rfc-editor.org/info/rfc5531>.

[RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, DOI 10.17487/RFC2203, September 1997, <https://www.rfc-editor.org/info/rfc2203>.

[hardlink] The Open Group, "Section 3.191 of Chapter 3 of Base Definitions of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <https://www.rfc-editor.org/info/rfc2743>.

[RFC5040] Recio, R., Metzler, B., Culley, P., Hilland, J., and D. Garcia, "A Remote Direct Memory Access Protocol Specification", RFC 5040, DOI 10.17487/RFC5040, October 2007, <https://www.rfc-editor.org/info/rfc5040>.

[RFC5403] Eisler, M., "RPCSEC_GSS Version 2", RFC 5403, DOI 10.17487/RFC5403, February 2009, <https://www.rfc-editor.org/info/rfc5403>.

[RFC5662] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, DOI 10.17487/RFC5662, January 2010, <https://www.rfc-editor.org/info/rfc5662>.

[symlink] The Open Group, "Section 3.372 of Chapter 3 of Base Definitions of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[RFC5665] Eisler, M., "IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats", RFC 5665, DOI 10.17487/RFC5665, January 2010, <https://www.rfc-editor.org/info/rfc5665>.

[read_atime] The Open Group, "Section 'read()' of System Interfaces of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[readdir_atime] The Open Group, "Section 'readdir()' of System Interfaces of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[write_atime] The Open Group, "Section 'write()' of System Interfaces of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[fcntl] The Open Group, "Section 'fcntl()' of System Interfaces of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[fsync] The Open Group, "Section 'fsync()' of System Interfaces of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[passwd] The Open Group, "Section 'getpwnam()' of System Interfaces of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[unlink] The Open Group, "Section 'unlink()' of System Interfaces of The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, HTML Version", ISBN 1931624232, 2004, <https://www.opengroup.org>.

[RFC4055] 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, DOI 10.17487/RFC4055, June 2005, <https://www.rfc-editor.org/info/rfc4055>.

[CSOR_AES] National Institute of Standards and Technology, "Computer Security Objects Register", May 2016, <https://csrc.nist.gov/projects/computer-security-objects-register/algorithm-registration>.

[RFC7861] Adamson, A. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", RFC 7861, DOI 10.17487/RFC7861, November 2016, <https://www.rfc-editor.org/info/rfc7861>.

[RFC8000] Adamson, A. and N. Williams, "Requirements for NFSv4 Multi-Domain Namespace Deployment", RFC 8000, DOI 10.17487/RFC8000, November 2016, <https://www.rfc-editor.org/info/rfc8000>.

[RFC8166] Lever, C., Ed., Simpson, W., and T. Talpey, "Remote Direct Memory Access Transport for Remote Procedure Call Version 1", RFC 8166, DOI 10.17487/RFC8166, June 2017, <https://www.rfc-editor.org/info/rfc8166>.

[RFC8267] Lever, C., "Network File System (NFS) Upper-Layer Binding to RPC-over-RDMA Version 1", RFC 8267, DOI 10.17487/RFC8267, October 2017, <https://www.rfc-editor.org/info/rfc8267>.

[RFC8154] Hellwig, C., "Parallel NFS (pNFS) Small Computer System Interface (SCSI) Layout", RFC 8154, DOI 10.17487/RFC8154, May 2017, <https://www.rfc-editor.org/info/rfc8154>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8178] Noveck, D., "Rules for NFSv4 Extensions and Minor Versions", RFC 8178, DOI 10.17487/RFC8178, July 2017, <https://www.rfc-editor.org/info/rfc8178>.

[RFC8435] Halevy, B. and T. Haynes, "Parallel NFS (pNFS) Flexible File Layout", RFC 8435, DOI 10.17487/RFC8435, August 2018, <https://www.rfc-editor.org/info/rfc8435>.

[RFC8587] Lever, C., Ed. and D. Noveck, "NFS Version 4.0 Trunking Update", RFC 8587, DOI 10.17487/RFC8587, May 2019, <https://www.rfc-editor.org/info/rfc8587>.

[RFC8881] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <https://www.rfc-editor.org/info/rfc8881>.

[RFC9289] Myklebust, T. and C. Lever, Ed., "Towards Remote Procedure Call Encryption by Default", RFC 9289, DOI 10.17487/RFC9289, September 2022, <https://www.rfc-editor.org/info/rfc9289>.

[I-D.ietf-nfsv4-internationalization] Noveck, D., "Internationalization for the NFSv4 Protocols", Work in Progress, Internet-Draft, draft-ietf-nfsv4-internationalization-13, 15 August 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-nfsv4-internationalization-13>.

[BCP09] Best Current Practice 9, <https://www.rfc-editor.org/info/bcp9>. At the time of writing, this BCP comprises the following:

Bradn er, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, DOI 10.17487/RFC2026, October 1996, <https://www.rfc-editor.org/info/rfc2026>.

Kolkman, O., Bradner, S., and S. Turner, "Characterization of Proposed Standards", BCP 9, RFC 7127, DOI 10.17487/RFC7127, January 2014, <https://www.rfc-editor.org/info/rfc7127>.

Dusseault, L. and R. Sparks, "Guidance on Interoperation and Implementation Reports for Advancement to Draft Standard", BCP 9, RFC 5657, DOI 10.17487/RFC5657, September 2009, <https://www.rfc-editor.org/info/rfc5657>.

Housley, R., Crocker, D., and E. Burger, "Reducing the Standards Track to Two Maturity Levels", BCP 9, RFC 6410, DOI 10.17487/RFC6410, October 2011, <https://www.rfc-editor.org/info/rfc6410>.

Resnick, P., "Retirement of the "Internet Official Protocol Standards" Summary Document", BCP 9, RFC 7100, DOI 10.17487/RFC7100, December 2013, <https://www.rfc-editor.org/info/rfc7100>.

Dawkins, S., "Increasing the Number of Area Directors in an IETF Area", BCP 9, RFC 7475, DOI 10.17487/RFC7475, March 2015, <https://www.rfc-editor.org/info/rfc7475>.

28.2. Informative References

[RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, DOI 10.17487/RFC3530, April 2003, <https://www.rfc-editor.org/info/rfc3530>.

[RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <https://www.rfc-editor.org/info/rfc1813>.

[RFC2847] Eisler, M., "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM", RFC 2847, DOI 10.17487/RFC2847, June 2000, <https://www.rfc-editor.org/info/rfc2847>.

[Chet] Juszczak, C., "Improving the Performance and Correctness of an NFS Server", USENIX Conference Proceedings, June 1990.

[RFC3232] Reynolds, J., Ed., "Assigned Numbers: RFC 1700 is Replaced by an On-line Database", RFC 3232, DOI 10.17487/RFC3232, January 2002, <https://www.rfc-editor.org/info/rfc3232>.

[RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, DOI 10.17487/RFC1833, August 1995, <https://www.rfc-editor.org/info/rfc1833>.

[rpc_xid_issues] Werme, R., "RPC XID Issues", USENIX Conference Proceedings, February 1996.

[RFC5664] Halevy, B., Welch, B., and J. Zelenka, "Object-Based Parallel NFS (pNFS) Operations", RFC 5664, DOI 10.17487/RFC5664, January 2010, <https://www.rfc-editor.org/info/rfc5664>.

[RFC5663] Black, D., Fridella, S., and J. Glasgow, "Parallel NFS (pNFS) Block/Volume Layout", RFC 5663, DOI 10.17487/RFC5663, January 2010, <https://www.rfc-editor.org/info/rfc5663>.

[RFC2054] Callaghan, B., "WebNFS Client Specification", RFC 2054, DOI 10.17487/RFC2054, October 1996, <https://www.rfc-editor.org/info/rfc2054>.

[RFC2055] Callaghan, B., "WebNFS Server Specification", RFC 2055, DOI 10.17487/RFC2055, October 1996, <https://www.rfc-editor.org/info/rfc2055>.

[errata] IESG, "IESG Processing of RFC Errata for the IETF Stream", July 2008, <https://www.ietf.org/about/groups/iesg/statements/processing-rfc-errata/>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[xnfs] The Open Group, "Protocols for Interworking: XNFS, Version 3W", ISBN 1-85912-184-5, February 1998.

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

[RFC7143] Chadalapaka, M., Satran, J., Meth, K., and D. Black, "Internet Small Computer System Interface (iSCSI) Protocol (Consolidated)", RFC 7143, DOI 10.17487/RFC7143, April 2014, <https://www.rfc-editor.org/info/rfc7143>.

[FCP-2] Snively, R., "Fibre Channel Protocol for SCSI, 2nd Version (FCP-2)", ANSI/INCITS, 350-2003, October 2003.

[OSD-T10] Weber, R.O., "Object-Based Storage Device Commands (OSD)", ANSI/INCITS, 400-2004, July 2004, <https://www.t10.org/drafts.htm>.

[PVFS] Carns, P. H., Ligon III, W. B., Ross, R. B., and R. Thakur, "PVFS: A Parallel File System for Linux Clusters.", Proceedings of the 4th Annual Linux Showcase and Conference, 2000.

[access_api] The Open Group, "The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition", 2004, <https://www.opengroup.org>.

[RFC2224] Callaghan, B., "NFS URL Scheme", RFC 2224, DOI 10.17487/RFC2224, October 1997, <https://www.rfc-editor.org/info/rfc2224>.

[RFC2755] Chiu, A., Eisler, M., and B. Callaghan, "Security Negotiation for WebNFS", RFC 2755, DOI 10.17487/RFC2755, January 2000, <https://www.rfc-editor.org/info/rfc2755>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[AFS] Spasojevic, M. and M. Satayanarayanan, "An Empirical Study of a Wide-Area Distributed File System", ACM Transactions on Computer Systems, Vol. 14, No. 2, pp. 200-222, DOI 10.1145/227695.227698, May 1996, <https://doi.org/10.1145/227695.227698>.

[RFC5661] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 5661, DOI 10.17487/RFC5661, January 2010, <https://www.rfc-editor.org/info/rfc5661>.

[RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <https://www.rfc-editor.org/info/rfc7530>.

[RFC7931] Noveck, D., Ed., Shivam, P., Lever, C., and B. Baker, "NFSv4.0 Migration: Specification Update", RFC 7931, DOI 10.17487/RFC7931, July 2016, <https://www.rfc-editor.org/info/rfc7931>.

[RFC8434] Haynes, T., "Requirements for Parallel NFS (pNFS) Layout Types", RFC 8434, DOI 10.17487/RFC8434, August 2018, <https://www.rfc-editor.org/info/rfc8434>.

[I-D.dnoveck-nfsv4-security] Noveck, D., "Security for the NFSv4 Protocols", Work in Progress, Internet-Draft, draft-dnoveck-nfsv4-security-12, 16 May 2025, <https://datatracker.ietf.org/doc/html/draft-dnoveck-nfsv4-security-12>.

[I-D.dnoveck-nfsv4-acls] Noveck, D., "ACLs within the NFSv4 Protocols", Work in Progress, Internet-Draft, draft-dnoveck-nfsv4-acls-07, 24 May 2025, <https://datatracker.ietf.org/doc/html/draft-dnoveck-nfsv4-acls-07>.

[I-D.dnoveck-nfsv4-rfc5662bis]
Noveck, D., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", Work in Progress, Internet-Draft, draft-dnoveck-nfsv4-rfc5662bis-06, 23 May 2025, <https://datatracker.ietf.org/doc/html/draft-dnoveck-nfsv4-rfc5662bis-06>.

