NFSv4 T. Haynes, Ed.
Internet-Draft Primary Data
Intended status: Standards Track April 13, 2014
Expires: October 15, 2014

NFS Version 4 Minor Version 2
draft-ietf-nfsv4-minorversion2-22.txt

Abstract

This Internet-Draft describes NFS version 4 minor version two, focusing mainly on the protocol extensions made from NFS version 4 minor version 0 and NFS version 4 minor version 1. Major extensions introduced in NFS version 4 minor version two include: Server-side Copy, Application I/O Advise, Space Reservations, Sparse Files, Application Data Blocks, and Labeled NFS.

Requirements Language

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

Table of Contents

1. Introduction

1.1. The NFS Version 4 Minor Version 2 Protocol

The NFS version 4 minor version 2 (NFSv4.2) protocol is the third minor version of the NFS version 4 (NFSv4) protocol. The first minor version, NFSv4.0, is described in [I-D.ietf-nfsv4-rfc3530bis] and the second minor version, NFSv4.1, is described in [RFC5661]. It follows the guidelines for minor versioning that are listed in Section 11 of [I-D.ietf-nfsv4-rfc3530bis].

As a minor version, NFSv4.2 is consistent with the overall goals for NFSv4, but extends the protocol so as to better meet those goals, based on experiences with NFSv4.1. In addition, NFSv4.2 has adopted some additional goals, which motivate some of the major extensions in NFSv4.2.

1.2. Scope of This Document

This document describes the NFSv4.2 protocol. With respect to NFSv4.0 and NFSv4.1, this document does not:

- o describe the NFSv4.0 or NFSv4.1 protocols, except where needed to contrast with NFSv4.2

- o modify the specification of the NFSv4.0 or NFSv4.1 protocols

- o clarify the NFSv4.0 or NFSv4.1 protocols. I.e., any clarifications made here apply to NFSv4.2 and neither of the prior protocols

The full XDR for NFSv4.2 is presented in [NFSv42xdr].

1.3. NFSv4.2 Goals

The goal of the design of NFSv4.2 is to take common local file system features and offer them remotely. These features might

- o already be available on the servers, e.g., sparse files
- o be under development as a new standard, e.g., SEEK_HOLE and SEEK_DATA
- o be used by clients with the servers via some proprietary means, e.g., Labeled NFS

but the clients are not able to leverage them on the server within the confines of the NFS protocol.

1.4. Overview of NFSv4.2 Features

1.4.1. Server-side Copy

A traditional file copy from one server to another results in the data being put on the network twice - source to client and then client to destination. New operations are introduced to allow the client to authorize the two servers to interact directly. As this copy can be lengthy, asynchronous support is also provided.

1.4.2. Application I/O Advise

Applications and clients want to advise the server as to expected I/O behavior. Using IO_ADVISE (see Section 14.7) to communicate future I /O behavior such as whether a file will be accessed sequentially or randomly, and whether a file will or will not be accessed in the near future, allows servers to optimize future I/O requests for a file by, for example, prefetching or evicting data. This operation can be used to support the posix_fadvise function as well as other applications such as databases and video editors.

1.4.3. Sparse Files

Sparse files are ones which have unallocated data blocks as holes in the file. Such holes are typically transferred as 0s during I/O. READ_PLUS (see Section 14.9) allows a server to send back to the client metadata describing the hole and WRITE_HOLE (see Section 14.11) allows the client to punch holes into a file. In addition, SEEK (see Section 14.10) is provided to scan for the next hole or data from a given location.

1.4.4. Space Reservation

When a file is sparse, one concern applications have is ensuring that there will always be enough data blocks available for the file during future writes. A new attribute, space_reserved (see Section 12.2.4) provides the client a guarantee that space will be available.

1.4.5. Application Data Hole (ADH) Support

Some applications treat a file as if it were a disk and as such want to initialize (or format) the file image. We extend both READ_PLUS and introduce WRITE_SAME (see Section 14.12) to understand this metadata as a new form of a hole.

1.4.6. Labeled NFS

While both clients and servers can employ Mandatory Access Control (MAC) security models to enforce data access, there has been no protocol support to allow full interoperability. A new file object attribute, sec_label (see Section 12.2.2) allows for the server to store and enforce MAC labels. The format of the sec_label accommodates any MAC security system.

1.5. Differences from NFSv4.1

In NFSv4.1, the only way to introduce new variants of an operation was to introduce a new operation. I.e., READ becomes either READ2 or READ_PLUS. With the use of discriminated unions as parameters to such functions in NFSv4.2, it is possible to add a new arm in a subsequent minor version. And it is also possible to move such an operation from OPTIONAL/RECOMMENDED to REQUIRED. Forcing an implementation to adopt each arm of a discriminated union at such a time does not meet the spirit of the minor versioning rules. As such, new arms of a discriminated union MUST follow the same guidelines for minor versioning as operations in NFSv4.1 - i.e., they may not be made REQUIRED. To support this, a new error code, NFS4ERR_UNION_NOTSUPP, is introduced which allows the server to communicate to the client that the operation is supported, but the specific arm of the discriminated union is not.

2. Minor Versioning

To address the requirement of an NFS protocol that can evolve as the need arises, the NFSv4 protocol contains the rules and framework to allow for future minor changes or versioning.

The base assumption with respect to minor versioning is that any future accepted minor version will be documented in one or more

Standards Track RFCs. Minor version 0 of the NFSv4 protocol is represented by [I-D.ietf-nfsv4-rfc3530bis], minor version 1 by [RFC5661], and minor version 2 by this document. The COMPOUND and CB_COMPOUND procedures support the encoding of the minor version being requested by the client.

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

1. Procedures are not added or deleted.

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

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

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

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

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

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

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

3. Minor versions must not modify the structure of an existing operation's arguments or results.

   Again, the complexity of handling multiple structure definitions for a single operation is too burdensome. New operations should be added instead of modifying existing structures for a minor version.

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

* adding bits to flag fields, such as new attributes to GETATTR's bitmap4 data type, and providing corresponding variants of opaque arrays, such as a notify4 used together with such bitmaps

* adding bits to existing attributes like ACLs that have flag words

* extending enumerated types (including NFS4ERR_*) with new values

* adding cases to a switched union

4. Note that when adding new cases to a switched union, a minor version must not make new cases be REQUIRED. While the encapsulating operation may be REQUIRED, the new cases (the specific arm of the discriminated union) is not. The error code NFS4ERR_UNION_NOTSUPP is used to notify the client when the server does not support such a case.

5. Minor versions must not modify the structure of existing attributes.

6. Minor versions must not delete operations.

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

7. Minor versions must not delete attributes.

8. Minor versions must not delete flag bits or enumeration values.

9. Minor versions may declare an operation MUST NOT be implemented.

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

   1. Minor versions may declare that an attribute MUST NOT be implemented.

2. Minor versions may declare that a flag bit or enumeration value MUST NOT be implemented.

10. Minor versions may declare an operation to be OBSOLESCENT, which indicates an intention to remove the operation (i.e., make it MANDATORY TO NOT implement) in a subsequent minor version. Such labeling is separate from the question of whether the operation is REQUIRED or RECOMMENDED or OPTIONAL in the current minor version. An operation may be both REQUIRED for the given minor version and marked OBSOLESCENT, with the expectation that it will be MANDATORY TO NOT implement in the next (or other subsequent) minor version.

11. Note that the early notification of operation obsolescence is put in place to mitigate the effects of design and implementation mistakes, and to allow protocol development to adapt to unexpected changes in the pace of implementation. Even if an operation is marked OBSOLESCENT in a given minor version, it may end up not being marked MANDATORY TO NOT implement in the next minor version. In unusual circumstances, it might not be marked OBSOLESCENT in a subsequent minor version, and never become MANDATORY TO NOT implement.

12. Minor versions may downgrade features from REQUIRED to RECOMMENDED, from RECOMMENDED to OPTIONAL, or from OPTIONAL to MANDATORY TO NOT implement. Also, if a feature was marked as OBSOLESCENT in the prior minor version, it may be downgraded from REQUIRED to OPTIONAL from RECOMMENDED to MANDATORY TO NOT implement, or from REQUIRED to MANDATORY TO NOT implement.

13. Minor versions may upgrade features from OPTIONAL to RECOMMENDED, or RECOMMENDED to REQUIRED. Also, if a feature was marked as OBSOLESCENT in the prior minor version, it may be upgraded to not be OBSOLESCENT.

14. A client and server that support minor version X SHOULD support minor versions 0 through X-1 as well.

15. Except for infrastructural changes, a minor version must not introduce REQUIRED new features.

    This rule allows for the introduction of new functionality and forces the use of implementation experience before designating a feature as REQUIRED. On the other hand, some classes of features are infrastructural and have broad effects. Allowing infrastructural features to be RECOMMENDED or OPTIONAL complicates implementation of the minor version.

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

3. Server-side Copy

3.1. Introduction

The server-side copy feature provides a mechanism for the NFS client to perform a file copy on a server or between two servers without the data being transmitted back and forth over the network through the NFS client. Without this feature, an NFS client copies data from one location to another by reading the data from the source server over the network, and then writing the data back over the network to the destiniation server.

If the source object and destination object are on different file servers, the file servers will communicate with one another to perform the copy operation. The server-to-server protocol by which this is accomplished is not defined in this document.

3.2. Protocol Overview

The server-side copy offload operations support both intra-server and inter-server file copies. An intra-server copy is a copy in which the source file and destination file reside on the same server. In an inter-server copy, the source file and destination file are on different servers. In both cases, the copy may be performed synchronously or asynchronously.

Throughout the rest of this document, we refer to the NFS server containing the source file as the "source server" and the NFS server to which the file is transferred as the "destination server". In the case of an intra-server copy, the source server and destination server are the same server. Therefore in the context of an intra-server copy, the terms source server and destination server refer to the single server performing the copy.

The operations described below are designed to copy files. Other file system objects can be copied by building on these operations or using other techniques. For example if the user wishes to copy a directory, the client can synthesize a directory copy by first creating the destination directory and then copying the source directory's files to the new destination directory. If the user wishes to copy a namespace junction [FEDFS-NSDB] [FEDFS-ADMIN], the client can use the ONC RPC Federated Filesystem protocol [FEDFS-ADMIN] to perform the copy. Specifically the client can

determine the source junction's attributes using the FEDFS_LOOKUP_FSN procedure and create a duplicate junction using the FEDFS_CREATE_JUNCTION procedure.

For the inter-server copy, the operations are defined to be compatible with the traditional copy authentication approach. The client and user are authorized at the source for reading. Then they are authorized at the destination for writing.

3.2.1. Overview of Copy Operations

COPY_NOTIFY: For inter-server copies, the client sends this operation to the source server to notify it of a future file copy from a given destination server for the given user. (Section 14.3)

OFFLOAD_REVOKE: Also for inter-server copies, the client sends this operation to the source server to revoke permission to copy a file for the given user. (Section 14.4)

COPY: Used by the client to request a file copy. (Section 14.1)

OFFLOAD_ABORT: Used by the client to abort an asynchronous file copy. (Section 14.2)

OFFLOAD_STATUS: Used by the client to poll the status of an asynchronous file copy. (Section 14.5)

CB_OFFLOAD: Used by the destination server to report the results of an asynchronous file copy to the client. (Section 15.1)

3.2.2. Locking the Files

Both the source and destination file may need to be locked to protect the content during the copy operations. A client can achieve this by a combination of OPEN and LOCK operations. I.e., either share or byte range locks might be desired.

3.2.3. Intra-Server Copy

To copy a file on a single server, the client uses a COPY operation. The server may respond to the copy operation with the final results of the copy or it may perform the copy asynchronously and deliver the results using a CB_OFFLOAD operation callback. If the copy is performed asynchronously, the client may poll the status of the copy using OFFLOAD_STATUS or cancel the copy using OFFLOAD_ABORT.

A synchronous intra-server copy is shown in Figure 1. In this example, the NFS server chooses to perform the copy synchronously. The copy operation is completed, either successfully or unsuccessfully, before the server replies to the client's request. The server's reply contains the final result of the operation.

```
  Client                                  Server
     +                                      +
     |                                      |
     |--- OPEN ---------------------------->| Client opens
     |<------------------------------------/| the source file
     |                                      |
     |--- OPEN ---------------------------->| Client opens
     |<------------------------------------/| the destination file
     |                                      |
     |--- COPY ---------------------------->| Client requests
     |<------------------------------------/| a file copy
     |                                      |
     |--- CLOSE --------------------------->| Client closes
     |<------------------------------------/| the destination file
     |                                      |
     |--- CLOSE --------------------------->| Client closes
     |<------------------------------------/| the source file
     |                                      |
     |                                      |
```

Figure 1: A synchronous intra-server copy.

An asynchronous intra-server copy is shown in Figure 2. In this example, the NFS server performs the copy asynchronously. The server's reply to the copy request indicates that the copy operation was initiated and the final result will be delivered at a later time. The server's reply also contains a copy stateid. The client may use this copy stateid to poll for status information (as shown) or to cancel the copy using a OFFLOAD_ABORT. When the server completes the copy, the server performs a callback to the client and reports the results.

```
     Client                                  Server
        +                                      +
        |                                      |
        |--- OPEN ---------------------------->| Client opens
        |<------------------------------------/| the source file
        |                                      |
        |--- OPEN ---------------------------->| Client opens
        |<------------------------------------/| the destination file
        |                                      |
        |--- COPY ---------------------------->| Client requests
        |<------------------------------------/| a file copy
        |                                      |
        |                                      |
        |--- OFFLOAD_STATUS ------------------>| Client may poll
        |<------------------------------------/| for status
        |                                      |
        |                  .                   | Multiple OFFLOAD_STATUS
        |                  .                   | operations may be sent.
        |                  .                   |
        |                                      |
        |<-- CB_OFFLOAD -----------------------| Server reports results
        |\------------------------------------>|
        |                                      |
        |--- CLOSE --------------------------->| Client closes
        |<------------------------------------/| the destination file
        |                                      |
        |--- CLOSE --------------------------->| Client closes
        |<------------------------------------/| the source file
        |                                      |
        |                                      |
```

Figure 2: An asynchronous intra-server copy.