Appendix A. Nature of the Changes Being Made for This Update

A large number of changes being made in this document are made to effect corrections to previous NFS Version 4.1 specifications. These include changes to address errata reports connected with those specifications, including some that were assigned the status REJECTED. In addition, similar changes are being made without explicit errata reports.

It is important to note that there are also a number of important organizational changes discussed below that will be made in this updated specification. As work on this document progresses, the status of those changes, together with other necessary changes, will be summarized in Appendix B.

* The updated specification will depend on a number of NFSv4-wide documents, as described in Appendix A.1, rather than trying to deal with every aspect of the protocol description itself.

  In the case of security, there will have to be decisions on how v4.1-specific security-related features will be addressed. See Appendix A.2 for details.

A.1. Reliance on NFSv4-wide Documents

In many cases, matters previously described within the NFSv4.1 specification, will be addressed in separate NFSv4-wide documents.

* The process of protocol extension and creation of new minor versions is described in a separate NFSv4-wide document, [RFC8178], dealing with the issue for the NFSv4 protocols as a whole.

* Internationalization will be described in a separate document describing internationalization for all of the NFSv4 protocols, currently [I-D.ietf-nfsv4-internationalization]. The only v4.1-specific feature is the fs_charset_cap attribute, which is described in the current document rather than the internationalization document, although that document does discuss our choices in the matter.

* Security will also be described in a separate document applying to all minor versions. The handling is different because there is a wider range of security-relevant features within v4.1, despite the fact that the fundamental approach is the same for all minor versions. As a result, for some features, the security document will have the lead role while, for others, the v4.1 specification will be the main source of information about the feature, although the basic security functionality will be as defined by the NFSv4-wide security document.

A.2. Adaptation of the NFSv4-wide Security Document to v4.1-specific Features

The v4.1-specific security-related features are dealt with as described below:

* Security issues regarding pNFS will be the responsibility of this v4.1 specification document. In doing this, we will rely, where we can, on the security facilities described in the NFSv4-wide security document.

  This is necessary because some layout types will access data without using the RPC-based facilities that are discussed in the security document. In addition, the requirements for security-related co-ordination between data server and metadata server are best dealt with in this document, including cases in which RPC is used by both the data server and the metadata server, in which the necessary co-ordination requirements are defined by the layout type specification document.

* The description of the SECINFO_NO_NAME operation, will remain in the v4.1 specification, even though the description of SECINFO pseudo-flavors will be consolidated in the security. document.

  This approach is necessary because the description of SECINFO pseudo-flavors needs to be augmented to allow negotiation of security-related transport characteristics in addition to auth-flavors, associated mechanisms and RPCSEC_GSS services.

* The description of the MACH_CRED and SSV features will remain in the v4.1 specification document and will only be mentioned in passing in the security document.

  Instead, the focus with regard to state protection will be on client-peer authentication which applies to all minor versions.

A.3. Changes to Effect Necessary Cleanup and Correction

The review of the existing specification text has discovered a set of areas that require clarification or correction, even though the problems had not been noticed as part of the pre-publication review of [RFC8881] and no errata reports have yet been filed.

In the following cases, it was necessary to make revisions to make the use of certain terms uniform throughout the document or to clarify the definitions which have come to disagree with the initial definitions.

* The treatment of the term "client owner" has been clarified to deal with the fact that previous specifications were inconsistent about whether the verifier was part of the client owner or added to it.

  In this draft, a "client owner" always includes a verifier. When it is necessary to refer to the opaque string within it, the term "client owner id" is used.

  These changes appear in Sections 2.5 and 5.5.

* The definition of "verifier" has needed to substantially revised to reflect the fact that there multiple verifiers within the protocol, each with its own use.

  These changes appear in Section 2.5

* There has been a set of changes motivated by a need to clarify the circumstances under which delegation might be revoked.

  This involved parallel changes in the description of leases where existing text was confusing because it was sometimes assumed that all locks were included rather than non-recallable ones, which obscures discussion of delegation/layout recall and revocation.

  These changes appear in Sections 2.5 and 3.

A large set of changes were made to address issues within Section 7.6. These include:

* The requirement to wait forever for a response before reusing a slot has been modified to allow such waits to be terminated because of extraordinary circumstances such as termination of the task issuing the request.

That had to be changed because clients were unable to conform and because of the weakness of the proposed justification for the prohibition, i.e., that it resulted in a choice as to the next sequence value to be used.

The replacement makes clear why the sequence number is to be advanced, which is useful in reducing the probability of false retries.

* The prohibition on request retry was changed from a normative requirement to implementation guidance because it was clearly not a "fundamental requirement of the specification". Also the justification for a strict prohibition was undercut by work done in NFSv4.1 to implement Exactly-One Semantics, whose goal was to avoid negative consequences due to retries.

  The replacement text clearly indicates why such retries are useless and best avoided, which is consistent with current practice. However it was necessary, in order to limit the occasions in which false retries could occur to use MUST NOT to forbid issuing of retries for abandoned requests once the slot had been used to send a later request.

* The discussion of false retries had to be extensively revised to make it clear that, while there were requirements to report certain false retries when detected, there were not corresponding requirements to check for this possibility. Instead situations in which such checking might be prudent are provided.

  In the revised section, it is clear that false retries cannot occur if requests are never abandoned without a response and the protocol is implemented correctly. In addition, it is made clear how unlikely such false retries are if such request retries are constrained as required by the text related to valid reasons for request abandonment.

Appendix B. Status of The Changes Being Made in this Update

Like all internet drafts, this document is a work in progress. In this particular case, that designation is particularly appropriate since there are specific changes that need to be made and either have not made or have been started but not completed. Information regarding changes made or to be made in this update is to be found in Appendix sections B.1 through B.4.

The current form in which the material is presented is designed for internal use within the working group, in order to help track the document’s progress towards its goals.

Ultimately, the material regarding these changes will be re-organized in an eventual RFC.

B.1. Changes Completed So Far in this Update

Work on the necessary changes discussed below has already been completed, although necessary review might not yet have occurred. At least for a while, changes made in later drafts of the working group document (i.e. those beyond -00) will not be reflected in this section and will be found within a subsection of Appendix B.5

The discussion of minor versioning has been updated to refer to [RFC8178], instead of the former approach which allowed each minor version to make its own versioning rules.

The document has been updated to eliminate the current (erroneous) treatment of internationalization, derived from earlier NFSv4.1 specifications [RFC5661], [RFC8881]. The section dealing with internationalization has been deleted, since it was never implemented. In its place, the specification has been modified to reference an external document which is to define the appropriate handling for internationalization for the NFSv4 protocols as a whole. Currently, that document is the I-D draft-ietf-nfsv4-internationalization [I-D.ietf-nfsv4-internationalization]. In addition, the treatment of the fs_charset_cap attribute has been revised to reflect the analysis presented in the internationalization document.

Despite the completion of the internationalization work within this document, there remains work to do, most of which involves the completion of review for the NFSv4-wide internationalization document

* The new document was based on the treatment of internationalization within [RFC7530], which served as a useful starting point, since implementation of all NFSv4 minor version followed the same approach to internationalization issues. However the following issues still needed to be addressed:

* There was a need to update the treatment within RFC7530 to reflect IDNA changes made soon after the document was published.

* There was a need to deal better with client name-caching issues, especially in the context of case-insensitive file systems. Text has been written and submitted but review is needed.

* There was a need to address more fully the provision of representation-independent name lookup, which maps all canonically equivalent name strings in a directory to the same file.

However, these issues are being addressed in the context of the internationalization document, rather than the NFSv4.1 specification.

B.2. Changes Made in this Update to Address NFSv4.1 Errata Reports

Work has been done to deal with errata reports associated previous NFSv4.1 specifications. These include a large set of errata reports associated with RFC5661 and a few associated with RFC8881. This work can be categorized as follows:

* The following errata reports associated with RFC5661 were dealt with in RFC8881, either because their substance related to issues to be dealt with in RFC8881 or because the simplicity of the needed change and its non-controversial nature made it simple to address the report as part of the RFC editing process for RFC8881: 2062, 2280, 2324, 2330, 2548, 3558, 5212.

* Work needed to be done to address many errata reports relevant to RFC 5661 that were not addressed in RFC8881, because the change was too large or too potentially too controversial to address in the context of RFC editing for RFC881.

* There are a number of errata reports associated with RFC8881, that also had to be addressed.

* A few of the existing errata reports might have implications for RFC5662 and would need to be addressed by an eventual rfc5662bis RFC.

The errata reports that remain and that are being addressed in this document include reports currently assigned a range of statuses in the errata reporting system, including reports marked Verified, Hold for Document Update, and Rejected. These statuses are relevant to the processing of the associated errata but not in a way as direct as might be anticipated since errata reports marked Rejected might be addressed, as a result of a justified change in working group consensus.

* The following errata reports associated with RFC5661, have already been addressed in this document draft, in some cases by splitting out the associated change, if still necessary, into a related document: 2005, 2006, 2249, 2291, 2299, 2326, 2327, 2328, 2505, 2722, 3064, 3065, 3066, 3068, 3208, 3379, 3653, 3714, 3901, 4119, 4215, 4492, 4572, 4711, 4712, 4914, 5040, 5417, 5467, 5476, 6015, 6324.

* The following errata reports associated with RFC8881 already been incorporated into this document draft: 6308, 6337, 6865, 6611.

* The following errata reports associated with RFC5661 had not previously been addressed but will be resolved with publication of this draft as described in Appendix B.5.5: 2751, 3067, 4118, 5982.

The following issues need to be discussed with the errata report authors and the rest of the working group to enable the issues raised to be addressed in the resulting RFC:

* Errata reports 2751 and 3067 are related as both have to do with LAYOUTCOMMIT on the file layout type. As a result they are best discussed together.

  The current status of 2751 is REJECTED which is justified given the scope of the proposed change. Nevertheless, it seems the working group needs to address this area, if not necessarily using the current proposed text for this report.

  These reports need further working group discussion before the necessary changes are made in the document proper.

* Errata report 4118 has a current status of DEFER and for the most part appears unproblematic.

  The proposed text uses the RFC2119-defined keyword "SHOULD" in a way that is not in accord with its definition and adds confusion to the proposal.

  Once agreement is reached on the details of the replacement text, this issue should be easy to address.

* Errata report 5982 has current status of REJECTED. After further consideration of the issues, the proposer decided that the proposed replacement text addressed the wrong issue and so will be dropped.

  Material related to the issues which this report was intended to address are being dealt with as described in Appendix C.2.1.

The only reports that still need to be addressed are 2751, 3067, and 4118.

B.3. Changes Being Made Now in this Update

Work on the necessary changes discussed below has started but is not yet complete. This includes cases in which work to be completed is not within this document, but in a document referred to by this document. In such cases, matters formerly dealt within the NFSv4.1 specification, in the form of a single document, need to be addressed in a number of documents, each dealing with all NFSv4 minor versions together.

As noted previously, there are significant problems with the treatment of security within previous NFSv4.1 specifications [RFC3530], [RFC5661], and within other current NFSv4 specifications (e.g. [RFC7530], [RFC8881]). These are listed in Section 26.1. Work has started on these issues, although it is not as advanced as that for internationalization, since many important decisions need to be made. There is now a security I-D [I-D.dnoveck-nfsv4-security] which will serve as a guide to the decisions that will need to be made to guide the further work to arrive at a Proposed Standard discussing security for all the NFSv4 protocols, which rfc56661bis will refer to normatively.

Work has been done in Sections 2.7 and 5.3 to make the presentation more suitable to an environment in which RPC makes transport-level encryption and client-host authentication available. However, there is a need for some working group decisions to be made before completion of the transition to a security framework that fully embraces these new elements. In addition, the writing of a new Security Considerations section will require substantial progress on a standards-track security document for NFSv4 as a whole. Once that work is done, there will need to be a re-organization of those sections and their role will primarily be to refer to the standards-track security document.

In addition, work has been done to address security issues for NFSv4.1-specific features:

* Significant work has been done to clarify security implications of pNFS.

  This work has primarily consisted of a major revision of Section 17.9 although there are significant updates to Sections 2.7 and 2.8.2.

  It has been made clear that the only cases in which there are essentially no security consequences from the use of pNFS, are those in which RPC is used by the storage protocol, correcting text in previous specifications which gives a contrary impression.

The text has been revised to take account of the existence of services provided by rpc-tls including encryption and client host authentication.

There has been a re-organization of Section 17.9, including separate subsections dealing with non-RPC-based storage protocols and RPC-based storage protocols with either loose or tight coupling between storage server and metadata server.

* Significant work has been done to provide rpc-tls-based state protection which can be taken advantage even by clients who have not implemented SP4_MACH_CRED or SP4_SSV or who are using AUTH_SYS.

  The Section 5.5.3 has been revised to allow, when SP4_NONE is used, client host authentication to be used for state protection.

  It is made clear in Section 7.8.3 that the use of SP4_NONE, when host-client authentication is active, provides state protection against other clients rather than waiving state protection.

  For many of the changes mentioned above, the definitive treatment will appear in the NFSv4-wide security document and there might also be a temporary references to the preliminary security I-D.

  Further changes along these lines will most likely be necessary wherever in the document the SP4_* values are referred to.

There are a number of issues relating to the use of the key words defined in [RFC2119]. While the issues below could be treated individually and distributed among Sections B.1 through B.4, for now, we will treat them together.

* A shift has been made from only citing [RFC2119] to citing [RFC8174]. While it is sometimes said that, in the absence of RFC8174, "must" and "MUST" are to be considered synonymous, the working group has never interpreted RFC2119 in that way, although the clarification provided by RFC8174 was helpful. In light of this, it might be considered that all the necessary work has been done, apart from necessary review. However, given the working group discussion about this issue in connection with RFC8881, it appears that the working group will need to further discuss this issue soon after this document becomes a working group document. That would enable us to consider this aspect of the work complete.

* The use of the term "RECOMMENDED" to describe NFSv4 attributes which are not REQUIRED as been addressed by switching to the term "OPTIONAL", since "RECOMMENDED" is not in accord with [RFC2119]. This work is considered complete.

* There is ongoing work to deal with what appears to be a misclassification of protocol-defined attributes, making a number of attributes OPTIONAL, when the practical difficulties for clients in dealing with the absence of server support, makes this an inappropriate choice.

  The security-related attributes owner, owner_group, and mode have been made REQUIRED, both in this document and in [I-D.dnoveck-nfsv4-security].

  The working group needs to review existing OPTIONAL attributes to see if similar changes need to be made for other attributes derived from NFSv3.

* For many of the existing uses of the terms "SHOULD" and "SHOULD NOT", it is not clear that the meaning is compatible with RFC2119. The difficulty is compounded by uncertainty left as to the proper use of these terms, about which there may be disagreement within the working group. See Appendix C.1.1 for a detailed discussion of issues that need to be resolved. Some work has started on this area by adjusting text in certain sections but the work cannot be completed until there is agreement about the proper use of these terms in this document.

  These issues have been addressed by changes in Sections 5.7.2, 5.7.3, and 7.6.1). In addition, similar changes will be made in [I-D.dnoveck-nfsv4-security].

* There are some cases in which the terms "MUST" and "MUST NOT" have essentially been ignored by implementations or there are other reasons to believe that these terms may have been used inappropriately. See Appendix C.1.2 for a detailed discussion. Some work has been done toward addressing these issues but it is not complete, because further discussion is needed regarding changes to be made in Section 7.6.2.

  These issues have been addressed by changes in Sections 5.7.1 and 5.7.2.

B.4. Changes That Will Need to be Made Later in this Update

Work on necessary changes discussed below has not started yet, although some discussion and planning may have occurred, possibly together with preliminary specification text within Appendix C proposing likely changes to be made late in the specification proper.

Work needs to incorporate the material within [RFC8434], which establishes the requirements for parallel NFS (pNFS) layout types, which are not clearly specified in RFC 5661, leading to confusion.

There are a number of issues that need to be addressed regarding the treatment of Directory Delegations. As initial discussion of these issues has resulted in no clear consensus on how these issues should be addressed in this minor version, further working group discussion will be needed. See Appendix C.2.2. for details.

There are issues that need to be addressed regarding how retried requests are to be terminated, including the fact the most common client handling of this situation violates a "MUST" in the existing specification. This is expected to take the form of a revised Section 7.6.2, as discussed in Appendix C.2.1.

B.5. Work Done in Various Drafts

The work in the subsections below cover changes made in various drafts of draft-ietf-nfsv4-rfc5661bis and does not cover changes made in drafts of draft-dnoveck-nfsv4-rfc5661bis. As a result all such changes appear in Appendix B.5.1.

B.5.1. Changes Made in Draft -00

This draft made major organizational changes in the text inherited from [RFC8881] and started the work to clean up many of the troublesome issues discussed in Appendices C.1 and C.2.1.

The organizational changes included the following:

* Creating a new top-level section describing the reasons for this update.

* Moving most of the security-related material into its own NFSv4-wide document.

* Deleting the existing treatment of internationalization and referring the reader to the new NFSv4-wide internationalization document.

* Creating the initial versions of Appendices A, B, and C to track and explain changes needed and made.

B.5.2. Changes Made in Draft -01

Beyond limited editorial changes, this section lists the work done in draft -01.

The toc depth has been returned to the default value of three, with exclusions for subsections of operations and callbacks. The value of two left too many important third-level sections that did not appear the table of contents.

A large part of the changes consist of the changes described in in more detail in Appendix A.3.

* The re-organization of the description of client owner

* The revised explanation of the term "verifier".

* The enhanced description of delegation revocation.

* The set of changes made to address issues regarding EOS, retry and false retry, made to Section 7.6.

An interrelated set of changes were made in the pNFS area in order to clarify and re-organize the treatment if pNFS security, with some of it being the responsibility of the security document and to revise the material to clearly address the issues dealt with RFC8434 which is now obsoleted by this document. The following specific changes were made:

* Changes were made in terminology so that the general description of pNFS is consistent both with the files layout type and the layout types that appropriately deal with "storage devices.

  The general term "data access protocol" refers both storage protocol and the use of file protocol to access a data server.

* The treatment of the term "control protocol" has been revised so as to avoid a contradiction between the presentation in RFC8881, which implied there was always a control protocol and RFC8435 which claimed it covered cases in which there was no control protocol.

* The section on pNFS security has been substantially revised for greater clarity and generality.

* A start has been made organizing the portions of the threat analysis that are to reside in rfc8881bis proper.

A new section for issues that need discussion was added as Appendix C.2.4. It deals with the following issues:

* A lack of clarity regarding possible persistence of lock state.

* A number of issues in the description of reply cache persistence that either make the feature difficult to implement or unnecessarily suggest that it needs to do things that are inherently difficult or impossible.

A new section (Appendix C.2.3) was added regarding issues raised by the discussion of memory-mapped files, now in Section 15.7. In addition, that section has been revised, in this draft to address the following issues:

* A neglect of the expected role of opening files for read, which would have caused delegation recall, rendering many of the issues worried about irrelevant.

* An unjustified expectation that, with mandatory locking in effect IO operations would result in obtaining locks making deadlock likely.

* An unjustified assumption that locking issues present in the last of mandatory locking apply as well in the case of advisory locking.

B.5.3. Changes Made in Draft -02

A number of changes related to the classification of attributes have been made:

* Attributed described as "Recommended", which previously has been described (incorrectly) as RECOMMENDED, were described as OPTIONAL, in accord with [RFC2119].

* The attributes Mode, Owner, Owner_group, previously OPTIONAL, although referred to as "Recommended", have been made REQUIRED.

  This change parallels similar changes in the NFSv4-wide security document [I-D.dnoveck-nfsv4-security].

B.5.4. Changes Made in Draft -03

A number of changes have been made to adapt to the splitting of ACL-related material from the security document and its presentation in a separate document devoted to ACLs [I-D.dnoveck-nfsv4-acls].

* Many reference to the security document have been updated to include the acls document as well [I-D.dnoveck-nfsv4-acls].

* Many reference to specific sections of the security document have been updated to reflect changes to that document.

  Some of these have been modified to reference sections that are now in the acls document [I-D.dnoveck-nfsv4-acls].

In addition, a number of changes were made regarding the handling of various security-related attributes, introducing the topic with the understanding that the full treatment of the associated issues will be done within the NFSv4-wide security document [I-D.dnoveck-nfsv4-security].

* The paragraph regarding GETATTR and SETATTR of the name attribute directory has been rewritten to eliminate the dubious logic even though the protocol has not been changed, leaving a gap that still need to be addressed separately for POSIX authorization and ACLs.

* The section regarding the interpretation of owner and owner-group strings has been rewritten to introduce the possible choices, leaving the policy issues to the NFSv4-wide security document.

Further work was done on the issues discussed in Appendix C.2.1 including addressing issues originally intended to be dealt with a part of errata report 5982.

The description of the errata report status has been revised making it clear that only three reports still need to be addressed

We have created new per-draft Appendices B.5.1, B.5.2, and B.5.4 to keep track of specification changes.

There has been considerable work within Appendix C.2 including the following:

* Reorganization of the listing of the statuses of Appendix C.2 subsections according to the drafting of corresponding changes

* Considering Appendix C.2.1 Complete and ready for review.

* Creation of Appendices C.2.5 and C.2.6.

B.5.5. Changes Made in Draft -04

Major revision have been made to Section 8 and Appendix C.2.4 in order to:

* update the description of persistence-related issues to reflect recent discoveries.

* provide an updated description of a persistence that has a good chance of being implemented and that could eliminate most grace period delays in the event of server restart.

* Clarify how a client becomes aware of persistence cross a server restart.

Considerable work was done in line with the suggestions in Appendix C.2.2. The following issues were addressed:

* The uncertainty about what assumptions could be made about the stability of cookie values and directory entry ordering by a directory delegation holder.

* The potential confusion about the effect of batching and delays of notifications needed to be addressed by making it clearer that these only applies to updates of attribute of file in the directory, rather than to the directory contents.

As a consequence of this work, the drafting associated with the various subsections of Appendix C.2 reached completion and it was necessary to revise Appendix C.2 proper to guide the necessary working group discussion of those changes.

There was considerable work clarifying the handling of CLAIM_DELEG_PREV. This includes:

* Distinguishing the special period allowed after client restart from the grace period used as part of server restart.

* Defining use of DELEG_PREV claim types as reclaim-type operations while making it clear that they have no relation to the grace period or RECLAIM_COMPLETE

Completed all necessary errata-based changes for this document by making the changes listed below:

* Errata report 2751, although rejected, has been incorporated into this draft.

  The proposed changes were followed fairly closely, although the proposed new section has been moved from the pNFS chapter to the pNFS file chapter.

* Errata report 3067 has been incorporated into this draft.