3.2.4. Inter-Server Copy

A copy may also be performed between two servers. The copy protocol is designed to accommodate a variety of network topologies. As shown in Figure 3, the client and servers may be connected by multiple networks. In particular, the servers may be connected by a specialized, high speed network (network 192.0.2.0/24 in the diagram) that does not include the client. The protocol allows the client to setup the copy between the servers (over network 203.0.113.0/24 in the diagram) and for the servers to communicate on the high speed network if they choose to do so.

```
                      192.0.2.0/24
               +-------------------------------------+
               |                                     |
               |                                     |
               | 192.0.2.18                          | 192.0.2.56
       +-------+------+                       +------+------+
       |     Source   |                       | Destination |
       +-------+------+                       +------+------+
               | 203.0.113.18                        | 203.0.113.56
               |                                     |
               |                                     |
               |             203.0.113.0/24          |
               +------------------+------------------+
                                  |
                                  |
                                  | 203.0.113.243
                            +-----+-----+
                            |   Client  |
                            +-----------+
```

Figure 3: An example inter-server network topology.

For an inter-server copy, the client notifies the source server that a file will be copied by the destination server using a COPY_NOTIFY operation. The client then initiates the copy by sending the COPY operation to the destination server. The destination server may perform the copy synchronously or asynchronously.

A synchronous inter-server copy is shown in Figure 4. In this case, the destination server chooses to perform the copy before responding to the client's COPY request.

An asynchronous copy is shown in Figure 5. In this case, the destination server chooses to respond to the client's COPY request immediately and then perform the copy asynchronously.

```
     Client              Source         Destination
        +                    +                 +
        |                    |                 |
        |--- OPEN        --->|                 | Returns os1
        |<------------------/|                 |
        |                    |                 |
        |--- COPY_NOTIFY --->|                 |
        |<------------------/|                 |
        |                    |                 |
        |--- OPEN ---------------------------->| Returns os2
        |<------------------------------------/|
        |                    |                 |
        |--- COPY ---------------------------->|
        |                    |                 |
        |                    |                 |
        |                    |<----- read -----|
        |                    |\--------------->|
        |                    |                 |
        |                    |        .        | Multiple reads may
        |                    |        .        | be necessary
        |                    |        .        |
        |                    |                 |
        |                    |                 |
        |<------------------------------------/| Destination replies
        |                    |                 | to COPY
        |                    |                 |
        |--- CLOSE --------------------------->| Release open state
        |<------------------------------------/|
        |                    |                 |
        |--- CLOSE       --->|                 | Release open state
        |<------------------/|                 |
```

Figure 4: A synchronous inter-server copy.

```
     Client              Source         Destination
       +                    +                 +
       |                    |                 |
       |--- OPEN        --->|                 | Returns os1
       |<------------------/|                 |
       |                    |                 |
       |--- LOCK        --->|                 | Optional, could be done
       |<------------------/|                 | with a share lock
       |                    |                 |
       |--- COPY_NOTIFY --->|                 | Need to pass in
       |<------------------/|                 | os1 or lock state
       |                    |                 |
       |                    |                 |
       |                    |                 |
```

```
      |--- OPEN ---------------------------->| Returns os2
      |<------------------------------------/|
      |                     |                |
      |--- LOCK ---------------------------->| Optional ...
      |<------------------------------------/|
      |                     |                |
      |--- COPY ---------------------------->| Need to pass in
      |<------------------------------------/| os2 or lock state
      |                     |                |
      |                     |                |
      |                     |<----- read -----|
      |                     |\--------------->|
      |                     |                |
      |                     |        .       | Multiple reads may
      |                     |        .       | be necessary
      |                     |        .       |
      |                     |                |
      |                     |                |
      |--- OFFLOAD_STATUS ------------------>| Client may poll
      |<------------------------------------/| for status
      |                     |                |
      |                     |        .       | Multiple OFFLOAD_STATUS
      |                     |        .       | operations may be sent
      |                     |        .       |
      |                     |                |
      |                     |                |
      |                     |                |
      |<-- CB_OFFLOAD -----------------------| Destination reports
      |\------------------------------------>| results
      |                     |                |
      |--- LOCKU --------------------------->| Only if LOCK was done
      |<------------------------------------/|
      |                     |                |
      |--- CLOSE --------------------------->| Release open state
      |<------------------------------------/|
      |                     |                |
      |--- LOCKU       --->|                 | Only if LOCK was done
      |<-----------------/|                  |
      |                     |                |
      |--- CLOSE       --->|                 | Release open state
      |<-----------------/|                  |
      |                     |                |
```

Figure 5: An asynchronous inter-server copy.

3.2.5. Server-to-Server Copy Protocol

The source server and destination server are not required to use a specific protocol to transfer the file data. The choice of what protocol to use is ultimately the destination server's decision.

3.2.5.1. Using NFSv4.x as a Server-to-Server Copy Protocol

The destination server MAY use standard NFSv4.x (where x >= 1) operations to read the data from the source server. If NFSv4.x is used for the server-to-server copy protocol, the destination server can use the source filehandle and ca_src_stateid provided in the COPY request with standard NFSv4.x operations to read data from the source server.

3.2.5.2. Using an alternative Server-to-Server Copy Protocol

In a homogeneous environment, the source and destination servers might be able to perform the file copy extremely efficiently using specialized protocols. For example the source and destination servers might be two nodes sharing a common file system format for the source and destination file systems. Thus the source and destination are in an ideal position to efficiently render the image of the source file to the destination file by replicating the file system formats at the block level. Another possibility is that the source and destination might be two nodes sharing a common storage area network, and thus there is no need to copy any data at all, and instead ownership of the file and its contents might simply be re-assigned to the destination. To allow for these possibilities, the destination server is allowed to use a server-to-server copy protocol of its choice.

In a heterogeneous environment, using a protocol other than NFSv4.x (e.g., HTTP [RFC2616] or FTP [RFC0959]) presents some challenges. In particular, the destination server is presented with the challenge of accessing the source file given only an NFSv4.x filehandle.

One option for protocols that identify source files with path names is to use an ASCII hexadecimal representation of the source filehandle as the file name.

Another option for the source server is to use URLs to direct the destination server to a specialized service. For example, the response to COPY_NOTIFY could include the URL ftp://s1.example.com:9999/_FH/0x12345, where 0x12345 is the ASCII hexadecimal representation of the source filehandle. When the destination server receives the source server's URL, it would use "_FH/0x12345" as the file name to pass to the FTP server listening on

port 9999 of s1.example.com. On port 9999 there would be a special instance of the FTP service that understands how to convert NFS filehandles to an open file descriptor (in many operating systems, this would require a new system call, one which is the inverse of the makefh() function that the pre-NFSv4 MOUNT service needs).

Authenticating and identifying the destination server to the source server is also a challenge. Recommendations for how to accomplish this are given in Section 3.4.1.4.

3.3. Requirements for Operations

The implementation of server-side copy is OPTIONAL by the client and the server. However, in order to successfully copy a file, some operations MUST be supported by the client and/or server.

If a client desires an intra-server file copy, then it MUST support the COPY and CB_OFFLOAD operations. If COPY returns a stateid, then the client MAY use the OFFLOAD_ABORT and OFFLOAD_STATUS operations.

If a client desires an inter-server file copy, then it MUST support the COPY, COPY_NOTICE, and CB_OFFLOAD operations, and MAY use the OFFLOAD_REVOKE operation. If COPY returns a stateid, then the client MAY use the OFFLOAD_ABORT and OFFLOAD_STATUS operations.

If a server supports intra-server copy, then the server MUST support the COPY operation. If a server's COPY operation returns a stateid, then the server MUST also support these operations: CB_OFFLOAD, OFFLOAD_ABORT, and OFFLOAD_STATUS.

If a source server supports inter-server copy, then the source server MUST support all these operations: COPY_NOTIFY and OFFLOAD_REVOKE. If a destination server supports inter-server copy, then the destination server MUST support the COPY operation. If a destination server's COPY operation returns a stateid, then the destination server MUST also support these operations: CB_OFFLOAD, OFFLOAD_ABORT, COPY_NOTIFY, OFFLOAD_REVOKE, and OFFLOAD_STATUS.

Each operation is performed in the context of the user identified by the ONC RPC credential of its containing COMPOUND or CB_COMPOUND request. For example, a OFFLOAD_ABORT operation issued by a given user indicates that a specified COPY operation initiated by the same user be canceled. Therefore a OFFLOAD_ABORT MUST NOT interfere with a copy of the same file initiated by another user.

An NFS server MAY allow an administrative user to monitor or cancel copy operations using an implementation specific interface.

3.3.1. netloc4 - Network Locations

The server-side copy operations specify network locations using the netloc4 data type shown below:

```
enum netloc_type4 {
        NL4_NAME        = 0,
        NL4_URL         = 1,
        NL4_NETADDR     = 2
};
union netloc4 switch (netloc_type4 nl_type) {
        case NL4_NAME:          utf8str_cis nl_name;
        case NL4_URL:           utf8str_cis nl_url;
        case NL4_NETADDR:       netaddr4    nl_addr;
};
```

If the netloc4 is of type NL4_NAME, the nl_name field MUST be specified as a UTF-8 string. The nl_name is expected to be resolved to a network address via DNS, LDAP, NIS, /etc/hosts, or some other means. If the netloc4 is of type NL4_URL, a server URL [RFC3986] appropriate for the server-to-server copy operation is specified as a UTF-8 string. If the netloc4 is of type NL4_NETADDR, the nl_addr field MUST contain a valid netaddr4 as defined in Section 3.3.9 of [RFC5661].

When netloc4 values are used for an inter-server copy as shown in Figure 3, their values may be evaluated on the source server, destination server, and client. The network environment in which these systems operate should be configured so that the netloc4 values are interpreted as intended on each system.

3.3.2. Copy Offload Stateids

A server may perform a copy offload operation asynchronously. An asynchronous copy is tracked using a copy offload stateid. Copy offload stateids are included in the COPY, OFFLOAD_ABORT, OFFLOAD_STATUS, and CB_OFFLOAD operations.

Section 8.2.4 of [RFC5661] specifies that stateids are valid until either (A) the client or server restart or (B) the client returns the resource.

A copy offload stateid will be valid until either (A) the client or server restarts or (B) the client returns the resource by issuing a OFFLOAD_ABORT operation or the client replies to a CB_OFFLOAD operation.

A copy offload stateid's seqid MUST NOT be 0. In the context of a copy offload operation, it is ambiguous to indicate the most recent copy offload operation using a stateid with seqid of 0. Therefore a copy offload stateid with seqid of 0 MUST be considered invalid.

3.4. Security Considerations

The security considerations pertaining to NFSv4 [I-D.ietf-nfsv4-rfc3530bis] apply to this chapter.

The standard security mechanisms provide by NFSv4 [I-D.ietf-nfsv4-rfc3530bis] may be used to secure the protocol described in this chapter.

NFSv4 clients and servers supporting the inter-server copy operations described in this chapter are REQUIRED to implement the mechanism described in Section 3.4.1.2, and to support rejecting COPY_NOTIFY requests that do not use RPCSEC_GSS with privacy. If the server-to-server copy protocol is ONC RPC based, the servers are also REQUIRED to implement [rpcsec_gssv3] including the RPCSEC_GSSv3 copy_to_auth, copy_from_auth, and copy_confirm_auth structured privileges. This requirement to implement is not a requirement to use; for example, a server may depending on configuration also allow COPY_NOTIFY requests that use only AUTH_SYS.

3.4.1. Inter-Server Copy Security

3.4.1.1. Requirements for Secure Inter-Server Copy

Inter-server copy is driven by several requirements:

o The specification must not mandate an inter-server copy protocol. There are many ways to copy data. Some will be more optimal than others depending on the identities of the source server and destination server. For example the source and destination servers might be two nodes sharing a common file system format for the source and destination file systems. Thus the source and destination are in an ideal position to efficiently render the image of the source file to the destination file by replicating the file system formats at the block level. In other cases, the source and destination might be two nodes sharing a common storage area network, and thus there is no need to copy any data at all, and instead ownership of the file and its contents simply gets re-assigned to the destination.

o The specification must provide guidance for using NFSv4.x as a copy protocol. For those source and destination servers willing

to use NFSv4.x there are specific security considerations that this specification can and does address.

- The specification must not mandate pre-configuration between the source and destination server. Requiring that the source and destination first have a "copying relationship" increases the administrative burden. However the specification MUST NOT preclude implementations that require pre-configuration.

- The specification must not mandate a trust relationship between the source and destination server. The NFSv4 security model requires mutual authentication between a principal on an NFS client and a principal on an NFS server. This model MUST continue with the introduction of COPY.

3.4.1.2. Inter-Server Copy via ONC RPC with RPCSEC_GSSv3

When the client sends a COPY_NOTIFY to the source server to expect the destination to attempt to copy data from the source server, it is expected that this copy is being done on behalf of the principal (called the "user principal") that sent the RPC request that encloses the COMPOUND procedure that contains the COPY_NOTIFY operation. The user principal is identified by the RPC credentials. A mechanism that allows the user principal to authorize the destination server to perform the copy, that lets the source server properly authenticate the destination's copy, and does not allow the destination server to exceed this authorization, is necessary.

An approach that sends delegated credentials of the client's user principal to the destination server is not used for the following reason. If the client's user delegated its credentials, the destination would authenticate as the user principal. If the destination were using the NFSv4 protocol to perform the copy, then the source server would authenticate the destination server as the user principal, and the file copy would securely proceed. However, this approach would allow the destination server to copy other files. The user principal would have to trust the destination server to not do so. This is counter to the requirements, and therefore is not considered.