  The only divergence from the proposed text were the deletion of the word "deprecated" which was replaced by "are no longer used". Some incorrect uses of RFC2119-defined keywords were made lower-case.

* Errata report 4118 has been incorporated into this draft.

  The only divergence from the proposed text were the deletion of the phrase "SHOULD be ignored" and its replacement by a statement that the field has no use. According to the author’s reading of [RFC2119], "no harm, no ’SHOULD’ applies here.

* Errata report 5982, which was rejected, has not been incorporated into this draft.

  The change proposed, mentioning XID in connection with the false retry discussion turned out to be inadvisable and was dropped.

  Despite that, other changes, discussed above, were made to satisfy the original motivation of this errata report, to make it clearer why extensive checking to detect false retry is not likely to be done, an issue which has needed to be addressed for a while.

Appendix C.2.7 was created to track discussion of errata that had been rejected.

B.5.6. Changes Made in Draft -05

Changes to make it clearer when a change in the link count made by anyone other than the delegation holder cause a recall of a delegation. In particular, the case of changes done by NFSv4.0 clients needed clarification as did the rules for write delegations.

Added a discussion of the defects fixed as part of the rfc8881bis effort, focusing on the possibility of compatibility issues and the limited use of protocol extension, as provided for in Section 9 of [RFC8178]. This work included:

* Adding a summary of the work done to correct existing protocol defects in a new Section 1.4

* Extension of the notification enum to enable changes discussed below to enable implementation of directory delegations.

* Preliminary discussion of a new OPTIONAL attribute aclchoices to avoid having to wait for v4.2 to implement an aclfeatures attribute, as previously anticipated.

Made a series of changes to the discussion of directory delegations in Appendix C.2.2 and elsewhere. This work was prompted by suggestions from Rick Macklem and others.

* Additions to Appendix C.2.2 discussing reasons to provide a more flexible approach to the provision of position information within content update notifications.

* Adding of new Appendices C.2.2.2 and C.2.2.3 discussing how that flexibility might be provided

* Adding of a new Appendix C.2.2.5 together with corresponding work in Section 15.9

In addition to the above work within the Appendices, a number of changes were made to the specification proper to create new facilities to deal with issues discussed above and to incorporate Rick's suggestions to make the discussion directory delegations clearer. The changes included the following:

* Creating a new top-level section (now Section 15.9) explaining the directory delegation feature.

* A major revision of Section 23.39 to explain the use of the input and result notification bitmaps.

* A major rework/restructuring of Section 25.4 providing separate subsections for notification types.

Additional work was done in material first discussed in Appendix C.2.1 in order more clarity about the motivation for the changes and the connection to probbility of false retries. These changes were made in response to Olga Kornievskaia's review of version of this work appearing in the -04 drafts. The changes include the following:

* Made a number of small changes to various subsection of Section 7.6. One of the most importamce of these is connected to the use of retry without disconnection, formerly normatively prohibited but now a just a pointlss, useless exercise. A normative prohinition was added about retries of requests that had been abandoned which was necessary to further limit the possibility of false retries.

* Major changes were made to Section 7.6.1.3.1 to make it clear that, while there were some requirements regarding the reporting of false retries that came to the replier’s attention, there were no requirements regarding the work that replier’s needed to do to make sure these would be found, if they occurred.

  In addition, it was made clear why it was quite inlikely for these to occur, if the requirements laid out in Section 7.6 are followed.

* Description of the changes and their motivations were added to Appendix A.3.

B.5.7. Changes Made in Draft -06

Significant additions were made to Section 1.3, adding new items to the existng list.

* In item 7, description of a new OPTIONAL attribute defining what extensions of the core UNIX ACL model are supported by the server.

* In item 8, discussion of a ew ACE flag to support differences in handling of partial ACE satisfaction for draft-POSIX ACLs,

* In Item 9, discussion of a new ACE flag to support implementation of "default" ACLs as provided for by draft-POSIX ACLs.

B.5.8. Changes Made in Draft -07

The following changes were made:

* A new item (#10) was added to the list in Section 1.3.

  This new item mentions the definition of GROUPNOTOWNER@ and OTHERS@ to be used to translae reverse-slope modes where DENY ACLs are not supported.

* Added definitio of utf8pref to be used for components, since use of UTF8 is preferred by not required for these.

B.5.9. Changes Made in Draft -08

The following changes were made in this draft:

* Moved Kerberos-specfic material from 61bis to the security doument.

* Moved SECINFO description back to this document.

B.5.10. Changes Made in Draft -09

The following changes were made in this draft:

* Changes were made to clarify the role of the grace period as described in Section 13.4.2.

  It appears that changes made in connection with the addition of RECLAIM_COMPLETE seemed to mke it necessary to delay all IO until completion of the frace period rather than the acquiring of new (i.e. not reclaimed) locks. This increased the negative effects of server recovery on clients, particularly if there were clients whose lock reclaim processing were delayed.

* Major additions were made to complete Section 26.2.

  These included writing Sections 26.2.1, 26.2.2.3, 26.2.2.1, 26.2.2.2, and 26.2.2.4.

B.5.11. Changes Made in Draft -10

The following changes were made in this draft:

* Aftr-the-fact additions of "Changes made in Draft" section for -06, -07.

* Creation of a new section for RFC Editor Notes containing definitions of RFCs to replace the strings RFCTBD{10,20,21,22,30} in the final published document.

  Use of these strings to simplify dscussions of the re;ationships among documents that are part of the rfc56661bis effort.

* Create a new section 1.3 dealing with Compatibility Issues.

* Remove outdated material from the current Section 1.4 (previously 1.3).

List tems 8-10 were removed because these had become unnecessary due to the change of approach to draft-POSIX ACLs. Instead of enhancing NFSv4 ACLs to support their semantics, it is now intended that support will be availble via an NFSv4.2 extension.

* Work was done to clarify the description regarding returning NFS4ERR_INVAL when inappropriate attributes are specified for (N)VERIFY.

B.5.12. Changes Made in Draft -11

A number of changes were necessary to clarify/correct issues with the text in [RFC8881] in responnse to Working Group questions. These questions did not arise as part of the rfc8881bis effort, but suggest areas where the current specfication is inadequate.

* Deal more thoroughly with situation regarding the interactions of delegation and REMOVE where the suggested recall can be safely dispensed with.

* Clarify/correcr the semantics of PRESERVED_UNLINKED to be POSIX-compatible, avoiding a gratuitous "MUST NOT".

* Restructure/revise the Implementation description sections for the REMOVE and RENAME operations.

  This involved a reorganization of REMOVE to treat possible rejections first and and a clear sepration of the various REMOVE-induced file system changes.

  RENAME was clarified to refer to REMOVE regrding the handling of files removed because there were renamed-over. This address issues regarding delegation non-recall and handling of deletion upon last close.

In addition, Appendix C.2.8 was created to provide helpful background for the review of the above changes.

B.5.13. Changes Made in 8881bis Draft -00

This document as been renamed as rfc8881bis since RFC8881 is obsoleted by it, despite the fact that the issues it addresses were introduced in RFC5661.

In addition, anumber of small clarification/corrections were made in this draft.

* It needed to be made clearer how unsupported attributes are dealt with when requested by READDIR. Unfortunately, [RFC8881] saus very little about the attribute fetch feature, leading to confusion aout the case in which the set of supported attributes changes as result of crossing a mount point.

* In the discussion of pNFS, the text is not clear as to the client's and server's responsibilities with regard to shared state and did not provide normative guidance, substituting implementtion advice in its place.

* Some work was done regarding pNFS teeminology, following Tom Haynes' suggestions.

* A discussion of future work with regard to [RFC8434] has been added to Appendix C.2.10

Appendix C. Issues Requiring Further Discussion

This Appendix discusses issues that the working group needs to discuss before making decisions regarding potentially necessary specification changes. Despite the need for working group decisions on certain policy matters, some of the specific examples cited have already been addressed by revised text within the draft specification proper.

C.1. Appropriate Uses of RFC2119 Keywords

Although, as stated in Section 1.1, this document intends to use these keywords as described in RFC2119, there are a number of issues that have resulted due to uses of these keywords in RFC5661 and RFC8881 that may not be clearly in accord with these definitions, possibly requiring some corrective action, once the working group has reached a consensus regarding the appropriate path forward.

* Because of a lack of clarity within RFC2119, there is considerable uncertainty about appropriate situations in which to use "SHOULD" and "SHOULD NOT", resulting in a number of cases in which they are inappropriately used or in which it is unclear whether particular uses are appropriate.

  The working group needs to discuss these examples (see Appendix C.1.1) so that these terms can be used consistently in this specification and within the boundaries established by RFC2119, even if those boundaries have some level of uncertainty surrounding them.

The use of the related term "RECOMMENDED" in connection with file attributes is not included in the above discussion, since it is already clearly understood that this use is incorrect.

* Although RFC2119 is appropriately clear, there are a number of cases in which uses of "MUST" and "MUST NOT" are problematic, since they are used in RFCs 5661 and 8881 in ways not in accord with their definition while the existence of clients and servers that ignore such statements gives one reason to doubt whether these are truly required for successful interoperation.

  The working group needs to discuss these examples (see Appendix C.1.2) so that such uses are corrected and to reduce the probability of similar occurrences in the future.

* Even apart from the definitions of these keywords, there is the further statement in RFC2119 that these terms are to be used "sparingly". Given the size of the v4.1 specification, it is desirable that all contributors adopt a common approach to issues about where these terms are appropriately used.

  The working group needs to discuss the issues described in Appendix C.1.3) so that the new specification has a consistent approach to these matters.

C.1.1. Appropriate Use of "SHOULD" and "SHOULD NOT"

RFC2119 defines "SHOULD" as follows, with the definition of "SHOULD NOT" paralleling it.

> This word, or the adjective "RECOMMENDED", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

This definition makes it clear how "SHOULD" differs from "MUST" but the specific difference with "MAY", while these terms are clearly intended to be distinct, is left unclear. Since it would not normally be expected for the other peer to be able to judge the validity of the reasons chosen by the SHOULD-using peer (or even whether the full implications of the choice made have been understood and carefully weighed by the peer’s implementer), the other peer is in the same position as it would have been if "MAY" had been used. It needs to be prepared for the "SHOULD" to be followed or not followed, as the SHOULD-directed peer chooses.

Although one gets the sense that not following "SHOULD" or "SHOULD NOT" is in some way disapproved of, since one does not to have "valid reasons" to either follow or not follow a "MAY". However, this leaves a great deal of uncertainty remaining as to when "SHOULD" is justified, especially given the indication within RFC2119 that these terms are to be used "sparingly".

One class of cases in which "SHOULD" is appropriately used, since not following such a directive might have the ability to cause harm, has to with situations in which security is an issue and some uses of "SHOULD" in the existing NFSv4.1 specifications fit this model. However a survey of the NFSv4.1 specifications shows many uses that take different approaches, some of which are clearly wrong and others which we need group discussion to establish a specification-wide policy:

* The statement "the client and server SHOULD use long-lived connections for at least three reasons" appearing in Section 2.9.1 of [RFC8881] raises a number of issues that make use of "SHOULD" questionable.

  There is no clear definition of "long-lived connections", making it hard to determine, in any particular case, whether the "SHOULD" has been adhered to or not. As a result, it might not be clear whether a particular implementation’s connections are long-lived leaving it unclear whether the "SHOULD" is being adhered to, so that the full implications of not adhering to it might not be obvious to those implementations not very clear about whether they are adhering to the guidance or not.

  It is hard to imagine what might valid reasons to ignore the reasons given, which are valid and worth mentioning, although there might be implementation considerations which cause connection lifetimes to be shorter than they would be otherwise.

  Overall this seems like useful implementation advice and could appropriately use the word "should" or a synonym.

* The statement "Instead, the replier SHOULD return an appropriate error (see Section 2.10.6.1 [Appears in this socument as Section 7.6.1]), or it MAY disconnect the connection" appearing in Section 2.9.1 of [RFC8881] raises a number of important issues.

  It is hard to imagine what might be valid reasons to either return an inappropriate error or no error.

The intention behind the "MAY" seems clear but given the definition of "SHOULD", it isn't clear exactly what item is to be ignored or what sort of knowledge of the implications would be necessary if that item were to be ignored.

If the construction were reordered to clarify it and so take disconnection off the table immediately, then it would be unclear how the "SHOULD" could be validly ignored, since it is stated elsewhere that the replier "MUST NOT" silently drop the request

Possible replacement text is discussed elsewhere in connection with an adjacent "MUST NOT" which is dubious as well.

* The statement "NFSv4.1 clients SHOULD NOT use the RPC binding protocols as described in RFC1833" appearing in Section 2.9.3 of [RFC8881] is confusing and appears not to be in accord with our understanding of RFC2119.

  Unlike other cases of "SHOULD", it does not seem that the server, unaware of the possibly valid reasons to ignore the "SHOULD", is being asked to essentially treat this as it would a "MAY".

  Perhaps something like the following would be needed to give appropriate guidance to the client and server implementers without use of RFC2119 keywords.

  > The use of a reserved port has been common for NFS implementations and it is expected that this will apply to NFSv4.1 as well. While the use of RPC binding protocols as described in RFC1833 [RFC1833] is a possibility, there is no requirement that servers provide support for it. In light of this, a client should avoid such use unless it has good reason to expect such support to be present.

* The statement "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 7.6.1). This way, if all RDMA credits are used, the non-RDMA connection can have at least one outstanding request" appearing in Section 2.10.3.1 of [RFC8881] presents another interesting use of "SHOULD" that the working group should consider as it decides how this term is to be used in the NFSv4.1 specification.

  The second sentence, indicates a generally desirable outcome, but its nature raises considerable doubts as to whether this is anything other than helpful implementation advice.

The fact that the RDMA credits are subject to change and that the client and server may have different views of this quantity make is hard to understand what exactly is being recommended and part of the implementation would be responsible for its implementation.

Overall, "should" seems a valid replacement , although rewriting the sentence to use the phrase "it would be helpful if" also seems possible.

C.1.2. Uses of "MUST" and "MUST NOT" that are Problematic

While the definitions of "MUST" and "MUST NOT" are quite clear, there are still instances within the existing specifications in which it not clear that particular uses are appropriate or in which common client and servers do not follow the offered direction while interoperating successfully.

Some interesting examples from RFC8881 [RFC8881]) follow. Note that, unlike the case in Appendix C.1.1 which looked at each instance of the target terms in a given section of the document, here we only look at a subset of uses which appear ,in some way, spurious or otherwise questionable.

* There are reasons to question to use of "MUST" in the following statement appearing in Section 5.7.1 of RFC8881:

  > 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.

  This statement would make invalid the use of NFSv4.1 using RPC-over-RDMA when the RDMA connection is implemented using RoCE while allowing it for Infiniband and iWARP.

  Although the peer might depend on operating together with an implementation having adequate congestion control, there is no basis for requiring that specific protocols (i.e. SCTP and TCP) be used, particularly since RFC2119 indicates that these keywords not be used "to try to impose a particular method on implementers where the method is not required for interoperability".

  Regardless of ones judgment of the propriety of using "MUST" in this context, the working group needs to discuss and decide, by consensus, how to address the issue of RoCE use in supporting NFSV4.1 using RPC-over-RDMA.

* There are reasons to question to use of "MUST NOT" in the following statement appearing in Section 5.7.2 of this document and the similar statement appearing in Section 7.6.2.

  > A requester MUST NOT retry a request unless the connection the request was sent over was lost before the reply received.

  Given that the text states that this is to "reduce congestion", it is hard to see how the mandated behavior is an "absolute requirement of the specification."

  The following statement appearing in Section 7.6.2, phased as implementation advice, provides a positive explanation of the motivation, without making the use of "MUST" or similar terms, resulting in a shift between a normative introduction and the implementation advice providing the underlying substance:

  > Note that it is not fatal for a requester 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.

  Not mentioned in this section but one possible motivation for such a restriction is the potential need to simplify the work discussed in Section 7.6.1.3 particularly the possible need of the server to checksum data to be written to detect false retry, possibly undercutting the performance benefits of RDMA, as discussed in Appendix C.2.1.

  If the issues relating to limiting the work necessary to detect false retries is not an appropriate basis for this prohibition, it seems better to avoid a shift between a normative introduction and later implementation advice by saying something like the following:

  > Given that NFSv4.1 uses transport providing reliable delivery, there is little point retrying a request, except in cases in which the occurrence of a connection disconnect leaves the requester uncertain as to whether the initial request was successfully delivered. The session-based reply cache allows the replier to deal correctly with retries after reconnect whether the initial request was delivered and executed or not.

* The following statement, appearing in Section 5.7.2 RFC8881, leaves one uncertain about whether the use of "MUST NOT" is justified, since it gives no clear explanation of why the prohibited behavior is troublesome.

  > 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 3.9.6.1), or it MAY disconnect the connection.

  This uncertainty is exacerbated by the introduction which states, incorrectly, that this is "to reduce congestion" and that it is paired in a bulleted list with the previous statement using "MUST NOT" where its use is also problematic.

  It should be considered whether the explanation would be clearer if the focus is on the responsibilities of the replier in the session model, rather than on one particular case of the replier ignoring those responsibilities. One possible approach:

  > The replier MUST attempt to obtain and send a reply each compound request received. This applies with equal force to the case in which the request is a retry, with the instructions in Section 7.6.1.3 followed in generating the reply which the replier needs to send to the requester in all cases, except where a disconnection event makes this impossible

* The following statement, appearing in Section 5.7.2 of RFC8881, requires further analysis since the justification provided for the prohibition merely cites a possible difficulty, without consideration of whether this difficulty could be resolved without this prohibition.

  > 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 Is 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).

Beyond the problem with the justification provided, is the fact, that many clients, including those most commonly used, essentially ignore the "MUST NOT", yet successfully interoperate with most servers. This essentially makes the "MUST NOT" untenable.

There are two problems with the current justification:

- Only a single value is assumed as the sequence value to be chosen for the next request (i.e. two) while the possibility of the alternative choice (i.e. one) is not addressed at all.

- The occurrence of an error is treated as disposing of the matter, without consideration of potential recovery approaches.

It appears likely that whichever value is used as the next sequence, the resulting error is not fatal, making use of "MUST NOT" inappropriate. Possible replacement approaches will be discussed in Appendix C.2.1 and explored in a modified Section 7.6

C.1.3. Issues Regarding Use of RFC2119 Keywords "Sparingly"

RFC2119 contains the following statement:

> Imperatives of the type defined in this memo must be used with care and sparingly. In particular, they MUST only be used where it is actually required for interoperation or to limit behavior which has potential for causing harm (e.g., limiting retransmissions) For example, they must not be used to try to impose a particular method on implementers where the method is not required for interoperability.

The following issues make this statement difficult to interpret.

* In fact, none of the terms defined in this RFC is an "imperative". They range among adjectives, participles, and modal auxiliaries, making it hard to determine which terms are being referred to.

* The terms "MUST", "MUST NOT", SHALL", and "REQUIRED might be thought of loosely as imperatives, since they are directing implementers to do something or to not do something.

* Although the terms "SHOULD", "SHOULD NOT", and "RECOMMENDED" do not have the sense of imperatives, they might be thought of as fundamentally carrying an imperative message, albeit one with a rather unclear provision for the recognition of exceptions.

* The terms "MAY" and "OPTIONAL", cannot reasonably be considered imperatives. Furthermore, the final sentences of the paragraph do not really make sense when applied to uses of these terms.

* Although the paragraph would normally be read assuming that the subject of the first sentence (i.e. "imperatives of the type defined in this memo") and "they" as used in the final two sentences, designate the same group of terms, that may not be possible since "MAY" and "OPTIONAL", do not make sense in the final sentences while is hard to believe that the author really meant that these terms did not need to be used with care. The case of "sparingly" is not as clear cut but it is hard to conclude that only the first two classes of terms need to be used sparingly.

If these terms are to be used "sparingly", whether terms like "MAY" are included or not, a meaningful distinction must be made between things that are "an absolute requirement of the protocol" and the far more numerous set of things that simply describe how the protocol works. While it is required for interoperability that the client and server agree on the XDR for operations and results, and the actions to be performed for each operation, it is not clear how one could decide which of those interoperability requirements is "an absolute requirement of the protocol" meriting use the word "MUST", since deciding that they all do would not use these terms "sparingly" and is likely to result in an unreadable specification as well.

At times in the past there has been inconclusive working group discussion of the possible use the word "MUST" in connection with the need to return certain errors. While it was clear that the need for interoperability meant that this was a requirement within the definition of "MUST", there was concern about what that did to the style of explanation since returning errors, like setting appropriate operation parameters and results and performing the requested operations, are simply "the way the protocol works", and not an "absolute requirement of the specification" assuming those can be distinguished from ordinary requirements of implementing the protocol. The possible need to use such terms "sparingly" adds additional weight to this concern.

In any case, there seems to be a need for the working group to discuss and come to some consensus regarding the routine use of the word "MUST" even when the situation is not one which the question to be addressed is whether the definition of the word is adhered to, as discussed in Appendix C.1.2.

C.1.4. Going Forward Regarding Use of RFC2119 Keywords

Although many of the specific issue discussed here have been addressed, the working group needs further discussion in order to arrive at a consensus regarding policies to be followed on these issues in general.

C.2. Issues Regarding Proposed and Actual Changes

The subsections within this appendix each concern a set of changes that have been made to address various issues in the existing specification for NFSv4.1 [RFC8881] which are discussed in Appendices C.2.1 through C.2.8. Some, although not all of these, relate to matters raised in Appendix C.1.

Each of these require further working group discussion, although The nature of the discussion may vary, based on the nature of work to address the cited issues and the possibility that further related work might be required. The author assumes that, all these cases, leaving the material in the form it had in [RFC8881] would not be acceptable.

* For sub-sections in which the changes already been made in the current draft, if correct, fully address the issues now known, The necessary discussion will involve discussion of the proposed changes, in the form suggested in this draft in an attempt to move to a working group consensus on their correctness, adequacy, and clarity.

  These sub-sections includes Appendices C.2.1, C.2.3, C.2.4, C.2.5, C.2.7 and C.2.8 changes have already been drafted and appear in the current document draft.

* For some other sub-sections, the author is unsure whether the changes made so far, even if correctly done, fully address the underlying issue. In thee case the working discussion of the prposed changes will need to be combined with a discussion of whether further changes are necessary and possible in the context of the current document.

  These sub-sections includes Appendices C.2.2 and C.2.6,

C.2.1. Changes Regarding Request Aborts, Retries, and the Session Model

This issues discussed in this section have been dealt with by a set of changes to the document proper in Sections 7.6 and 7.6.1.3.1. These changes have been made in multiple drafts as described below and now need working group review to make sure that the changes made are adequate to deal with the problems in the existing text in [RFC8881]

* Previously, the question had been whether and how to deal with the misuse of RFC2119-defined keywords and the problem that the ability to terminate RPC requests, required by many client implementations, was inconsistent with exactly-once semantics, since zero times is not "exactly once".