Instead, we employ a combination of two features of the RPCSEC_GSSv3 [rpcsec_gssv3] protocol: compound authentication and RPC application defined structured privilege assertions. The combination of these features allows the destination server to authenticate to the source server as acting on behalf of the user prinicpal, and to authorize the destination server to perform READs of the file to be copied from the source on behalf of the user principal. Once the copy is

complete, the client can destroy the RPCSEC_GSSv3 handles to end the source and destination servers authorization to copy.

RPCSEC_GSSv3 introduces the notion of RPC application defined structured privileges. We define three structured privileges that work in tandum to authorize the copy:

copy_from_auth: A user principal is authorizing a source principal ("nfs@<source>") to allow a destination principal ("nfs@<destination>") to setup the copy_confirm_auth privilege required to copy a file from the source to the destination on behalf of the user principal. This privilege is established on the source server before the user principal sends a COPY_NOTIFY operation to the source server, and the resultant RPCSEC_GSSv3 context is used to secure the COPY_NOTIFY operation.

```
struct copy_from_auth_priv {
        secret4             cfap_shared_secret;
        netloc4             cfap_destination;
        /* the NFSv4 user name that the user principal maps to */
        utf8str_mixed       cfap_username;
};
```

cfp_shared_secret is an automatically generated random number secret value.

copy_to_auth: A user principal is authorizing a destination principal ("nfs@<destination>") to setup a copy_confirm_auth privilege with a source principal ("nfs@<source>") to allow it to copy a file from the source to the destination on behalf of the user principal. This privilege is established on the destination server before the user principal sends a COPY operation to the destination server, and the resultant RPCSEC_GSSv3 context is used to secure the COPY operation.

```
struct copy_to_auth_priv {
        /* equal to cfap_shared_secret */
        secret4              ctap_shared_secret;
        netloc4              ctap_source;
        /* the NFSv4 user name that the user principal maps to */
        utf8str_mixed        ctap_username;
        /*
         * user principal RPCSEC_GSSv1 (or v2) handle shared
         * with the source server
         */
        opaque               ctap_handle;
        int                  ctap_handle_vers;
        /* A nounce and a mic of the nounce using ctap_handle */
        opaque               ctap_nounce;
        opaque               ctap_nounce_mic;
};
```

ctap_shared_secret is the automatically generated secret value used to establish the copy_from_auth privilege with the source principal. ctap_handle, ctap_handle_vers, ctap_nounce and ctap_nounce_mic are used to construct the compound authentication portion of the copy_confirm_auth RPCGSS_GSSv3 context between the destination server and the source server. See Section 3.4.1.2.1

copy_confirm_auth: A destination principal ("nfs@<destination>") is confirming with the source principal ("nfs@<source>") that it is authorized to copy data from the source. Note that besides the rpc_gss3_privs payload (struct copy_confirm_auth_priv), the copy_confirm_auth RPCSEC_GSS3_CREATE message also contains an rpc_gss3_gss_binding payload so that the copy is done on behalf of the user principal. This privilege is established on the destination server before the file is copied from the source to the destination. The resultant RPCSEC_GSSv3 context is used to secure the READ operations from the source to the destination server.

```
struct copy_confirm_auth_priv {
        /* equal to GSS_GetMIC() of cfap_shared_secret */
        opaque              ccap_shared_secret_mic<>;
        /* the NFSv4 user name that the user principal maps to */
        utf8str_mixed       ccap_username;
};
```

3.4.1.2.1. Establishing a Security Context

The RPCSEC_GSSv3 compound authentication feature allows a server to act on behalf of a user if the server identifies the user and trusts the client. In the inter-server server side copy case, the server is the source server, and the client is the destination server acting as a client when performing the copy.

The user principal is not required (nor expected) to have an RPCSEC_GSS secured connection and context between the destination server (acting as a client) and the source server. The user principal does have an RPCSEC_GSS secured connection and context between the client and the source server established for the OPEN of the file to be copied.

We use the RPCSEC_GSS context established between the user principal and the source server to OPEN the file to be copied to provide the the necessary user principal identification to the source server from the destination server (acting as a client). This is accomplished by sending the user principal indentification information: e.g the rpc_gss3_gss_binding fields, in the copy_to_auth privilege established between the client and the destination server. This same information is then placed in the rpc_gss3_gss_binding fields of the copy_confirm_auth RPCSEC_GSS3_CREATE message sent from the destination server (acting as a client) to the source server.

When the user principal wants to COPY a file between two servers, if it has not established copy_from_auth and copy_to_auth privileges on the servers, it establishes them:

- o As noted in [rpcsec_gssv3] the client uses an existing RPCSEC_GSSv1 (or v2) context termed the "parent" handle to establish and protect RPCSEC_GSSv3 exchanges. The copy_from_auth privilege will use the context established between the user principal and the source server used to OPEN the source file as the RPCSEC_GSSv3 parent handle. The copy_to_auth privilege will use the context established between the user principal and the destination server used to OPEN the destination file as the RPCSEC_GSSv3 parent handle.

- o A random number is generated to use as a secret to be shared between the two servers. This shared secret will be placed in the cfap_shared_secret and ctap_shared_secret fields of the appropriate privilege data types, copy_from_auth_priv and copy_to_auth_priv. Because of this shared_secret the RPCSEC_GSS3_CREATE control messages for copy_from_auth and copy_to_auth MUST use a QOP of rpc_gss_svc_privacy.

o An instance of copy_from_auth_priv is filled in with the shared secret, the destination server, and the NFSv4 user id of the user principal and is placed in rpc_gss3_create_args assertions[0].assertion.privs.privilege. The string "copy_from_auth" is placed in assertions[0].assertion.privs.name. The field assertions[0].critical is set to TRUE. The source server unwraps the rpc_gss_svc_privacy RPCSEC_GSS3_CREATE payload and verifies that the NFSv4 user id being asserted matches the source server's mapping of the user principal. If it does, the privilege is established on the source server as: <"copy_from_auth", user id, destination>. The field "handle" in a successful reply is the RPCSEC_GSSv3 "child" handle that the client will use on COPY_NOTIFY requests to the source server involving the destination server. granted_assertions[0].assertion.privs.name will be equal to "copy_from_auth".

o An instance of copy_to_auth_priv is filled in with the shared secret, the cnr_source_server list returned by COPY_NOTIFY, and the NFSv4 user id of the user principal. The next four fields are passed in the copy_to_auth privilege to be used by the copy_confirm_auth rpc_gss3_gss_binding fields as explained above. A nounce is created, and GSS_MIC() is invoked on the nounce using the RPCSEC_GSSv1 (or v2) context shared between user prinicpal and the source server. The nounce, nounce MIC, context handle used to create the nounce MIC, and the context handle version are added to the copy_to_auth_priv instance which is placed in rpc_gss3_create_args assertions[0].assertion.privs.privilege. The string "copy_to_auth" is placed in assertions[0].assertion.privs.name. The field assertions[0].critical is set to TRUE. The destination server unwraps the rpc_gss_svc_privacy RPCSEC_GSS3_CREATE payload and verifies that the NFSv4 user id being asserted matches the destination server's mapping of the user principal. If it does, the privilege is established on the destination server as: <"copy_to_auth", user id, source list, nounce, nounce MIC, context handle, handle version>. The field "handle" in a successful reply is the RPCSEC_GSSv3 "child" handle that the client will use on COPY requests to the destination server involving the source server. granted_assertions[0].assertion.privs.name will be equal to "copy_to_auth".

As noted in [rpcsec_gssv3] section 2.3.1 "Create Request", both the client and the source server should associate the RPCSEC_GSSv3 "child" handle with the parent RPCSEC_GSSv1 (or v2) handle used to create the RPCSEC_GSSv3 child handle.

3.4.1.2.2. Starting a Secure Inter-Server Copy

When the client sends a COPY_NOTIFY request to the source server, it uses the privileged "copy_from_auth" RPCSEC_GSSv3 handle. cna_destination_server in COPY_NOTIFY MUST be the same as cfap_destination specified in copy_from_auth_priv. Otherwise, COPY_NOTIFY will fail with NFS4ERR_ACCESS. The source server verifies that the privilege <"copy_from_auth", user id, destination> exists, and annotates it with the source filehandle, if the user principal has read access to the source file, and if administrative policies give the user principal and the NFS client read access to the source file (i.e., if the ACCESS operation would grant read access). Otherwise, COPY_NOTIFY will fail with NFS4ERR_ACCESS.

When the client sends a COPY request to the destination server, it uses the privileged "copy_to_auth" RPCSEC_GSSv3 handle. ca_source_server list in COPY MUST be the same as ctap_source list specified in copy_to_auth_priv. Otherwise, COPY will fail with NFS4ERR_ACCESS. The destination server verifies that the privilege <"copy_to_auth", user id, source list, nounce, nounce MIC, context handle, handle version> exists, and annotates it with the source and destination filehandles. If the COPY returns a wr_callback_id, then this is an asynchronous copy and the wr_callback_id must also must be annotated to the copy_to_auth privilege. If the client has failed to establish the "copy_to_auth" privilege it will reject the request with NFS4ERR_PARTNER_NO_AUTH.

If either the COPY_NOTIFY, or the COPY operations fail, the associated "copy_from_auth" and "copy_to_auth" RPCSEC_GSSv3 handles MUST be destroyed.

3.4.1.2.3. Securing ONC RPC Server-to-Server Copy Protocols

After a destination server has a "copy_to_auth" privilege established on it, and it receives a COPY request, if it knows it will use an ONC RPC protocol to copy data, it will establish a "copy_confirm_auth" privilege on the source server prior to responding to the COPY operation as follows:

o Before establishing an RPCSEC_GSSv3 context, a parent context needs to exist between nfs@<destination> as the initiator principal, and nfs@<source> as the target principal. If NFS is to be used as the copy protocol, this means that the destiniation server must mount the source server using RPCSEC_GSS.

o An instance of copy_confirm_auth_priv is filled in with information from the established "copy_to_auth" privilege. The value of the field ccap_shared_secret_mic is a GSS_GetMIC() of the

ctap_shared_secret in the copy_to_auth privilege using the parent handle context. The field ccap_username is the mapping of the user principal to an NFSv4 user name ("user"@"domain" form), and MUST be the same as the ctap_username in the copy_to_auth privilege. The copy_confirm_auth_priv instance is placed in rpc_gss3_create_args assertions[0].assertion.privs.privilege. The string "copy_confirm_auth" is placed in assertions[0].assertion.privs.name. The field assertions[0].critical is set to TRUE.

- o The copy_confirm_auth RPCSEC_GSS3_CREATE call also includes a compound authentication component. The rpc_gss3_gss_binding fields are filled in with information from the estalished "copy_to_auth" privilege (see Section 3.4.1.2.1). The ctap_handle_vers, ctap_handle, ctap_nounce, and ctap_nounce_mic are assigned to the vers, handle, nounce, and mic fields of an rpc_gss3_gss_binding instance respectively.

- o The RPCSEC_GSS3_CREATE copy_from_auth message is sent to the source server with a QOP of rpc_gss_svc_privacy. The source server unwraps the rpc_gss_svc_privacy RPCSEC_GSS3_CREATE payload and verifies the cap_shared_secret_mic by calling GSS_VerifyMIC() using the parent context on the cfap_shared_secret from the established "copy_from_auth" privilege, and verifies the that the ccap_username equals the cfap_username. The source server then locates the ctap_handle in it's GSS context cache and verifies that the handle belongs to the user principal that maps to the ccap_username and that the cached handle version equals ctap_handle_vers. The ctap_nounce_mic is verified by calling GSS_VerifyMIC() on the ctap_nounce using the cached handle context. If all verification succeeds, the "copy_confirm_auth" privilege is established on the source server as < "copy_confirm_auth", shared_secret_mic, user id, nounce, nounce MIC, context handle, context handle version>, and the resultant child handle is noted to be acting on behalf of the user principal. If the source server fails to verify either the privilege or the compound_binding, the COPY operation will be rejected with NFS4ERR_PARTNER_NO_AUTH.

- o All subsequent ONC RPC requests sent from the destination to copy data from the source to the destination will use the RPCSEC_GSSv3 handle returned by the source's RPCSEC_GSS3_CREATE response. Note that as per the Compound Authenticaion section of [rpcsec_gssv3] the resultant RPCSEC_GSSv3 context handle is bound to the user principal RPCSEC_GSS context and so it MUST be treated by servers as authenticating the user principal.

Note that the use of the "copy_confirm_auth" privilege accomplishes the following:

- o If a protocol like NFS is being used, with export policies, export policies can be overridden in case the destination server as-an-NFS-client is not authorized

- o Manual configuration to allow a copy relationship between the source and destination is not needed.

3.4.1.2.4. Finishing or Stoping a Secure Inter-Server Copy

Under normal operation, the client MUST destroy the copy_from_auth and the copy_to_auth RPCSEC_GSSv3 handle once the COPY operation returns for a synchronous inter-server copy or a CB_OFFLOAD reports the result of an asynchronous copy.

The copy_confirm_auth privilege and compound authentication RPCSEC_GSSv3 handle is constructed from information held by the copy_to_auth privilege, and MUST be destroyed by the destination server (via an RPCSEC_GSS3_DESTROY call) when the copy_to_auth RPCSEC_GSSv3 handle is destroyed.

If the client sends a OFFLOAD_REVOKE to the source server to rescind the destination server's synchronous copy privilege, it uses the privileged "copy_from_auth" RPCSEC_GSSv3 handle and the cra_destination_server in OFFLOAD_REVOKE MUST be the same as the name of the destination server specified in copy_from_auth_priv. The source server will then delete the <"copy_from_auth", user id, destination> privilege and fail any subsequent copy requests sent under the auspices of this privilege from the destination server. The client MUST destroy both the "copy_from_auth" and the "copy_to_auth" RPCSEC_GSSv3 handles.

If the client sends a OFFLOAD_STATUS to the destination server to check on the status of an asynchronous copy, it uses the privileged "copy_to_auth" RPCSEC_GSSv3 handle and the osa_stateid in OFFLOAD_STATUS MUST be the same as the wr_callback_id specified in the "copy_to_auth" privilege stored on the destiniation server.