* Now that there is an alternate approach available for consideration, the relevant question is whether that approach is adequate and how it might need to be changed. For more detail on the changes made, see Appendix A.3.

We deal here with three related issues that are connected, in some way, with the new session feature and the associated reply cache logic;

1. The potential need, as might be inferred from the discussion in Section 7.6.1.3 of [RFC8881], to checksum request data, particularly data to be written in order to eliminate the possibility of not actually acting on a request which is a "false retry", potentially resulting in data corruption.

2. The prohibition, discussed in Appendix C.1.2 on reissuing a request without the occurrence of a disconnection of the connection on which the request as issued.

3. The prohibition, discussed in Appendix C.1.2 on ceasing to wait for a response without actually receiving the response.

These issues have been addressed together since considering them in the context of the design of the sessions feature sheds light on the troublesome issues mentioned above. Specifically, we are looking at the possibility that we have arrived at a suitable framework to discuss these issues so that:

* This framework provides a more realistic and convincing explanation for any necessary prohibitions and/or recommendations.

* This framework allows such prohibitions to be safely downgraded to recommendations or implementation advice

* This framework might encourage client implementations to implement EOS without allowing the possibility of false retries, making it advisable for server implementations to avoid extensive recording of request contents or the checksumming of requests in order to prevent the undetected occurrence of false retries.

The issues and the changes made to address them can be summarized as follows:

* The use of "MUST" in Section 7.6 and force clients to wait for responses for all requests needed to be adjusted because many clients were unable to comply when tasks were terminated, rendering the requirement useless.

  In draft-01, the text was modified to make it clear that, realistically, there were situations in which the client could not wait forever and that real goal of the EOS logic was at-most-once semantics. In addition, the discussion now covers the possibility of getting SEQ_MISORDERED in this situation.

* The use of "MUST NOT" in Section 5.7.2 to prohibit retransmissions needed to be revised since sending such retries while undesirable does not cause the kinds of harm that use of the RFC2119-defined term implies exist.

  In draft-00, we eliminated the prohibition regarding retry and replaced it with implementation advice indicating why there is little reason for retries without clear motivation and explaining the unfortunate consequence of such retries.

* The text in Section 7.6.1.3.1 needed revision to make it clear when checks for false retries were either required or desirable and to make clearer how they could arise given the implementation of Exactly-once semantics.

  In draft-01, added an initial paragraph indicated reasons that checks for false retry result from implementation problems and that there are practical limits as to when they will be done.

  In draft-03, extensive changes were made to explain the situations in which false retry was a real concern and suggesting approaches to checking that can realistically be implemented. This includes addressing issues that were intended to be addressed by errata report 5982.

C.2.2. Issues Regarding Directory Delegation that Need to be Resolved

Directory delegations and notifications were added to NFSv4.1 but have never been implemented. During working group discussions of NFSv4 performance issues with regard to directory handling and later discussions with those working on implementations, it was discovered that there are a number of issues with regard to handling of directory delegations that need to be addressed, including cases where the design is adequate but substantial clarification is needed.:

* It was assumed that clients holding a delegation would maintain locally an image of the server directory, that needed to match that of the server with regard to directory entry order and the values of directory cookies.

  As implementation efforts proceeded, it became apparent that this assumption was unduly limiting and needed to be addressed. In addition, this approach made it unduly difficult to use the feature for file systems in which sets of distinct characters are treated as equivalent (i.e those supporting normalization-related processing or case- insensitivity.

  As a result, it was decided that allowance be made for clients using a different approach to caching, as described in Appendix C.2.2.2

* In addition, for clients that were prepared to maintain a local directory image, there were important gaps in the explanation that resulted in a lack of clarity regarding the ability of the notification scheme to allow the client image of the directory to be kept in sync with that of the server. It is not made as explicit as it might be that server support for continuation of directory delegations requires that the information provided in directory notifications is adequate to provide to the client the information needed to appropriately update the client’s image of the directory to so that it can serve in place of a READDIR to the server. This includes the ability to maintain a directory ordering matching that on the server and READDIR cookies that match those held on the server. In situations in which the changes to the directory are of such a nature that this sort of update cannot be done based on a directory notification, the directory delegation needs to be recalled and returned. With the clarified, the value of directory delegations in avoiding the need to refetch large directories because of a small number of directory changes, would be more obvious. See Appendix C.2.2.1 for some suggestions in that regard.

While it is unusual for CREATEs, RENAMEs, and REMOVEs to cause wholesale changes in the directory entry ordering or READDIR cookie values, there has previously been no way for the client to be sure that no such changes are being made, even when no other client is changing the directory. As a result, many clients are accustomed to refetch directories when they are changed, despite the consequent negative effect on performance.

As a result, it has seemed to many that there is little value in implementing directory delegations and notifications, leaving concerns about directory performance unaddressed.

* In addition, there is uncertainty as to whether a client making a change to a directory will receive timely notice of the details of the changes that will be made to the modified directory. The means by which notification is typically provided, using an asynchronous callback with provision for notification delay and batching of notifications was primarily directed to cases in which another client is making the modifications and the client receiving the notifications needs to sheltered from excessive notifications. This requires two issues to be addressed.

* It needs to made clearer that the batching and delay of notifications apply only to the notifications of directory attribute changes and not to those notifying the client of changes in directory contents.

  Although this might ultimately require a major rework of the text of Section 15.9, some useful suggestions can be found in Appendix C.2.2.4

* Since the notification model for changes in directory contents is an asynchronous one, it needs to be made clearer how clients making changes to directory contents can use these notifications can avoid refetching directory contents.

  Some suggestions regarding useful change in this area can be found in Appendix C.2.2.1

C.2.2.1. Clarifying the Role of Directory Delegations and Notifications in Avoiding the Need to Refetch Directory Contents

The first step is to clearly define the problem that content notifications address. This could be addressed by adding the following new paragraph to the end of Section 15.8.2:

Even when a client is certain that no other client is modifying a cached directory, there still might be a need to refetch the directory contents to satisfy additional READDIR requests. This is because there is no way to determine the modified ordering of the directory or the associated cookie entries using the knowledge of the directory changes requested and the corresponding responses. For a discussion of how directory delegation together with directory content updates can avoid the need to refetch directory contents, see Section 15.9.7.

In addition the following paragraphs need to be added at an appropriate place within Section 15.9.

When a directory delegation is held and notifications of changes of directory content updates are provided, the need to refetch the directory contents to satisfy READDIR requests can be avoided. This is of considerable benefit when the directories are large.

Although the directory content updates provided are asynchronous, they are not batched or delayed for considerable periods of time. Because of this, clients can keep track of the set of pending updates expected to avoid refetching directories when a content update is likely to enable the client to avoid the extra work.

C.2.2.2. Dealing with Various Client Caching Schemes

As a result of discussions with those involved in working on directory delegation implementations, it was discovered that:

* There are a significant set of clients that want to be aware of directory content changes but have no need for the position information currently provided since they either do not try to avoid repeated READDIRs by means of caching previous ones or synthesize READDIR results from cached contents without depending on the server’s choice of directory entry order or directory entry cookies.

  While the client is free to ignore position information provided, the effort to provide it where it is not needed might be a significant barrier to implementation.

* Some servers could produce useful position information with less difficulty if the directory cookies were defined so as to be acceptable to clients who do want to replicate the server’s directory entry order and cookie values.

* Also relevant are those clients who do wish to provide cached READDIR results conforming to the server's order (not formally required but shuffling these might not be accepted by some users).

Addressing these issues requires providing the server more flexibility in the form of notifications used to inform clients of directory content changes while respecting client needs, which might be different for different clients. One way of doing so which has already been explored would allow three forms of content update notifications, such as are listed below:

(A): Provision of position information in the way defined in [RFC8881].

(B): Provision of position information in a simplified fashion using directory cookies only and avoiding the need to provide the names of nearby entries.

This approach is valuable for use in the large class of servers for which the entry cookie value is monotonically increasing as successive directory entries are transmitted as part of READDIR.

(C): Omission of position information i content update notifications.

This approach is only useful to clients that do not try to maintain the server's directory order but are only interested maintaining the set of entries, independent of their order.

Selection of the proper format for any given update requires a new mechanism for the client server to arrive at the chosen format based on client needs and server capabilities. This mechanism is described in Appendix C.2.2.3. Because existing operations were specified without any provision for such selection, certain desirable options will only be available once a v4.2-based extension is available. However, as discussed below, there will be opportunities to avoid the Procrustean approach currently described in [RFC8881] in which only (A) is allowed.

In addition to the matters discussed above, the issues raised in Appendices C.2.2.5 C.2.2.6 need to be addressed.

C.2.2.3. Version Control and Extension

We need an mechanism that is acceptable in the v4.1 context to allow some limited extension of the approach to directory delegation specified in [RFC8881]. This needed to:

* Provide a way of including more substantial authorization support using additional notifications.

* Allow selection of the form of position information in content update notifications and to decide on the necessity of certain recalls based on the following factors:

  - Certain clients are concerned about the order of directory while other might not care.

  - There are clients concerned about entry order who want to have knowledge about server directory cookies, while there might be other that do not.

  - There are servers that maintain directory cookies so they are always monotonically increasing with directory position while there others where that connection cannot be relied upon.

  - There might be servers and clients written based on the approach of [RFC8881] rather than the more flexible one described in this document.

    We need to deal with this possibility even though there are good reason to believe that no such implementations exist. Until [RFC8881] is obsoleted, which could take a while, we have no way of tracking ongoing development activities.

Dealing with all of the above, in a general way, requires a general extension mechanism, the overall structure of which will be discussed below, while the details will be decided in later extension document.

As pat of this effort, we will need a way to provide greater flexibility, if possible, in an NFSv4.1 context, while interoperating correctly with client and server's using the [RFC8881] approach. This requires some way of distinguishing implementations without excessive XDR additions.

Regarding the selection of extension mechanism, it appears that the best approach is to generalize the use of the existing argument and result notification bitmaps. This requires only very limited XDR changes that follow the approach laid out in Section 9 of [RFC8178]. Bits can be defined without corresponding notifications and allowing other interface changes to be inferred from the presence or absence of particular bits.

For distinguishing new and old implementations, it seems the best approach is to rely on the notification bitmaps. Inclusion of an extension could signal client awareness of the extension with the appearance of the bit in the response signaling the server's knowledge of the extensions.

C.2.2.4. Clearly Separating Directory Content Updates from Other Notifications

Although an extensive set of small changes to clearly split content notifications from attribute notifications is probably necessary, to begin the clarification of this issue, Section 11.15 needs a clarifying final paragraph, to read as follows:

> Note that these attributes apply only to directory attribute notifications and not to directory content updates, which, although asynchronous are not subject to batching or explicit delays.

Also important are the following replacement paragraphs for the first two paragraphs of the IMPLEMENTATION section of GET_DIR_DELEGATION, Section 23.39.4.

> 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 provide a client a way to maintain an accurate client-side image of a changing directory (through client content updates) and to reduce network traffic as well (through both sorts of notifications)
>
> Directory update 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), and renames (NOTIFY4_RENAME_ENTRY). Cookie verifier changes, although not directory content updates, can be obtained by setting NOTIFY4_CHANGE_COOKIE_VERIFIER gdda_notification_types field. All of these notifications are asynchronous but they are not, like directory attribute notifications, subject to batching or time-based delays.
>
> Directory attribute changes are requested using the notification type NOTIFY4_CHANGE_DIR_ATTRIBUTES), with the specific attributes that require notifications specified by gddr_dir_attributes. Like

the child attribute notifications discussed below, these notifications are subject to matching and time-based delay in order to limit network traffic.

C.2.2.5. Directory Delegations and Permissions

It appears that Section 15.9 was written without sufficient attention to authorization issues that arise when LOOKUP, READDIR, GETATTR, and ACCESS operations are satisfied from cached data.

As a result, significant work will need to be done in related subsections to address that gap. This will inevitably, have to involve consideration of the increased difficulty of dealing with situations in the presence of ACLs. Given the current uncertain state of ACLs, this will require, at least for NFSv4.1, steps prohibit or give permission to the server to prohibit use of directory delegations in situations in which their existence might compromise needed authorization restrictions.

C.2.2.6. Going Forward Regarding Directory Delegations

The material in this section should provide a suitable basis for working group discussion, in the hope that it will enable those changes to be moved into the specification proper in a later draft revision.

Once that work is done, the working group will be able to decide;

* Whether, with implementation of directory delegations, NFSv4.1 still has a directory performance problem that needs to be addressed

* Whether there is a need for extensions to improve directory delegations (synchronous notifications).

* Whether additional directory performance features are worth pursuing.

C.2.3. Changes Regarding Memory Mapping

It has been necessary to make major changes to material currently dealt with in Section 10.7 of RFC8881. The replacement is Section 15.7. Extensive changes have been necessary for the following reasons:

* The previous text consistently ignores the need for those reading and writing files to open them.

As a result, many the problems the previous section was concerned with, regarding a concurrently held write delegation only apply in the unusual case of files being read using special stateids.

* There had been assumption that CB_GETATTR would always be used when attributes are interrogated, ignoring the possibility of the delegation being recalled.

  This ignored the fact that CB_GETATTR is an OPTIONAL feature and that there is no requirement for clients implementing to use it for access and modified time.

* In citing byte-range locking, there was no consideration of the fact that none of the cited issues poses any difficulty in the case if advisory byte-range lock.

* The treatment of mandatory byte-range locking assume, incorrectly that it requires as part a each IO, that a lock be obtained to enable that operation.

  In fact, mandatory byte-range locking only requires that no inconsistent lock be held by another process performing IO.

  As a result most of the potential issues cited do not exist for NFSv4.1 and if they did, they would apply to local IO as well, making this sort of locking untenable.

C.2.4. Issues Regarding Handling of Persistence

There are a number of important issues relating to persistence that need working group discussion and corresponding specification changes. These involve both reply cache persistence and the potential persistence of locking state to allow lock reclaim to be avoided.

The treatment of issues related to the persistence of protocol data shared by the client and server needs substantial remedial work, as described below. Without such remedial work, we would be stuck with a confusing description of a hypothetical feature that has never been implemented and has no prospect of being implemented, whose description is unusable due to confusion about the handling of locking state persistence. The issues to be addressed include the following;

* The excessive demands as to request atomicity and continuation across server restart, leading to a feature which cannot be implemented, as described in Appendix C.2.4.2.

* Ambiguity about the possibility of transparent state recovery and the means by which the client might be informed of its existence, as described in Appendix C.2.4.1.

* The confusion about the role of the clientid in connection with session recovery as described in Appendix C.2.4.3.

An alternative approach to these issues is presented in Appendix C.2.4.4. and will be the basis for a revised Section 8.

C.2.4.1. Ambiguity Regarding Locking State Persistence

As to the potential persistence of locking state, current specifications are unclear, mostly due to different handling of the issue in different sections and undue focus on what the server will provide with no attention to the question of how the client finds out about persistence or the lack thereof and deals appropriately with the situation.

First of all, the section entitled "Client Identifiers and Client Owners"(Section 2.4 in [RFC8881] and 5.5 in this document) gives the impression that, in the event of a server restart, the client will inevitably find out, by getting an NFS4ERR_BAD_CLIENTID error that locking state has been lost. While there is no explicit statement to this effect, the presentation of expected sequences of events (there are separate discussions of this for the cases of persistent and non-persistent sessions) leads one to suppose alternatives are not anticipated.

On the other hand, the section entitled "Loss of Session" (Section 2.10.13.1.4 in [RFC8881] and 7.13.1.4 in this document strongly suggests that the NFS4ERR_BAD_CLIENTID is not inevitable, opening the way for locking state to be persisted across a server reboot, even though there is no explicit statement allowing servers to do so.

Given this divergence, it makes sense to determine which of these approaches is correct and make explicit descriptions of recovery make clear how clients are to deal with servers that do maintain state across reboot and avoid reclaim just as they do in the event of migration. A part of that discussion will concern potential compatibility issues which are not troublesome if clients do follow the approach laid out in the loss-of-session section

Adding to the existing confusion are occasional references to the possibility of certain forms of state persistence, with no discussion of how the client might find out about this potentially persistent state. For example, Section 23.43.3 contains the following paragraph:

> 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 13.4.2.1).

While this strongly implies that metadata servers could persist layouts across server failure, given the existing confusion it is hard to see how clients could effectively use this functionality or why server might provide it other than by providing a general lock persistence.

C.2.4.2. Implementability of Persistent Reply Cache as Currently Described

Another important topic of discussion concerns a number of statements in the section entitled "Persistence" (Section 2.10.6.5 in [RFC8881] and 8 in this document, which make implementation of persistent reply caches significantly harder than it needs to be or give the reader the impression that it is nearly unimplementable. This might have led to lack of implementation effort as part of a vicious spiral, that might result in the loss of this helpful feature, that needs implementation to take advantage the availability of lower-latency persistent storage. The following issues need to be addressed:

* One concern is that the statement "The execution of the sequence of operations (starting with SEQUENCE) and placement of its results in the persistent cache MUST be atomic" might convince the reader that the execution of each COMPOUND needs to be atomic as well, making conformance difficult and would seriously undercut any attempt to provide file system parallelism.

  This might not have been the author’s intention, even though it is the most natural reading of the sentence in question

  There are necessary atomicity guarantees required but they have to be more limited and explicit to make implementation possible.

* Even more troubling is the issue raised in the statement "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". The text goes on to say, as mildly as possible, "This creates an even higher challenge for atomic execution and placement of results in the reply cache.", but the indicating that

this is a greater challenge is likely to convince most reader that the feature is essentially unimplementable. The rest of the paragraph gives no reason to expect something workable except in special environments in which implementing this feature is the only goal.

Fortunately, the essential unimplementability derives, not from the feature but from the assumption that COMPOUNDs be executed atomically across a server restart rather than being terminated as part of the termination of the previous server instance, which this paragraph assume will never happen.

There needs to be a way to terminate CONPOUNDs still active at the time of server reboot, if there is no way to forbid execution of such troublesome COMPOUNDs.

* The final paragraph does nothing to correct this impression of unimplementability.

  First, it says the following which would be unexceptionable for features for which there is at least one way to implement them: "While the description of the implementation for atomic execution of the request and caching of the reply is beyond the scope of this document".

  Following this, it drives the final nail into the coffin of this feature by saying "An example implementation for NFSv2 [45] is described in [46]". The important fact here is that NFSv2 does not have COMPOUND, allowing the troublesome atomicity and cross-server-instance request continuity issues dealt with in the first two paragraphs to be bypassed, making this citation in this context inapposite.

C.2.4.3. Confusion about Clientid Role in Persistent Reply Cache

The existing discussion of reply cache persistence describes two possible variants that primarily differ as to the persistent storage of clientid-related information, with each variant deficient in some important way. This leads us to conclude that confusion about the role of the clientid and clientid-scoped state information and its persistent storage was not taken into account when this arrangement was arrived at. For example:

* While the persistent recording of some clientid-related information is presented as part of the second option given, there is no mention whatsoever of clientid-scoped locking state and its persistent storage.

Given the lack of explicit discussion of these matters, it is hard to tell whether it was expected that the server would or could persist this state.

* If clientid-related information is not saved (i.e. the first option), then the persistent reply cache does provide EOS across the server failure but does not allow the existing session to be used for new requests.

  While this is a valid use case for clients worried about the possibility of EOS problems across server failures, the fact that there is no locking state persistence means that server failure will disrupt operation by requiring a grace period before, for example, opening a file. However, giving this limited benefit, it is troubling that the existing spec, dure to confusion about clientid state, requires persistent recording of idempotent non-modifying operations (e.g. READs) with no real benefit because taking advantage of that benefit would require issuing new requests and checking their sequence ids against a persistent sored sequence id.

C.2.4.4. Replacement Approach to Persistence in the Case of Server Failure

Regardless of the original intent with regard how to how these two aspects of data persistence were to be tied together, it seems that these need to be defined as independent features. Even if one could determine that these were intended to be tied together, which seems unlikely it is not possible to tie these two aspects of data persistence, each with its own scope, together at this point. Making that choice now would undercut the adaptation of NFSv4 to more available low-latency persistent storage in a number of ways:

* Since the amount of locking state is not bounded, there would need to be accommodations to the situation in which a surfeit of locking state makes use of persistent storage impossible. If these two sets of state were tied together, such situations would unnecessarily interfere with the a need to provide EOS semantics across server failure.

* In the context of clustered servers, the loci for the update of session-related and clientid-related data might be different, especially where clientid trunking is used. In such situations, there will inevitably be occasions where only one of these two forms of persistence is implemented.

* Given that the flow of locking operations is often a small part of the total and likely to be below the total of non-idempotent and modifying operations well, for many server implementors it would have a higher priority for implementation and use, As a result, it reasonable to expect servers that implement it without implementing reply cache persistence even after the issues discussed in Appendix C.2.4.2 are successfully addressed

If persistence of locking state is to be made available as its own feature, allowing clientids to persist across server failure, then it is necessary to decide how to deal appropriately with the existing two options for session-based state persistence, once the issue of clientid-based state persistence is put aside.

* Persistence of the reply cache (only) will still be a viable useful option, not providing session continuation across server failure.

  In defining this as a possible choice, it needs to be stressed that servers aiming to provide this functionality to not need to persistently store changes in sequence ids that would not be perceivable by a reconnecting client.

* Full session persistence would remain an option, even though there would be no need to persist data beyond the reply cache, current sequence id array and clientid.

  Actual use of a persistent session would require persistence of the associated clientid-based locking state information. However, the server would not commit itself to maintain this information at session creation time and would find out if full session continuation was available, at the point at which the first new requests were issued

C.2.5. Changes in Attribute Categorization

The following changes in the categorization of attributes have been completed in draft -03 nut need to be further discussed by the working group. That discussion might involves other documents in addition to this one.

* The detailed description of authorization-related attributes has been moved to the documents [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls].

  In line with this shift, the attribute categorizations have been made the province of [I-D.dnoveck-nfsv4-security] with that controlling in the event of any conflict.

* The attributes mode, owner, and owner_group have been made REQUIRED rather than RECOMMENDED (with the meaning OPTIONAL)

* The attributes acl, sacl, and dacl have been described as "Experimental" in NFSv4.1, since, unlike other OPTIONAL attributes, the existing specifications do not describe the attribute sufficiently to allow interoperable client and server implementations to be developed.