If the client sends a OFFLOAD_ABORT to the destination server to cancel an asynchronous copy, it uses the privileged "copy_to_auth" RPCSEC_GSSv3 handle and the oaa_stateid in OFFLOAD_ABORT MUST be the same as the wr_callback_id specified in the "copy_to_auth" privilege stored on the destiniation server. The destiniation server will then delete the <"copy_to_auth", user id, source list, nounce, nounce MIC, context handle, handle version> privilege and the associated

"copy_confirm_auth" RPCSEC_GSSv3 handle. The client MUST destroy both the copy_to_auth and copy_from_auth RPCSEC_GSSv3 handles.

3.4.1.3. Inter-Server Copy via ONC RPC without RPCSEC_GSS

ONC RPC security flavors other than RPCSEC_GSS MAY be used with the server-side copy offload operations described in this chapter. In particular, host-based ONC RPC security flavors such as AUTH_NONE and AUTH_SYS MAY be used. If a host-based security flavor is used, a minimal level of protection for the server-to-server copy protocol is possible.

In the absence of a strong security mechanism designed for the purpose, the challenge is how the source server and destination server identify themselves to each other, especially in the presence of multi-homed source and destination servers. In a multi-homed environment, the destination server might not contact the source server from the same network address specified by the client in the COPY_NOTIFY. This can be overcome using the procedure described below.

When the client sends the source server the COPY_NOTIFY operation, the source server may reply to the client with a list of target addresses, names, and/or URLs and assign them to the unique quadruple: <random number, source fh, user ID, destination address Y>. If the destination uses one of these target netlocs to contact the source server, the source server will be able to uniquely identify the destination server, even if the destination server does not connect from the address specified by the client in COPY_NOTIFY. The level of assurance in this identification depends on the unpredictability, strength and secrecy of the random number.

For example, suppose the network topology is as shown in Figure 3. If the source filehandle is 0x12345, the source server may respond to a COPY_NOTIFY for destination 203.0.113.56 with the URLs:

```
nfs://203.0.113.18//_COPY/FvhH1OKbu8VrxvV1erdjvR7N/203.0.113.56/
_FH/0x12345

nfs://192.0.2.18//_COPY/FvhH1OKbu8VrxvV1erdjvR7N/203.0.113.56/_FH/
0x12345
```

The name component after _COPY is 24 characters of base 64, more than enough to encode a 128 bit random number.

The client will then send these URLs to the destination server in the COPY operation. Suppose that the 192.0.2.0/24 network is a high speed network and the destination server decides to transfer the file

over this network. If the destination contacts the source server from 192.0.2.56 over this network using NFSv4.1, it does the following:

```
COMPOUND  { PUTROOTFH, LOOKUP "_COPY" ; LOOKUP
   "FvhH1OKbu8VrxvV1erdjvR7N" ; LOOKUP "203.0.113.56"; LOOKUP "_FH" ;
   OPEN "0x12345" ; GETFH }
```

Provided that the random number is unpredictable and has been kept secret by the parties involved, the source server will therefore know that these NFSv4.x operations are being issued by the destination server identified in the COPY_NOTIFY. This random number technique only provides initial authentication of the destination server, and cannot defend against man-in-the-middle attacks after authentication or an eavesdropper that observes the random number on the wire. Other secure communication techniques (e.g., IPsec) are necessary to block these attacks.

Servers SHOULD reject COPY_NOTIFY requests that do not use RPCSEC_GSS with privacy, thus ensuring the URL in the COPY_NOTIFY reply is encrypted. For the same reason, clients SHOULD send COPY requests to the destination using RPCSEC_GSS with privacy.

3.4.1.4. Inter-Server Copy without ONC RPC

The same techniques as Section 3.4.1.3, using unique URLs for each destination server, can be used for other protocols (e.g., HTTP [RFC2616] and FTP [RFC0959]) as well.

4. Support for Application IO Hints

Applications can issue client I/O hints via posix_fadvise() [posix_fadvise] to the NFS client. While this can help the NFS client optimize I/O and caching for a file, it does not allow the NFS server and its exported file system to do likewise. We add an IO_ADVISE procedure (Section 14.7) to communicate the client file access patterns to the NFS server. The NFS server upon receiving a IO_ADVISE operation MAY choose to alter its I/O and caching behavior, but is under no obligation to do so.

Application specific NFS clients such as those used by hypervisors and databases can also leverage application hints to communicate their specialized requirements.

5. Sparse Files

5.1. Introduction

A sparse file is a common way of representing a large file without having to utilize all of the disk space for it. Consequently, a sparse file uses less physical space than its size indicates. This means the file contains 'holes', byte ranges within the file that contain no data. Most modern file systems support sparse files, including most UNIX file systems and NTFS, but notably not Apple's HFS+. Common examples of sparse files include Virtual Machine (VM) OS/disk images, database files, log files, and even checkpoint recovery files most commonly used by the HPC community.

If an application reads a hole in a sparse file, the file system must return all zeros to the application. For local data access there is little penalty, but with NFS these zeroes must be transferred back to the client. If an application uses the NFS client to read data into memory, this wastes time and bandwidth as the application waits for the zeroes to be transferred.

A sparse file is typically created by initializing the file to be all zeros - nothing is written to the data in the file, instead the hole is recorded in the metadata for the file. So a 8G disk image might be represented initially by a couple hundred bits in the inode and nothing on the disk. If the VM then writes 100M to a file in the middle of the image, there would now be two holes represented in the metadata and 100M in the data.

Three new operations WRITE_HOLE (Section 14.11), WRITE_SAME (Section 14.12), and READ_PLUS (Section 14.9) are introduced. WRITE_HOLE allows for the creation of a sparse file and/or hole punching. I.e, An application might want to zero out a range of the file. WRITE_SAME allows for the creation of application specific block structures in a file which is treated by the application as if it were a disk. READ_PLUS supports all the features of READ but includes an extension to support sparse pattern files (Section 7.1.2). READ_PLUS is guaranteed to perform no worse than READ, and can dramatically improve performance with sparse files. READ_PLUS does not depend on pNFS protocol features, but can be used by pNFS to support sparse files.

5.2. Terminology

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

Sparse file: A Regular file that contains one or more Holes.

Hole: A byte range within a Sparse file that contains regions of all zeroes. For block-based file systems, this could also be an unallocated region of the file.

Hole Threshold: The minimum length of a Hole as determined by the server. If a server chooses to define a Hole Threshold, then it would not return hole information about holes with a length shorter than the Hole Threshold.

5.3. New Operations

READ_PLUS, WRITE_HOLE, and WRITE_SAME are new variants of the NFSv4.1 READ and WRITE operations [RFC5661]. Besides being able to support all of the data semantics of those operations, they can also be used by the client and server to efficiently transfer both holes and ADHs (see Section 7.1.1). As READ is inefficient for transfer of sparse sections of the file, it is marked as OBSOLESCENT in NFSv4.2. Instead, a client should utilize READ_PLUS. Note that as the client has no a priori knowledge of whether either an ADH or a hole is present or not, if it supports these operations and so does the server, then it should always use these operations.

5.3.1. READ_PLUS

For holes, READ_PLUS extends the response to avoid returning data for portions of the file which are initialized and contain no backing store. Additionally it will do so if the result would appear to be a hole. I.e., if the result was a data block composed entirely of zeros, then it is easier to return a hole. Returning data blocks of uninitialized data wastes computational and network resources, thus reducing performance. For ADHs, READ_PLUS is used to return the metadata describing the portions of the file which are initialized and contain no backing store.

If the client sends a READ operation, it is explicitly stating that it is neither supporting sparse files nor ADHs. So if a READ occurs on a sparse ADH or file, then the server must expand such data to be raw bytes. If a READ occurs in the middle of a hole or ADH, the server can only send back bytes starting from that offset. In contrast, if a READ_PLUS occurs in the middle of a hole or ADH, the server can send back a range which starts before the offset and extends past the range.

5.3.2. WRITE_HOLE and WRITE_SAME

WRITE_HOLE can be used to hole punch and WRITE_SAME can be used to initialize ADHs. For either purpose, the client can avoid the transfer of a repetitive pattern across the network. If the

filesystem on the server does not support sparse files, the WRITE_HOLE and WRITE_SAME operations may return the result asynchronously via the CB_OFFLOAD operation. As a hole punch may entail deallocating data blocks, even if the filesystem supports sparse files, it may still have to return the result via CB_OFFLOAD.

6. Space Reservation

6.1. Introduction

Applications such as hypervisors want to be able to reserve space for a file, report the amount of actual disk space a file occupies, and free-up the backing space of a file when it is not required. In virtualized environments, virtual disk files are often stored on NFS mounted volumes. Since virtual disk files represent the hard disks of virtual machines, hypervisors often have to guarantee certain properties for the file.

One such example is space reservation. When a hypervisor creates a virtual disk file, it often tries to preallocate the space for the file so that there are no future allocation related errors during the operation of the virtual machine. Such errors prevent a virtual machine from continuing execution and result in downtime.

Currently, in order to achieve such a guarantee, applications zero the entire file. The initial zeroing allocates the backing blocks and all subsequent writes are overwrites of already allocated blocks. This approach is not only inefficient in terms of the amount of I/O done, it is also not guaranteed to work on file systems that are log structured or deduplicated. An efficient way of guaranteeing space reservation would be beneficial to such applications.

We define a "reservation" as being the combination of the space_reserved attribute (see Section 12.2.4) and the size attribute (see Section 5.8.1.5 of [RFC5661]). If space_reserved attribute is set on a file, it is guaranteed that writes that do not grow the file past the size will not fail with NFS4ERR_NOSPC. Once the size is changed, then the reservation is changed to that new size.

Another useful feature is the ability to report the number of blocks that would be freed when a file is deleted. Currently, NFS reports two size attributes:

size The logical file size of the file.

space_used The size in bytes that the file occupies on disk

While these attributes are sufficient for space accounting in traditional file systems, they prove to be inadequate in modern file systems that support block sharing. In such file systems, multiple inodes can point to a single block with a block reference count to guard against premature freeing. Having a way to tell the number of blocks that would be freed if the file was deleted would be useful to applications that wish to migrate files when a volume is low on space.

Since virtual disks represent a hard drive in a virtual machine, a virtual disk can be viewed as a file system within a file. Since not all blocks within a file system are in use, there is an opportunity to reclaim blocks that are no longer in use. A call to deallocate blocks could result in better space efficiency. Lesser space MAY be consumed for backups after block deallocation.

The following operations and attributes can be used to resolve this issues:

space_reserved This attribute specifies that writes to the reserved area of the file will not fail with NFS4ERR_NOSPACE.

space_freed This attribute specifies the space freed when a file is deleted, taking block sharing into consideration.

WRITE_HOLE and WRITE_SAME These operations zero and/or deallocate the blocks backing a region of the file.

If space_used of a file is interpreted to mean the size in bytes of all disk blocks pointed to by the inode of the file, then shared blocks get double counted, over-reporting the space utilization. This also has the adverse effect that the deletion of a file with shared blocks frees up less than space_used bytes.

On the other hand, if space_used is interpreted to mean the size in bytes of those disk blocks unique to the inode of the file, then shared blocks are not counted in any file, resulting in under-reporting of the space utilization.

For example, two files A and B have 10 blocks each. Let 6 of these blocks be shared between them. Thus, the combined space utilized by the two files is 14 * BLOCK_SIZE bytes. In the former case, the combined space utilization of the two files would be reported as 20 * BLOCK_SIZE. However, deleting either would only result in 4 * BLOCK_SIZE being freed. Conversely, the latter interpretation would report that the space utilization is only 8 * BLOCK_SIZE.

Adding another size attribute, space_freed (see Section 12.2.5), is helpful in solving this problem. space_freed is the number of blocks that are allocated to the given file that would be freed on its deletion. In the example, both A and B would report space_freed as 4 * BLOCK_SIZE and space_used as 10 * BLOCK_SIZE. If A is deleted, B will report space_freed as 10 * BLOCK_SIZE as the deletion of B would result in the deallocation of all 10 blocks.

The addition of this problem does not solve the problem of space being over-reported. However, over-reporting is better than under-reporting.

7. Application Data Hole Support

At the OS level, files are contained on disk blocks. Applications are also free to impose structure on the data contained in a file and we can define an Application Data Block (ADB) to be such a structure. From the application's viewpoint, it only wants to handle ADBs and not raw bytes (see [Strohm11]). An ADB is typically comprised of two sections: a header and data. The header describes the characteristics of the block and can provide a means to detect corruption in the data payload. The data section is typically initialized to all zeros.

The format of the header is application specific, but there are two main components typically encountered:

1. A logical block number which allows the application to determine which data block is being referenced. This is useful when the client is not storing the blocks in contiguous memory.

2. Fields to describe the state of the ADB and a means to detect block corruption. For both pieces of data, a useful property is that allowed values be unique in that if passed across the network, corruption due to translation between big and little endian architectures are detectable. For example, 0xF0DEDEF0 has the same bit pattern in both architectures.

Applications already impose structures on files [Strohm11] and detect corruption in data blocks [Ashdown08]. What they are not able to do is efficiently transfer and store ADBs. To initialize a file with ADBs, the client must send the full ADB to the server and that must be stored on the server.

In this section, we are going to define an Application Data Hole (ADH), which is a generic framework for transferring the ADB, present one approach to detecting corruption in a given ADH implementation, and describe the model for how the client and server can support

efficient initialization of ADHs, reading of ADH holes, punching ADH holes in a file, and space reservation. We define the ADHN to be the Application Data Hole Number, which is the logical block number discussed earlier.

7.1. Generic Framework

We want the representation of the ADH to be flexible enough to support many different applications. The most basic approach is no imposition of a block at all, which means we are working with the raw bytes. Such an approach would be useful for storing holes, punching holes, etc. In more complex deployments, a server might be supporting multiple applications, each with their own definition of the ADH. One might store the ADHN at the start of the block and then have a guard pattern to detect corruption [McDougall07]. The next might store the ADHN at an offset of 100 bytes within the block and have no guard pattern at all, i.e., existing applications might already have well defined formats for their data blocks.

The guard pattern can be used to represent the state of the block, to protect against corruption, or both. Again, it needs to be able to be placed anywhere within the ADH.

We need to be able to represent the starting offset of the block and the size of the block. Note that nothing prevents the application from defining different sized blocks in a file.

7.1.1. Data Hole Representation

```
struct app_data_hole4 {
        offset4         adh_offset;
        length4         adh_block_size;
        length4         adh_block_count;
        length4         adh_reloff_blocknum;
        count4          adh_block_num;
        length4         adh_reloff_pattern;
        opaque          adh_pattern<>;
};
```

The app_data_hole4 structure captures the abstraction presented for the ADH. The additional fields present are to allow the transmission of adh_block_count ADHs at one time. We also use adh_block_num to convey the ADHN of the first block in the sequence. Each ADH will contain the same adh_pattern string.

As both adh_block_num and adh_pattern are optional, if either adh_reloff_pattern or adh_reloff_blocknum is set to NFS4_UINT64_MAX, then the corresponding field is not set in any of the ADH.

7.1.2. Data Content

```
/*
 * Use an enum such that we can extend new types.
 */
enum data_content4 {
        NFS4_CONTENT_DATA = 0,
        NFS4_CONTENT_APP_DATA_HOLE = 1,
        NFS4_CONTENT_HOLE = 2
};
```

New operations might need to differentiate between wanting to access data versus an ADH. Also, future minor versions might want to introduce new data formats. This enumeration allows that to occur.

7.2. An Example of Detecting Corruption

In this section, we define an ADH format in which corruption can be detected. Note that this is just one possible format and means to detect corruption.

Consider a very basic implementation of an operating system's disk blocks. A block is either data or it is an indirect block which allows for files to be larger than one block. It is desired to be able to initialize a block. Lastly, to quickly unlink a file, a block can be marked invalid. The contents remain intact - which would enable this OS application to undelete a file.

The application defines 4k sized data blocks, with an 8 byte block counter occurring at offset 0 in the block, and with the guard pattern occurring at offset 8 inside the block. Furthermore, the guard pattern can take one of four states:

0xfeedface - This is the FREE state and indicates that the ADH format has been applied.

0xcafedead - This is the DATA state and indicates that real data has been written to this block.

0xe4e5c001 - This is the INDIRECT state and indicates that the block contains block counter numbers that are chained off of this block.

0xba1ed4a3 - This is the INVALID state and indicates that the block contains data whose contents are garbage.

Finally, it also defines an 8 byte checksum [Baira08] starting at byte 16 which applies to the remaining contents of the block. If the

state is FREE, then that checksum is trivially zero. As such, the application has no need to transfer the checksum implicitly inside the ADH - it need not make the transfer layer aware of the fact that there is a checksum (see [Ashdown08] for an example of checksums used to detect corruption in application data blocks).

Corruption in each ADH can thus be detected:

o If the guard pattern is anything other than one of the allowed values, including all zeros.

o If the guard pattern is FREE and any other byte in the remainder of the ADH is anything other than zero.

o If the guard pattern is anything other than FREE, then if the stored checksum does not match the computed checksum.

o If the guard pattern is INDIRECT and one of the stored indirect block numbers has a value greater than the number of ADHs in the file.

o If the guard pattern is INDIRECT and one of the stored indirect block numbers is a duplicate of another stored indirect block number.

As can be seen, the application can detect errors based on the combination of the guard pattern state and the checksum. But also, the application can detect corruption based on the state and the contents of the ADH. This last point is important in validating the minimum amount of data we incorporated into our generic framework. I.e., the guard pattern is sufficient in allowing applications to design their own corruption detection.

Finally, it is important to note that none of these corruption checks occur in the transport layer. The server and client components are totally unaware of the file format and might report everything as being transferred correctly even in the case the application detects corruption.

7.3. Example of READ_PLUS

The hypothetical application presented in Section 7.2 can be used to illustrate how READ_PLUS would return an array of results. A file is created and initialized with 100 4k ADHs in the FREE state:

```
WRITE_SAME {0, 4k, 100, 0, 0, 8, 0xfeedface}
```

Further, assume the application writes a single ADH at 16k, changing the guard pattern to 0xcafedead, we would then have in memory:

```
0 -> (16k - 1)   : 4k, 4, 0, 0, 8, 0xfeedface
16k -> (20k - 1) : 00 00 00 05 ca fe de ad XX XX ... XX XX
20k -> 400k      : 4k, 95, 0, 6, 0xfeedface
```

And when the client did a READ_PLUS of 64k at the start of the file, it would get back a result of an ADH, some data, and a final ADH:

```
ADH {0, 4, 0, 0, 8, 0xfeedface}
data 4k
ADH {20k, 4k, 59, 0, 6, 0xfeedface}
```

8. Labeled NFS

8.1. Introduction

Access control models such as Unix permissions or Access Control Lists are commonly referred to as Discretionary Access Control (DAC) models. These systems base their access decisions on user identity and resource ownership. In contrast Mandatory Access Control (MAC) models base their access control decisions on the label on the subject (usually a process) and the object it wishes to access [Haynes13]. These labels may contain user identity information but usually contain additional information. In DAC systems users are free to specify the access rules for resources that they own. MAC models base their security decisions on a system wide policy established by an administrator or organization which the users do not have the ability to override. In this section, we add a MAC model to NFSv4.2.

The first change necessary is to devise a method for transporting and storing security label data on NFSv4 file objects. Security labels have several semantics that are met by NFSv4 recommended attributes such as the ability to set the label value upon object creation. Access control on these attributes are done through a combination of two mechanisms. As with other recommended attributes on file objects the usual DAC checks (ACLs and permission bits) will be performed to ensure that proper file ownership is enforced. In addition a MAC system MAY be employed on the client, server, or both to enforce additional policy on what subjects may modify security label information.

The second change is to provide methods for the client to determine if the security label has changed. A client which needs to know if a label is going to change SHOULD request a delegation on that file. In order to change the security label, the server will have to recall

all delegations. This will inform the client of the change. If a client wants to detect if the label has changed, it MAY use VERIFY and NVERIFY on FATTR4_CHANGE_SEC_LABEL to detect that the FATTR4_SEC_LABEL has been modified.

An additional useful change would be modification to the RPC layer used in NFSv4 to allow RPC calls to carry security labels. Such modifications are outside the scope of this document.

8.2. Definitions

Label Format Specifier (LFS): is an identifier used by the client to establish the syntactic format of the security label and the semantic meaning of its components. These specifiers exist in a registry associated with documents describing the format and semantics of the label.

Label Format Registry: is the IANA registry containing all registered LFS along with references to the documents that describe the syntactic format and semantics of the security label.

Policy Identifier (PI): is an optional part of the definition of a Label Format Specifier which allows for clients and server to identify specific security policies.

Object: is a passive resource within the system that we wish to be protected. Objects can be entities such as files, directories, pipes, sockets, and many other system resources relevant to the protection of the system state.

Subject: is an active entity usually a process which is requesting access to an object.

MAC-Aware: is a server which can transmit and store object labels.

MAC-Functional: is a client or server which is Labeled NFS enabled. Such a system can interpret labels and apply policies based on the security system.

Multi-Level Security (MLS): is a traditional model where objects are given a sensitivity level (Unclassified, Secret, Top Secret, etc) and a category set [MLS].

8.3. MAC Security Attribute

MAC models base access decisions on security attributes bound to subjects and objects. This information can range from a user identity for an identity based MAC model, sensitivity levels for

Multi-level security, or a type for Type Enforcement. These models base their decisions on different criteria but the semantics of the security attribute remain the same. The semantics required by the security attributes are listed below:

- MUST provide flexibility with respect to the MAC model.
- MUST provide the ability to atomically set security information upon object creation.
- MUST provide the ability to enforce access control decisions both on the client and the server.
- MUST NOT expose an object to either the client or server name space before its security information has been bound to it.

NFSv4 implements the security attribute as a recommended attribute. These attributes have a fixed format and semantics, which conflicts with the flexible nature of the security attribute. To resolve this the security attribute consists of two components. The first component is a LFS as defined in [Quigley11] to allow for interoperability between MAC mechanisms. The second component is an opaque field which is the actual security attribute data. To allow for various MAC models, NFSv4 should be used solely as a transport mechanism for the security attribute. It is the responsibility of the endpoints to consume the security attribute and make access decisions based on their respective models. In addition, creation of objects through OPEN and CREATE allows for the security attribute to be specified upon creation. By providing an atomic create and set operation for the security attribute it is possible to enforce the second and fourth requirements. The recommended attribute FATTR4_SEC_LABEL (see Section 12.2.2) will be used to satisfy this requirement.

8.3.1. Delegations

In the event that a security attribute is changed on the server while a client holds a delegation on the file, both the server and the client MUST follow the NFSv4.1 protocol (see Chapter 10 of [RFC5661]) with respect to attribute changes. It SHOULD flush all changes back to the server and relinquish the delegation.

8.3.2. Permission Checking

It is not feasible to enumerate all possible MAC models and even levels of protection within a subset of these models. This means that the NFSv4 client and servers cannot be expected to directly make access control decisions based on the security attribute. Instead

NFSv4 should defer permission checking on this attribute to the host system. These checks are performed in addition to existing DAC and ACL checks outlined in the NFSv4 protocol. Section 8.6 gives a specific example of how the security attribute is handled under a particular MAC model.

8.3.3. Object Creation

When creating files in NFSv4 the OPEN and CREATE operations are used. One of the parameters to these operations is an fattr4 structure containing the attributes the file is to be created with. This allows NFSv4 to atomically set the security attribute of files upon creation. When a client is MAC-Functional it must always provide the initial security attribute upon file creation. In the event that the server is MAC-Functional as well, it should determine by policy whether it will accept the attribute from the client or instead make the determination itself. If the client is not MAC-Functional, then the MAC-Functional server must decide on a default label. A more in depth explanation can be found in Section 8.6.

8.3.4. Existing Objects

Note that under the MAC model, all objects must have labels. Therefore, if an existing server is upgraded to include Labeled NFS support, then it is the responsibility of the security system to define the behavior for existing objects.

8.3.5. Label Changes

If there are open delegations on the file belonging to client other than the one making the label change, then the process described in Section 8.3.1 must be followed. In short, the delegation will be recalled, which effectively notifies the client of the change.

Consider a system in which the clients enforce MAC checks and and the server has a very simple security system which just stores the labels. In this system, the MAC label check always allows access, regardless of the subject label.

The way in which MAC labels are enforced is by the client. The security policies on the client can be such that the client does not have access to the file unless it has a delegation. The recall of the delegation will force the client to flush any cached content of the file. The clients could also be configured to periodically VERIFY/NVERIFY the FATTR4_CHANGE_SEC_LABEL attribute to determine when the label has changed. When a change is detected, then the client could take the costlier action of retrieving the FATTR4_SEC_LABEL.

8.4. pNFS Considerations

The new FATTR4_SEC_LABEL attribute is metadata information and as such the DS is not aware of the value contained on the MDS. Fortunately, the NFSv4.1 protocol [RFC5661] already has provisions for doing access level checks from the DS to the MDS. In order for the DS to validate the subject label presented by the client, it SHOULD utilize this mechanism.

8.5. Discovery of Server Labeled NFS Support

The server can easily determine that a client supports Labeled NFS when it queries for the FATTR4_SEC_LABEL label for an object. The client might need to discover which LFS the server supports.

The following compound MUST NOT be denied by any MAC label check:

```
PUTROOTFH, GETATTR {FATTR4_SEC_LABEL}
```

Note that the server might have imposed a security flavor on the root that precludes such access. I.e., if the server requires kerberized access and the client presents a compound with AUTH_SYS, then the server is allowed to return NFS4ERR_WRONGSEC in this case. But if the client presents a correct security flavor, then the server MUST return the FATTR4_SEC_LABEL attribute with the supported LFS filled in.

8.6. MAC Security NFS Modes of Operation

A system using Labeled NFS may operate in two modes. The first mode provides the most protection and is called "full mode". In this mode both the client and server implement a MAC model allowing each end to make an access control decision. The remaining mode is called the "guest mode" and in this mode one end of the connection is not implementing a MAC model and thus offers less protection than full mode.

8.6.1. Full Mode

Full mode environments consist of MAC-Functional NFSv4 servers and clients and may be composed of mixed MAC models and policies. The system requires that both the client and server have an opportunity to perform an access control check based on all relevant information within the network. The file object security attribute is provided using the mechanism described in Section 8.3.

Fully MAC-Functional NFSv4 servers are not possible in the absence of RPC layer modifications to support subject label transport. However,

servers may make decisions based on the RPC credential information available and future specifications may provide subject label transport.

8.6.1.1. Initial Labeling and Translation

The ability to create a file is an action that a MAC model may wish to mediate. The client is given the responsibility to determine the initial security attribute to be placed on a file. This allows the client to make a decision as to the acceptable security attributes to create a file with before sending the request to the server. Once the server receives the creation request from the client it may choose to evaluate if the security attribute is acceptable.

Security attributes on the client and server may vary based on MAC model and policy. To handle this the security attribute field has an LFS component. This component is a mechanism for the host to identify the format and meaning of the opaque portion of the security attribute. A full mode environment may contain hosts operating in several different LFSs. In this case a mechanism for translating the opaque portion of the security attribute is needed. The actual translation function will vary based on MAC model and policy and is out of the scope of this document. If a translation is unavailable for a given LFS then the request MUST be denied. Another recourse is to allow the host to provide a fallback mapping for unknown security attributes.

8.6.1.2. Policy Enforcement

In full mode access control decisions are made by both the clients and servers. When a client makes a request it takes the security attribute from the requesting process and makes an access control decision based on that attribute and the security attribute of the object it is trying to access. If the client denies that access an RPC call to the server is never made. If however the access is allowed the client will make a call to the NFS server.