C.2.6. Changes in Treatment of Attributes for Named Attribute Directories

Previous specification were self-contradictory in that:

* There were statements that made SETATTR and GETATTR on named attribute directories were undefined operations.

  The explanation offered for this exclusion did not make sense. For an explanation of why this text was eventually removed See Author Aside #66a in Section 5.3.5 of [I-D.dnoveck-nfsv4-security].

* There were other statements saying that named attribute directories could have attribute and stating that they were to include all the REQUIRED ones.

Changes have been made to eliminate this contradiction, as described below:

* The statement regarding SETATTR and GETATTR on named attribute directories being undefined operations was retained although the text "explaining" this exclusion was deleted.

  Since this material had been moved to [I-D.dnoveck-nfsv4-security], the replacement appears in Section 5.3,5 of that document.

* The statements about the ability to have (non-named) attributes for the named attribute directory in Section 11.7 have been deleted.

Discussion of these changes needs to continue to address the following issues:

* Even though the contradiction has been resolved, it is not certain why the exclusion is justified, given the inadequacy of the existing explanation.

Providing the ability to access and modify the attributes associated with named attribute directories might address some of the authorization isues discussed below, but could be expected to add additional complexity.

Leaving this as it is in the current set of documents would avoid additional complexity but still make it possible to reference named attribute directory attributes as part of dealing with authorization of operation involving the named attribute directory.

* There is no existing discussion of POSIX-based authentication of operations involving the named attribute directory, leaving a gap that needs to be filled.

  Using the mode, owner and owner_group attributes of the base object in place of those for the named attribute directory runs into troublesome issues since the X bit, controlling exec privileges for a (non-directory) base file controls lookup for the named attribute directory.

  Any necessary changes will be made as part of Consensus Item #66 in [I-D.dnoveck-nfsv4-security].

* There exist ACE mask bits devoted to control of named attribute directories but it is clear that some changes need to be made.

  As currently defined, these bits only control access to and creation of a named attribute directory, while allowing creation of new named attributes without authorization controls. Cleaning this up will be easier when we know how implementations behave but so far, none have been found.

  Any necessary changes will be made as part of Consensus Item #100 in [I-D.dnoveck-nfsv4-acls].

C.2.7. Changes Made as a Result of REJECTED Errata Reports

Changes were made in response to the errata reports listed below, each of which was assigned a REJECTED status. The author, based on his own sense of the working group's need and wants, has addressed those errata reports. Even though there is no reason to suppose these reports do not need to be addressed, their rejection needs to be addressed by establishing that their is a working group consensus to make the change.

* Errata report 2722, reported by Ricardo Labiaga, was an editorial change that was rejected for reasons that are not clear. In addition, it is also unclear why this report was retracted.

  In any case, the author addressed the troublesome area in way he feels satisfactory without necessarily following the text in the retracted report. The rejection, whatever its motivation implies we need a working group consensus as to the acceptability of the change made.

* Errata report 2751, reported by Ricardo Labiaga, was a technical change that was rejected because it proposed a substantive change in the handling of LAYOUTCOMMIT. Despite this justified rejection, it appears that implementation adopted the suggested approach, making it necessary that we, even at this late date, to adjust the specification so that implementation and specification no longer differ.

  This specification draft has adopted the proposed changes without major changes except in one respect: The prposed new section, slated to be part of pNFS chapter will be done as part of the chapter devoted to the pNFS files layout. In addition, the presentation of changes in the form of text replacement complicated the process by requiring decompilation of the changes into xml. While the author did as well as he could, the complexity of the process calls for extra review.

  In any case, further review of these change is necessary to make sure that the resultant text has working group consensus.

* Errata report 5982, reported by David Noveck was a technical change that was rejected. As things turned out the proposed text was misguided and was dropped.

  Although other changes were made with same ultimate motivation and do need review, no special review is needed based on the rejection of the errata report.

C.2.8. Changes Made to Address Problems in Description of REMOVE/RENAME

In adddition to the restructurings/clarifications discussed in Appendix B.5.12, there are a number of substantive changes for which a consensus needs to be arrived at. These include changes in which the change is arguably substantive because ambiguities in the existing text makes it hard to determine whether the exististing text is or is not compatible with the new treatment.

* A change of approach to the PRSERVE_UNLINKED flag, making it only apply to the OPEN which returned it. This avoids dealing with the possibility of it being returened differently for successive opens and makes it clear why it does not apply to NFSv3 or NFSv4.0 OPENs.

* Providing rules allowing the recall of delegations help by a client doing a REMOVE can be dispensed with, depending on the nature of the server’s restrictions regarding REMOVE of open files.

  This includes a requirement for the client holding the delegation and doing a REMOVE to make the server aware of any OPENs denying READ or WRITE.

There arso a set of potential changes that might be made once it is clear whether or not compatibility issues would prevent any substantve change.

* Making it explicit that the restrictions regarding removal of a renamed-over file are iddentical to those removed using REMOVE.

* Applying similar logic regarding the non-recall of delegations for OPENs of the file being removed.

It is importnt to note that the need for s write-delegation-holding client to make the server aware of OPENs denying write before doing a removal operation potentially raises compatibility issues. However, the practical problems arsing are likely to be small since:

* A large portion of existing clients do not issue OPENs denying read or write.

* For those that do issue them, the complexity cost of doing so locally is likely to inhibit broad use of this technique. If a client were to do this locally, it would be taking on the work of the server in dealing with multiple processes doing such OPENs and corresponding CLOSEs, and detecting conflicts. This is unlikely to be wirth doing since the benefit is likely to be quite small.

C.2.9. Changes Made to Address Lack of Clariy Regarding "The Forgetful Model"

Section 12.5.3 of [RFC8881] (and Section 17.5.3 of the previous draft this document) contains the following statement;

> 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 (Secttion 12.5.5.1/>), it MUST first update its record of what ranges of the file’s layout it has before using the seqid.

This statement raises the following issues/questions:

* There is no explanation of what "using the forgetful model" means and how it might differ from a presumably more persistent model.

* There is no explanation of how a client might choose a particlar model and how a server might be affected by the client’s choice of model.

* The sentence fragment "if the client is not using the forgetful model (Section 12.5.5.1), it MUST" is hard to understand since "MUST",according to [RFC2119] is used to introduce "a fundamental requirement of the specification". Given that definition, it is hard to understnd how or why a client not following this requirement might be excused due to following this model.

* The section refernced in the troublesome text (i.e. Section 12.5.5.1), does not define "forgetful model"

* The title of the referenced section, referring to "Recall Robustness" seems to make its use where recalls are not involved, confusing.

In order to address the existing confusion and provde suitable normative text regarding the layout sequenceing requirements and server and client requiremets regarding obligations to retain layout information (independent of a presumed model choice), we are doing the following:

* Rename the sections so it is clear that it is about possible protocol requirements (and non-requirements).

* Re-organizing the section so that we are clear that what was previously referred to as "assumption" which was not aways the case is now explicitly a non-requirement.

* Put all the normative requirements together near te start of the section.

* Retain the useful implementation suggestion but make it clearer that this mateial is non-normative.

* Divide the previous ancillary section regardingg recall robustness into two sections, one about retention requirements and another anout recall/return interaction.

C.2.10. Need for Replacement of RFC8434

Discussion of recent drafts has made it clear thar further work is neeeded to provide an up-to-date replacement for [RFC8434]. For that reason, this draft is no longer marked as obsoleting RFC8434, with an intention to renew that designation in a later draft.

Overall, it seems best if most of the material in this RFC is placed in a new Chapter after the pNFS chpter and before the pNFS files chapter.

The new placement, the passage of time and existence of new layyout types leads to a number of issues that will need to be addressed when this new capter is added.

* The use of RFC2119-defined keywords wthin [RFC8434] needs further discusssion because these keywords are defined as applying to implementations rather than possible future documents.

  The Working Group faced a similar issue in the drafting of [RFC8178]. During discussion of that document some group members argued that each working group was free to specify things as it might chooose and it was inappropriate for one Working Group to foreclose what a future working group might do.

  As a result we adopted the apprach documented in Section 2.1 of [RFC8178], which might be adopted in a replacement for [RFC8434]

* The nature of this respecfication means that we can no longer assume that the new section updates rfc8881bis, since it is part of rfc8881bis.

  As a result there is no longer a meaningful disinction between Sections 3.1 and 3.2 of [RFC8434]

  Although Sevtion 4.2 and 4.3 still make sense, section 4.1 makes more sense in the pNFS files section. With regard to flexible files, it should only need a brief reference to [RFC8435].

Acknowledgments

Acknowledgments for This Update

The author wishes to thank Tom Haynes of Hammerspace for drawing the working group's attention to the fact that internationalization and security might best be handled in documents dealing with these individual protocol areas, addressing those issues as they apply to all NFSv4 minor versions.

The author wishes to thank Rick Macklem for his help in resolving the previous confusion regarding the proper timing for use of the PREV_DELEG claim types.

The author wishes to thank Rick Macklem and Trond Myklebust of Hammerspace for their helpful discussion of issues related to the existing text regarding REMOVE and RENAME.

The author wishes to thank Olga Kornievskaia of Netapp for her insights regarding the existing prohibition on ceasing to wait for a request that has not yet been replied to.

The author wishes to thank all those who contributed corrections/ suggestions to drafts of this specification, including Chuck Lever of Oracle and Yang Jing of HuaWei.

Acknowledgments for Previous Specification Documents

In addition to the authors/editors, the following people made important contributions to RFC 5661:

* The initial text for the SECINFO extensions were edited by Mike Eisler with contributions from Peng Dai, Sergey Klyushin, and Carl Burnett.

* The initial text 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 text 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 Brown, and Jon Haswell.

* The initial text 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 text 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 the IETF that formed the basis for the pNFS work in NFSv4.1.

* The initial text for the parallel NFS support was 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 that contributed to the construction of the initial pNFS drafts; specific acknowledgment 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 documents.

The following contributions regarding work done in RFC8881 need to be acknowledged:

* The important role of Andy Adamson of Netapp in clarifying the need for trunking discovery functionality, and exploring the role of the file system location attributes in providing the necessary support.

* The work of Xuan Qi of Oracle with NFSv4.1 client and server prototypes of Transparent State Migration functionality.

* The comments of Trond Myklebust of Primary Data related to trunking helped to clarify the role of DNS in trunking discovery.

* Rick Macklem’s comments brought attention to problems in the handling of the per-fs version of RECLAIM_COMPLETE.

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 replace all occurrences of RFCTBD20 with RFCyyyy where yyyy is the RFC number of the document providing descrptions of NFSv4 internationalization, currently expected to result from completion of the document referenced in [I-D.ietf-nfsv4-internationalization] or a document replacing that one.

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCTBD21 with RFCyyyy where yyyy is the RFC number of the document providing an overall description of NFSv4 security, currently expected to result from completion of the document referenced in [I-D.dnoveck-nfsv4-security] or a document replacing that one.

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCTBD22 with RFCyyyy where yyyy is the RFC number of the document providing descrptions of ACLs, currently expected to result from completion of the document referenced in [I-D.dnoveck-nfsv4-acls] or a document replacing that one.

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCTBD20 with RFCyyyy where yyyy is the RFC number of the document providing updated XDR for NFSv4.1, currently expected to result from completion of the document referenced in [I-D.dnoveck-nfsv4-rfc5662bis] or a document replacing that one.

Author’s Address

David Noveck (editor)
NetApp
201 Jones Road
Waltham, MA 02451
United States of America
Phone: +1-781-572-8038
Email: davenoveck@gmail.com