When the server receives the request from the client it uses any credential information conveyed in the RPC request and the attributes of the object the client is trying to access to make an access control decision. If the server's policy allows this access it will fulfill the client's request, otherwise it will return NFS4ERR_ACCESS.

Future protocol extensions may also allow the server to factor into the decision a security label extracted from the RPC request.

Implementations MAY validate security attributes supplied over the network to ensure that they are within a set of attributes permitted from a specific peer, and if not, reject them. Note that a system may permit a different set of attributes to be accepted from each peer.

8.6.1.3. Limited Server

A Limited Server mode (see Section 3.5.2 of [Haynes13]) consists of a server which is label aware, but does not enforce policies. Such a server will store and retrieve all object labels presented by clients, utilize the methods described in Section 8.3.5 to allow the clients to detect changing labels, but may not factor the label into access decisions. Instead, it will expect the clients to enforce all such access locally.

8.6.2. Guest Mode

Guest mode implies that either the client or the server does not handle labels. If the client is not Labeled NFS aware, then it will not offer subject labels to the server. The server is the only entity enforcing policy, and may selectively provide standard NFS services to clients based on their authentication credentials and/or associated network attributes (e.g., IP address, network interface). The level of trust and access extended to a client in this mode is configuration-specific. If the server is not Labeled NFS aware, then it will not return object labels to the client. Clients in this environment are may consist of groups implementing different MAC model policies. The system requires that all clients in the environment be responsible for access control checks.

8.7. Security Considerations

This entire chapter deals with security issues.

Depending on the level of protection the MAC system offers there may be a requirement to tightly bind the security attribute to the data.

When only one of the client or server enforces labels, it is important to realize that the other side is not enforcing MAC protections. Alternate methods might be in use to handle the lack of MAC support and care should be taken to identify and mitigate threats from possible tampering outside of these methods.

An example of this is that a server that modifies READDIR or LOOKUP results based on the client's subject label might want to always construct the same subject label for a client which does not present

one. This will prevent a non-Labeled NFS client from mixing entries in the directory cache.

9. Sharing change attribute implementation details with NFSv4 clients

9.1. Introduction

Although both the NFSv4 [I-D.ietf-nfsv4-rfc3530bis] and NFSv4.1 protocol [RFC5661], define the change attribute as being mandatory to implement, there is little in the way of guidance. The only mandated feature is that the value must change whenever the file data or metadata change.

While this allows for a wide range of implementations, it also leaves the client with a conundrum: how does it determine which is the most recent value for the change attribute in a case where several RPC calls have been issued in parallel? In other words if two COMPOUNDs, both containing WRITE and GETATTR requests for the same file, have been issued in parallel, how does the client determine which of the two change attribute values returned in the replies to the GETATTR requests correspond to the most recent state of the file? In some cases, the only recourse may be to send another COMPOUND containing a third GETATTR that is fully serialized with the first two.

NFSv4.2 avoids this kind of inefficiency by allowing the server to share details about how the change attribute is expected to evolve, so that the client may immediately determine which, out of the several change attribute values returned by the server, is the most recent. change_attr_type is defined as a new recommended attribute (see Section 12.2.1), and is per file system.

10. Security Considerations

NFSv4.2 has all of the security concerns present in NFSv4.1 (see Section 21 of [RFC5661]) and those present in the Server-side Copy (see Section 3.4) and in Labeled NFS (see Section 8.7).

11. 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.

11.1. Error Definitions

Protocol Error Definitions

Error	Number	Description
NFS4ERR_BADLABEL	10093	Section 11.1.3.1
NFS4ERR_OFFLOAD_DENIED	10091	Section 11.1.2.1
NFS4ERR_PARTNER_NO_AUTH	10089	Section 11.1.2.2
NFS4ERR_PARTNER_NOTSUPP	10088	Section 11.1.2.3
NFS4ERR_UNION_NOTSUPP	10090	Section 11.1.1.1
NFS4ERR_WRONG_LFS	10092	Section 11.1.3.2

Table 1

11.1.1. General Errors

This section deals with errors that are applicable to a broad set of different purposes.

11.1.1.1. NFS4ERR_UNION_NOTSUPP (Error Code 10090)

One of the arguments to the operation is a discriminated union and while the server supports the given operation, it does not support the selected arm of the discriminated union.

11.1.2. Server to Server Copy Errors

These errors deal with the interaction between server to server copies.

11.1.2.1. NFS4ERR_OFFLOAD_DENIED (Error Code 10091)

The copy offload operation is supported by both the source and the destination, but the destination is not allowing it for this file. If the client sees this error, it should fall back to the normal copy semantics.

11.1.2.2. NFS4ERR_PARTNER_NO_AUTH (Error Code 10089)

The source server does not authorize a server-to-server copy offload operation. This may be due to the client's failure to send the COPY_NOTIFY operation to the source server, the source server receiving a server-to-server copy offload request after the copy lease time expired, or for some other permission problem.

11.1.2.3. NFS4ERR_PARTNER_NOTSUPP (Error Code 10088)

The remote server does not support the server-to-server copy offload protocol.

11.1.3. Labeled NFS Errors

These errors are used in Labeled NFS.

11.1.3.1. NFS4ERR_BADLABEL (Error Code 10093)

The label specified is invalid in some manner.

11.1.3.2. NFS4ERR_WRONG_LFS (Error Code 10092)

The LFS specified in the subject label is not compatible with the LFS in the object label.

11.2. New Operations and Their Valid Errors

This section contains a table that gives the valid error returns for each new NFSv4.2 protocol operation. The error code NFS4_OK (indicating no error) is not listed but should be understood to be returnable by all new operations. The error values for all other operations are defined in Section 15.2 of [RFC5661].

Valid Error Returns for Each New Protocol Operation

Operation	Errors
COPY	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_METADATA_NOTSUPP, NFS4ERR_MOVED, NFS4ERR_NOFILEHANDLE, NFS4ERR_NOSPC, NFS4ERR_OFFLOAD_DENIED, NFS4ERR_OLD_STATEID, NFS4ERR_OPENMODE, NFS4ERR_OP_NOT_IN_SESSION, NFS4ERR_PARTNER_NO_AUTH, NFS4ERR_PARTNER_NOTSUPP, 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
COPY_NOTIFY	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
OFFLOAD_ABORT	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR,
	NFS4ERR_BAD_STATEID, NFS4ERR_COMPLETE_ALREADY,
	NFS4ERR_DEADSESSION, NFS4ERR_EXPIRED,
	NFS4ERR_DELAY, NFS4ERR_GRACE, NFS4ERR_NOTSUPP,
	NFS4ERR_OLD_STATEID, NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS
OFFLOAD_REVOKE	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR,
	NFS4ERR_COMPLETE_ALREADY, NFS4ERR_DELAY,
	NFS4ERR_GRACE, NFS4ERR_INVALID, NFS4ERR_MOVED,
	NFS4ERR_NOTSUPP, NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS
OFFLOAD_STATUS	NFS4ERR_ADMIN_REVOKED, NFS4ERR_BADXDR,
	NFS4ERR_BAD_STATEID, NFS4ERR_COMPLETE_ALREADY,
	NFS4ERR_DEADSESSION, NFS4ERR_EXPIRED,
	NFS4ERR_DELAY, NFS4ERR_GRACE, NFS4ERR_NOTSUPP,
	NFS4ERR_OLD_STATEID, NFS4ERR_OP_NOT_IN_SESSION,
	NFS4ERR_SERVERFAULT, NFS4ERR_TOO_MANY_OPS
READ_PLUS	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
SEEK	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_UNION_NOTSUPP, NFS4ERR_WRONG_TYPE
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
WRITE_HOLE	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_NOTSUPP, 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
WRITE_SAME	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_NOTSUPP, 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 2

11.3. New Callback Operations and Their Valid Errors

This section contains a table that gives the valid error returns for each new NFSv4.2 callback operation. The error code NFS4_OK (indicating no error) is not listed but should be understood to be returnable by all new callback operations. The error values for all other callback operations are defined in Section 15.3 of [RFC5661].

Valid Error Returns for Each New Protocol Callback Operation

Callback Operation	Errors
CB_OFFLOAD	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

Table 3

12. New File Attributes

12.1. New RECOMMENDED Attributes - List and Definition References

The list of new RECOMMENDED 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 assigned number and [NFSv42xdr], the latter is likely authoritative, but should be resolved with Errata to this document and/or [NFSv42xdr]. See [IESG08] 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
change_attr_type	79	change_attr_type4	R	Section 12.2.1
sec_label	80	sec_label4	R W	Section 12.2.2
change_sec_label	81	change_sec_label4	R	Section 12.2.3
space_reserved	77	boolean	R W	Section 12.2.4
space_freed	78	length4	R	Section 12.2.5

Table 4

12.2. Attribute Definitions

12.2.1. Attribute 79: change_attr_type

```
enum change_attr_type4 {
           NFS4_CHANGE_TYPE_IS_MONOTONIC_INCR         = 0,
           NFS4_CHANGE_TYPE_IS_VERSION_COUNTER        = 1,
           NFS4_CHANGE_TYPE_IS_VERSION_COUNTER_NOPNFS = 2,
           NFS4_CHANGE_TYPE_IS_TIME_METADATA          = 3,
           NFS4_CHANGE_TYPE_IS_UNDEFINED              = 4
};
```

change_attr_type is a per file system attribute which enables the NFSv4.2 server to provide additional information about how it expects the change attribute value to evolve after the file data, or metadata has changed. While Section 5.4 of [RFC5661] discusses per file system attributes, it is expected that the value of change_attr_type

not depend on the value of "homogeneous" and only changes in the event of a migration.

NFS4_CHANGE_TYPE_IS_UNDEFINED: The change attribute does not take values that fit into any of these categories.

NFS4_CHANGE_TYPE_IS_MONOTONIC_INCR: The change attribute value MUST monotonically increase for every atomic change to the file attributes, data, or directory contents.

NFS4_CHANGE_TYPE_IS_VERSION_COUNTER: The change attribute value MUST be incremented by one unit for every atomic change to the file attributes, data, or directory contents. This property is preserved when writing to pNFS data servers.

NFS4_CHANGE_TYPE_IS_VERSION_COUNTER_NOPNFS: The change attribute value MUST be incremented by one unit for every atomic change to the file attributes, data, or directory contents. In the case where the client is writing to pNFS data servers, the number of increments is not guaranteed to exactly match the number of writes.

NFS4_CHANGE_TYPE_IS_TIME_METADATA: The change attribute is implemented as suggested in the NFSv4 spec [I-D.ietf-nfsv4-rfc3530bis] in terms of the time_metadata attribute.

If either NFS4_CHANGE_TYPE_IS_MONOTONIC_INCR, NFS4_CHANGE_TYPE_IS_VERSION_COUNTER, or NFS4_CHANGE_TYPE_IS_TIME_METADATA are set, then the client knows at the very least that the change attribute is monotonically increasing, which is sufficient to resolve the question of which value is the most recent.

If the client sees the value NFS4_CHANGE_TYPE_IS_TIME_METADATA, then by inspecting the value of the 'time_delta' attribute it additionally has the option of detecting rogue server implementations that use time_metadata in violation of the spec.

If the client sees NFS4_CHANGE_TYPE_IS_VERSION_COUNTER, it has the ability to predict what the resulting change attribute value should be after a COMPOUND containing a SETATTR, WRITE, or CREATE. This again allows it to detect changes made in parallel by another client. The value NFS4_CHANGE_TYPE_IS_VERSION_COUNTER_NOPNFS permits the same, but only if the client is not doing pNFS WRITEs.

Finally, if the server does not support change_attr_type or if NFS4_CHANGE_TYPE_IS_UNDEFINED is set, then the server SHOULD make an

effort to implement the change attribute in terms of the time_metadata attribute.

12.2.2. Attribute 80: sec_label

```
typedef uint32_t  policy4;

struct labelformat_spec4 {
        policy4 lfs_lfs;
        policy4 lfs_pi;
};

struct sec_label4 {
        labelformat_spec4       slai_lfs;
        opaque                  slai_data<>;
};
```

The FATTR4_SEC_LABEL contains an array of two components with the first component being an LFS. It serves to provide the receiving end with the information necessary to translate the security attribute into a form that is usable by the endpoint. Label Formats assigned an LFS may optionally choose to include a Policy Identifier field to allow for complex policy deployments. The LFS and Label Format Registry are described in detail in [Quigley11]. The translation used to interpret the security attribute is not specified as part of the protocol as it may depend on various factors. The second component is an opaque section which contains the data of the attribute. This component is dependent on the MAC model to interpret and enforce.

In particular, it is the responsibility of the LFS specification to define a maximum size for the opaque section, slai_data<>. When creating or modifying a label for an object, the client needs to be guaranteed that the server will accept a label that is sized correctly. By both client and server being part of a specific MAC model, the client will be aware of the size.

If a server supports sec_label, then it MUST also support change_sec_label. Any modification to sec_label MUST modify the value for change_sec_label.

12.2.3. Attribute 81: change_sec_label

The change_sec_label attribute is a read-only attribute per file. If the value of sec_label for a file is not the same at two disparate times then the values of change_sec_label at those times MUST be different as well. The value of change_sec_label MAY change at other times as well, but this should be rare, as that will require the

client to abort any operation in progress, re-read the label, and retry the operation. As the sec_label is not bounded by size, this attribute allows for VERIFY and NVERIFY to quickly determine if the sec_label has been modified.

12.2.4. Attribute 77: space_reserved

The space_reserve attribute is a read/write attribute of type boolean. It is a per file attribute and applies during the lifetime of the file or until it is turned off. When the space_reserved attribute is set via SETATTR, the server must ensure that there is disk space to accommodate every byte in the file before it can return success. If the server cannot guarantee this, it must return NFS4ERR_NOSPC.

If the client tries to grow a file which has the space_reserved attribute set, the server must guarantee that there is disk space to accommodate every byte in the file with the new size before it can return success. If the server cannot guarantee this, it must return NFS4ERR_NOSPC.

It is not required that the server allocate the space to the file before returning success. The allocation can be deferred, however, it must be guaranteed that it will not fail for lack of space.

The value of space_reserved can be obtained at any time through GETATTR. If the size is retrieved at the same time, the client can determine the size of the reservation.

In order to avoid ambiguity, the space_reserve bit cannot be set along with the size bit in SETATTR. Increasing the size of a file with space_reserve set will fail if space reservation cannot be guaranteed for the new size. If the file size is decreased, space reservation is only guaranteed for the new size. If a hole is punched into the file, then the reservation is not changed.

12.2.5. Attribute 78: space_freed

space_freed gives the number of bytes freed if the file is deleted. This attribute is read only and is of type length4. It is a per file attribute.

13. Operations: REQUIRED, RECOMMENDED, or OPTIONAL

The following tables summarize the operations of the NFSv4.2 protocol and the corresponding designation of REQUIRED, RECOMMENDED, and OPTIONAL to implement or either OBSOLESCENT or MUST NOT implement. The designation of OBSOLESCENT is reserved for those operations which

are defined in either NFSv4.0 or NFSv4.1 and are intended to be classified as MUST NOT be implemented in NFSv4.3. The designation of MUST NOT implement is reserved for those operations that were defined in either NFSv4.0 or NFSV4.1 and MUST NOT be implemented in NFSv4.2.

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 NFSv4.2 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 either specification or that of NFSv4.1 [RFC5661].

The abbreviations used in the second and third columns of the table are defined as follows.

REQ REQUIRED to implement

REC RECOMMENDED to implement

OPT OPTIONAL to implement

MNI MUST NOT implement

OBS Also OBSOLESCENT for future versions.

For the NFSv4.2 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

COPY Server Side Copy

ADH Application Data Holes

Operations

Operation	EOL, REQ, REC, OPT, or MNI	Feature (REQ, REC, or OPT)
ACCESS	REQ	
BACKCHANNEL_CTL	REQ	
BIND_CONN_TO_SESSION	REQ	
CLOSE	REQ	
COMMIT	REQ	
COPY	OPT	COPY (REQ)
OFFLOAD_ABORT	OPT	COPY (REQ)
COPY_NOTIFY	OPT	COPY (REQ)
OFFLOAD_REVOKE	OPT	COPY (REQ)
OFFLOAD_STATUS	OPT	COPY (REQ)
CREATE	REQ	
CREATE_SESSION	REQ	
DELEGPURGE	OPT	FDELG (REQ)
DELEGRETURN	OPT	FDELG, DDELG, pNFS (REQ)
DESTROY_CLIENTID	REQ	
DESTROY_SESSION	REQ	
EXCHANGE_ID	REQ	
FREE_STATEID	REQ	
GETATTR	REQ	
GETDEVICEINFO	OPT	pNFS (REQ)
GETDEVICELIST	OPT	pNFS (OPT)
GETFH	REQ	
GET_DIR_DELEGATION	OPT	DDELG (REQ)
LAYOUTCOMMIT	OPT	pNFS (REQ)
LAYOUTGET	OPT	pNFS (REQ)
LAYOUTRETURN	OPT	pNFS (REQ)
LINK	OPT	
LOCK	REQ	

LOCKT	REQ	
LOCKU	REQ	
LOOKUP	REQ	
LOOKUPP	REQ	
NVERIFY	REQ	
OPEN	REQ	
OPENATTR	OPT	
OPEN_CONFIRM	MNI	
OPEN_DOWNGRADE	REQ	
PUTFH	REQ	
PUTPUBFH	REQ	
PUTROOTFH	REQ	
READ	REQ (OBS)	
READDIR	REQ	
READLINK	OPT	
READ_PLUS	OPT	ADH (REQ)
RECLAIM_COMPLETE	REQ	
RELEASE_LOCKOWNER	MNI	
REMOVE	REQ	
RENAME	REQ	
RENEW	MNI	
RESTOREFH	REQ	
SAVEFH	REQ	
SECINFO	REQ	
SECINFO_NO_NAME	REC	pNFS file layout (REQ)
SEQUENCE	REQ	
SETATTR	REQ	
SETCLIENTID	MNI	
SETCLIENTID_CONFIRM	MNI	
SET_SSV	REQ	
TEST_STATEID	REQ	
VERIFY	REQ	
WANT_DELEGATION	OPT	FDELG (OPT)
WRITE	REQ	
WRITE_HOLE	OPT	
WRITE_SAME	OPT	ADH (REQ)

Callback Operations

Operation	REQ, REC, OPT, or MNI	Feature (REQ, REC, or OPT)
CB_OFFLOAD	OPT	COPY (REQ)
CB_GETATTR	OPT	FDELG (REQ)
CB_LAYOUTRECALL	OPT	pNFS (REQ)
CB_NOTIFY	OPT	DDELG (REQ)
CB_NOTIFY_DEVICEID	OPT	pNFS (OPT)
CB_NOTIFY_LOCK	OPT	
CB_PUSH_DELEG	OPT	FDELG (OPT)
CB_RECALL	OPT	FDELG, DDELG, pNFS (REQ)
CB_RECALL_ANY	OPT	FDELG, DDELG, pNFS (REQ)
CB_RECALL_SLOT	REQ	
CB_RECALLABLE_OBJ_AVAIL	OPT	DDELG, pNFS (REQ)
CB_SEQUENCE	OPT	FDELG, DDELG, pNFS (REQ)
CB_WANTS_CANCELLED	OPT	FDELG, DDELG, pNFS (REQ)

14. NFSv4.2 Operations

14.1. Operation 59: COPY - Initiate a server-side copy

14.1.1. ARGUMENT

```
struct COPY4args {
        /* SAVED_FH: source file */
        /* CURRENT_FH: destination file */
        stateid4        ca_src_stateid;
        stateid4        ca_dst_stateid;
        offset4         ca_src_offset;
        offset4         ca_dst_offset;
        length4         ca_count;
        netloc4         ca_source_server<>;
};
```

14.1.2. RESULT

```
union COPY4res switch (nfsstat4 cr_status) {
case NFS4_OK:
        write_response4 resok4;
default:
        length4         cr_bytes_copied;
};
```

14.1.3. DESCRIPTION

The COPY operation is used for both intra-server and inter-server copies. In both cases, the COPY is always sent from the client to the destination server of the file copy. The COPY operation requests that a file be copied from the location specified by the SAVED_FH value to the location specified by the CURRENT_FH.

The SAVED_FH must be a regular file. If SAVED_FH is not a regular file, the operation MUST fail and return NFS4ERR_WRONG_TYPE.

In order to set SAVED_FH to the source file handle, the compound procedure requesting the COPY will include a sub-sequence of operations such as

```
PUTFH source-fh
SAVEFH
```

If the request is for a server-to-server copy, the source-fh is a filehandle from the source server and the compound procedure is being executed on the destination server. In this case, the source-fh is a foreign filehandle on the server receiving the COPY request. If either PUTFH or SAVEFH checked the validity of the filehandle, the operation would likely fail and return NFS4ERR_STALE.

If a server supports the server-to-server COPY feature, a PUTFH followed by a SAVEFH MUST NOT return NFS4ERR_STALE for either operation. These restrictions do not pose substantial difficulties for servers. The CURRENT_FH and SAVED_FH may be validated in the context of the operation referencing them and an NFS4ERR_STALE error returned for an invalid file handle at that point.

For an intra-server copy, both the ca_src_stateid and ca_dst_stateid MUST refer to either open or locking states provided earlier by the server. If either stateid is invalid, then the operation MUST fail. If the request is for a inter-server copy, then the ca_src_stateid can be ignored. If ca_dst_stateid is invalid, then the operation MUST fail.

The CURRENT_FH specifies the destination of the copy operation. The CURRENT_FH MUST be a regular file and not a directory. Note, the file MUST exist before the COPY operation begins. It is the responsibility of the client to create the file if necessary, regardless of the actual copy protocol used. If the file cannot be created in the destination file system (due to file name restrictions, such as case or length), the COPY operation MUST NOT be called.

The ca_src_offset is the offset within the source file from which the data will be read, the ca_dst_offset is the offset within the destination file to which the data will be written, and the ca_count is the number of bytes that will be copied. An offset of 0 (zero) specifies the start of the file. A count of 0 (zero) requests that all bytes from ca_src_offset through EOF be copied to the destination. If concurrent modifications to the source file overlap with the source file region being copied, the data copied may include all, some, or none of the modifications. The client can use standard NFS operations (e.g., OPEN with OPEN4_SHARE_DENY_WRITE or mandatory byte range locks) to protect against concurrent modifications if the client is concerned about this. If the source file's end of file is being modified in parallel with a copy that specifies a count of 0 (zero) bytes, the amount of data copied is implementation dependent (clients may guard against this case by specifying a non-zero count value or preventing modification of the source file as mentioned above).

If the source offset or the source offset plus count is greater than or equal to the size of the source file, the operation will fail with NFS4ERR_INVAL. The destination offset or destination offset plus count may be greater than the size of the destination file. This allows for the client to issue parallel copies to implement operations such as "cat file1 file2 file3 file4 > dest".

If the ca_source_server list is specified, then this is an inter-server copy operation and the source file is on a remote server. The client is expected to have previously issued a successful COPY_NOTIFY request to the remote source server. The ca_source_server list MUST be the same as the COPY_NOTIFY response's cnr_source_server list. If the client includes the entries from the COPY_NOTIFY response's cnr_source_server list in the ca_source_server list, the source server can indicate a specific copy protocol for the destination server to use by returning a URL, which specifies both a protocol service and server name. Server-to-server copy protocol considerations are described in Section 3.2.5 and Section 3.4.1.

The copying of any and all attributes on the source file is the responsibility of both the client and the copy protocol. Any

attribute which is both exposed via the NFS protocol on the source file and set SHOULD be copied to the destination file. Any attribute supported by the destination server that is not set on the source file SHOULD be left unset. If the client cannot copy an attribute from the source to destination, it MAY fail the copy transaction.

Metadata attributes not exposed via the NFS protocol SHOULD be copied to the destination file where appropriate via the copy protocol. Note that if the copy protocol is NFSv4.x, then these attributes will be lost.

The destination file's named attributes are not duplicated from the source file. After the copy process completes, the client MAY attempt to duplicate named attributes using standard NFSv4 operations. However, the destination file's named attribute capabilities MAY be different from the source file's named attribute capabilities.

If the operation does not result in an immediate failure, the server will return NFS4_OK, and the CURRENT_FH will remain the destination's filehandle.

If an immediate failure does occur, cr_bytes_copied will be set to the number of bytes copied to the destination file before the error occurred. The cr_bytes_copied value indicates the number of bytes copied but not which specific bytes have been copied.

A return of NFS4_OK indicates that either the operation is complete or the operation was initiated and a callback will be used to deliver the final status of the operation.

If the wr_callback_id is returned, this indicates that the operation was initiated and a CB_OFFLOAD callback will deliver the final results of the operation. The wr_callback_id stateid is termed a copy stateid in this context. The server is given the option of returning the results in a callback because the data may require a relatively long period of time to copy.

If no wr_callback_id is returned, the operation completed synchronously and no callback will be issued by the server. The completion status of the operation is indicated by cr_status.

If the copy completes successfully, either synchronously or asynchronously, the data copied from the source file to the destination file MUST appear identical to the NFS client. However, the NFS server's on disk representation of the data in the source file and destination file MAY differ. For example, the NFS server

might encrypt, compress, deduplicate, or otherwise represent the on disk data in the source and destination file differently.

14.2. Operation 60: OFFLOAD_ABORT - Cancel a server-side copy

14.2.1. ARGUMENT

```
struct OFFLOAD_ABORT4args {
        /* CURRENT_FH: destination file */
        stateid4        oaa_stateid;
};
```

14.2.2. RESULT

```
struct OFFLOAD_ABORT4res {
        nfsstat4        oar_status;
};
```

14.2.3. DESCRIPTION

OFFLOAD_ABORT is used for both intra- and inter-server asynchronous copies. The OFFLOAD_ABORT operation allows the client to cancel a server-side copy operation that it initiated. This operation is sent in a COMPOUND request from the client to the destination server. This operation may be used to cancel a copy when the application that requested the copy exits before the operation is completed or for some other reason.

The request contains the filehandle and copy stateid cookies that act as the context for the previously initiated copy operation.

The result's oar_status field indicates whether the cancel was successful or not. A value of NFS4_OK indicates that the copy operation was canceled and no callback will be issued by the server. A copy operation that is successfully canceled may result in none, some, or all of the data and/or metadata copied.

If the server supports asynchronous copies, the server is REQUIRED to support the OFFLOAD_ABORT operation.

14.3. Operation 61: COPY_NOTIFY - Notify a source server of a future copy

14.3.1. ARGUMENT

```
struct COPY_NOTIFY4args {
        /* CURRENT_FH: source file */
        stateid4        cna_src_stateid;
        netloc4         cna_destination_server;
};
```

14.3.2. RESULT

```
struct COPY_NOTIFY4resok {
        nfstime4        cnr_lease_time;
        netloc4         cnr_source_server<>;
};

union COPY_NOTIFY4res switch (nfsstat4 cnr_status) {
case NFS4_OK:
        COPY_NOTIFY4resok       resok4;
default:
        void;
};
```

14.3.3. DESCRIPTION

This operation is used for an inter-server copy. A client sends this operation in a COMPOUND request to the source server to authorize a destination server identified by cna_destination_server to read the file specified by CURRENT_FH on behalf of the given user.

The cna_src_stateid MUST refer to either open or locking states provided earlier by the server. If it is invalid, then the operation MUST fail.

The cna_destination_server MUST be specified using the netloc4 network location format. The server is not required to resolve the cna_destination_server address before completing this operation.

If this operation succeeds, the source server will allow the cna_destination_server to copy the specified file on behalf of the given user as long as both of the following conditions are met:

o The destination server begins reading the source file before the cnr_lease_time expires. If the cnr_lease_time expires while the destination server is still reading the source file, the destination server is allowed to finish reading the file.

o The client has not issued a COPY_REVOKE for the same combination of user, filehandle, and destination server.

The cnr_lease_time is chosen by the source server. A cnr_lease_time of 0 (zero) indicates an infinite lease. To avoid the need for synchronized clocks, copy lease times are granted by the server as a time delta. To renew the copy lease time the client should resend the same copy notification request to the source server.

A successful response will also contain a list of netloc4 network location formats called cnr_source_server, on which the source is willing to accept connections from the destination. These might not be reachable from the client and might be located on networks to which the client has no connection.

If the client wishes to perform an inter-server copy, the client MUST send a COPY_NOTIFY to the source server. Therefore, the source server MUST support COPY_NOTIFY.

For a copy only involving one server (the source and destination are on the same server), this operation is unnecessary.

14.4. Operation 62: OFFLOAD_REVOKE - Revoke a destination server's copy privileges

14.4.1. ARGUMENT

```
struct OFFLOAD_REVOKE4args {
        /* CURRENT_FH: source file */
        netloc4         ora_destination_server;
};
```

14.4.2. RESULT

```
struct OFFLOAD_REVOKE4res {
        nfsstat4        orr_status;
};
```

14.4.3. DESCRIPTION

This operation is used for an inter-server copy. A client sends this operation in a COMPOUND request to the source server to revoke the authorization of a destination server identified by ora_destination_server from reading the file specified by CURRENT_FH on behalf of given user. If the ora_destination_server has already

begun copying the file, a successful return from this operation indicates that further access will be prevented.

The ora_destination_server MUST be specified using the netloc4 network location format. The server is not required to resolve the ora_destination_server address before completing this operation.

The client uses OFFLOAD_ABORT to inform the destination to stop the active transfer and OFFLOAD_REVOKE to inform the source to not allow any more copy requests from the destination. The OFFLOAD_REVOKE operation is also useful in situations in which the source server granted a very long or infinite lease on the destination server's ability to read the source file and all copy operations on the source file have been completed.

For a copy only involving one server (the source and destination are on the same server), this operation is unnecessary.

If the server supports COPY_NOTIFY, the server is REQUIRED to support the OFFLOAD_REVOKE operation.

14.5. Operation 63: OFFLOAD_STATUS - Poll for status of a server-side copy

14.5.1. ARGUMENT

```
struct OFFLOAD_STATUS4args {
        /* CURRENT_FH: destination file */
        stateid4        osa_stateid;
};
```

14.5.2. RESULT

```
struct OFFLOAD_STATUS4resok {
        length4         osr_bytes_copied;
        nfsstat4        osr_complete<1>;
};

union OFFLOAD_STATUS4res switch (nfsstat4 osr_status) {
case NFS4_OK:
        OFFLOAD_STATUS4resok            osr_resok4;
default:
        void;
};
```

14.5.3. DESCRIPTION

OFFLOAD_STATUS is used for both intra- and inter-server asynchronous copies. The OFFLOAD_STATUS operation allows the client to poll the destination server to determine the status of an asynchronous copy operation.

If this operation is successful, the number of bytes copied are returned to the client in the osr_bytes_copied field. The osr_bytes_copied value indicates the number of bytes copied but not which specific bytes have been copied.

If the optional osr_complete field is present, the copy has completed. In this case the status value indicates the result of the asynchronous copy operation. In all cases, the server will also deliver the final results of the asynchronous copy in a CB_OFFLOAD operation.

The failure of this operation does not indicate the result of the asynchronous copy in any way.

If the server supports asynchronous copies, the server is REQUIRED to support the OFFLOAD_STATUS operation.

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

14.6.1. ARGUMENT

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

14.6.2. RESULT

Unchanged

14.6.3. MOTIVATION

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

14.6.4. DESCRIPTION

A client SHOULD request the EXCHGID4_FLAG_SUPP_FENCE_OPS capability when it sends an EXCHANGE_ID operation. The server SHOULD set this capability in the EXCHANGE_ID reply whether the client requests it or not. It is the server's return that determines whether this capability is in effect. When it is in effect, the following will occur:

- The server will not reply to any DESTROY_SESSION invoked with the client ID until all operations in progress are completed or aborted.

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

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

14.7. Operation 67: IO_ADVISE - Application I/O access pattern hints

14.7.1. ARGUMENT

```
enum IO_ADVISE_type4 {
        IO_ADVISE4_NORMAL                       = 0,
        IO_ADVISE4_SEQUENTIAL                   = 1,
        IO_ADVISE4_SEQUENTIAL_BACKWARDS         = 2,
        IO_ADVISE4_RANDOM                       = 3,
        IO_ADVISE4_WILLNEED                     = 4,
        IO_ADVISE4_WILLNEED_OPPORTUNISTIC       = 5,
        IO_ADVISE4_DONTNEED                     = 6,
        IO_ADVISE4_NOREUSE                      = 7,
        IO_ADVISE4_READ                         = 8,
        IO_ADVISE4_WRITE                        = 9,
        IO_ADVISE4_INIT_PROXIMITY               = 10
};

struct IO_ADVISE4args {
        /* CURRENT_FH: file */
        stateid4        iar_stateid;
        offset4         iar_offset;
        length4         iar_count;
        bitmap4         iar_hints;
};
```

14.7.2. RESULT

```
struct IO_ADVISE4resok {
        bitmap4 ior_hints;
};

union IO_ADVISE4res switch (nfsstat4 _status) {
case NFS4_OK:
        IO_ADVISE4resok resok4;
default:
        void;
};
```

14.7.3. DESCRIPTION

The IO_ADVISE operation sends an I/O access pattern hint to the server for the owner of the stateid for a given byte range specified by iar_offset and iar_count. The byte range specified by iar_offset and iar_count need not currently exist in the file, but the iar_hints will apply to the byte range when it does exist. If iar_count is 0, all data following iar_offset is specified. The server MAY ignore the advice.

The following are the allowed hints for a stateid holder:

IO_ADVISE4_NORMAL There is no advice to give, this is the default behavior.

IO_ADVISE4_SEQUENTIAL Expects to access the specified data sequentially from lower offsets to higher offsets.

IO_ADVISE4_SEQUENTIAL_BACKWARDS Expects to access the specified data sequentially from higher offsets to lower offsets.

IO_ADVISE4_RANDOM Expects to access the specified data in a random order.

IO_ADVISE4_WILLNEED Expects to access the specified data in the near future.

IO_ADVISE4_WILLNEED_OPPORTUNISTIC Expects to possibly access the data in the near future. This is a speculative hint, and therefore the server should prefetch data or indirect blocks only if it can be done at a marginal cost.

IO_ADVISE_DONTNEED Expects that it will not access the specified data in the near future.

IO_ADVISE_NOREUSE Expects to access the specified data once and then not reuse it thereafter.

IO_ADVISE4_READ Expects to read the specified data in the near future.

IO_ADVISE4_WRITE Expects to write the specified data in the near future.

IO_ADVISE4_INIT_PROXIMITY Informs the server that the data in the byte range remains important to the client.

Since IO_ADVISE is a hint, a server SHOULD NOT return an error and invalidate a entire Compound request if one of the sent hints in iar_hints is not supported by the server. Also, the server MUST NOT return an error if the client sends contradictory hints to the server, e.g., IO_ADVISE4_SEQUENTIAL and IO_ADVISE4_RANDOM in a single IO_ADVISE operation. In these cases, the server MUST return success and a ior_hints value that indicates the hint it intends to implement. This may mean simply returning IO_ADVISE4_NORMAL.

The ior_hints returned by the server is primarily for debugging purposes since the server is under no obligation to carry out the hints that it describes in the ior_hints result. In addition, while the server may have intended to implement the hints returned in

ior_hints, as time progresses, the server may need to change its handling of a given file due to several reasons including, but not limited to, memory pressure, additional IO_ADVISE hints sent by other clients, and heuristically detected file access patterns.

The server MAY return different advice than what the client requested. If it does, then this might be due to one of several conditions, including, but not limited to another client advising of a different I/O access pattern; a different I/O access pattern from another client that that the server has heuristically detected; or the server is not able to support the requested I/O access pattern, perhaps due to a temporary resource limitation.

Each issuance of the IO_ADVISE operation overrides all previous issuances of IO_ADVISE for a given byte range. This effectively follows a strategy of last hint wins for a given stateid and byte range.

Clients should assume that hints included in an IO_ADVISE operation will be forgotten once the file is closed.

14.7.4. IMPLEMENTATION

The NFS client may choose to issue an IO_ADVISE operation to the server in several different instances.

The most obvious is in direct response to an application's execution of posix_fadvise(). In this case, IO_ADVISE4_WRITE and IO_ADVISE4_READ may be set based upon the type of file access specified when the file was opened.

14.7.5. IO_ADVISE4_INIT_PROXIMITY

The IO_ADVISE4_INIT_PROXIMITY hint is non-posix in origin and conveys that the client has recently accessed the byte range in its own cache. I.e., it has not accessed it on the server, but it has locally. When the server reaches resource exhaustion, knowing which data is more important allows the server to make better choices about which data to, for example purge from a cache, or move to secondary storage. It also informs the server which delegations are more important, since if delegations are working correctly, once delegated to a client and the client has read the content for that byte range, a server might never receive another read request for that byte range.

This hint is also useful in the case of NFS clients which are network booting from a server. If the first client to be booted sends this hint, then it keeps the cache warm for the remaining clients.

14.7.6. pNFS File Layout Data Type Considerations

The IO_ADVISE considerations for pNFS are very similar to the COMMIT considerations for pNFS. That is, as with COMMIT, some NFS server implementations prefer IO_ADVISE be done on the DS, and some prefer it be done on the MDS.

So for the file's layout type, it is proposed that NFSv4.2 include an additional hint NFL42_CARE_IO_ADVISE_THRU_MDS which is valid only on NFSv4.2 or higher. Any file's layout obtained with NFSv4.1 MUST NOT have NFL42_UFLG_IO_ADVISE_THRU_MDS set. Any file's layout obtained with NFSv4.2 MAY have NFL42_UFLG_IO_ADVISE_THRU_MDS set. If the client does not implement IO_ADVISE, then it MUST ignore NFL42_UFLG_IO_ADVISE_THRU_MDS.

If NFL42_UFLG_IO_ADVISE_THRU_MDS is set, the client MUST send the IO_ADVISE operation to the MDS in order for it to be honored by the DS. Once the MDS receives the IO_ADVISE operation, it will communicate the advice to each DS.

If NFL42_UFLG_IO_ADVISE_THRU_MDS is not set, then the client SHOULD send an IO_ADVISE operation to the appropriate DS for the specified byte range. While the client MAY always send IO_ADVISE to the MDS, if the server has not set NFL42_UFLG_IO_ADVISE_THRU_MDS, the client should expect that such an IO_ADVISE is futile. Note that a client SHOULD use the same set of arguments on each IO_ADVISE sent to a DS for the same open file reference.

The server is not required to support different advice for different DS's with the same open file reference.

14.7.6.1. Dense and Sparse Packing Considerations

The IO_ADVISE operation MUST use the iar_offset and byte range as dictated by the presence or absence of NFL4_UFLG_DENSE.

E.g., if NFL4_UFLG_DENSE is present, and a READ or WRITE to the DS for iar_offset 0 really means iar_offset 10000 in the logical file, then an IO_ADVISE for iar_offset 0 means iar_offset 10000.

E.g., if NFL4_UFLG_DENSE is absent, then a READ or WRITE to the DS for iar_offset 0 really means iar_offset 0 in the logical file, then an IO_ADVISE for iar_offset 0 means iar_offset 0 in the logical file.

E.g., if NFL4_UFLG_DENSE is present, the stripe unit is 1000 bytes and the stripe count is 10, and the dense DS file is serving iar_offset 0. A READ or WRITE to the DS for iar_offsets 0, 1000, 2000, and 3000, really mean iar_offsets 10000, 20000, 30000, and

40000 (implying a stripe count of 10 and a stripe unit of 1000), then an IO_ADVISE sent to the same DS with an iar_offset of 500, and a iar_count of 3000 means that the IO_ADVISE applies to these byte ranges of the dense DS file:

- 500 to 999
- 1000 to 1999
- 2000 to 2999
- 3000 to 3499

I.e., the contiguous range 500 to 3499 as specified in IO_ADVISE.

It also applies to these byte ranges of the logical file:

```
- 10500 to 10999 (500 bytes)
- 20000 to 20999 (1000 bytes)
- 30000 to 30999 (1000 bytes)
- 40000 to 40499 (500 bytes)
(total            3000 bytes)
```

E.g., if NFL4_UFLG_DENSE is absent, the stripe unit is 250 bytes, the stripe count is 4, and the sparse DS file is serving iar_offset 0. Then a READ or WRITE to the DS for iar_offsets 0, 1000, 2000, and 3000, really mean iar_offsets 0, 1000, 2000, and 3000 in the logical file, keeping in mind that on the DS file,. byte ranges 250 to 999, 1250 to 1999, 2250 to 2999, and 3250 to 3999 are not accessible. Then an IO_ADVISE sent to the same DS with an iar_offset of 500, and a iar_count of 3000 means that the IO_ADVISE applies to these byte ranges of the logical file and the sparse DS file:

```
- 500 to 999 (500 bytes)   - no effect
- 1000 to 1249 (250 bytes) - effective
- 1250 to 1999 (750 bytes) - no effect
- 2000 to 2249 (250 bytes) - effective
- 2250 to 2999 (750 bytes) - no effect
- 3000 to 3249 (250 bytes) - effective
- 3250 to 3499 (250 bytes) - no effect
(subtotal      2250 bytes) - no effect
(subtotal       750 bytes) - effective
(grand total   3000 bytes) - no effect + effective
```

If neither of the flags NFL42_UFLG_IO_ADVISE_THRU_MDS and NFL4_UFLG_DENSE are set in the layout, then any IO_ADVISE request sent to the data server with a byte range that overlaps stripe unit that the data server does not serve MUST NOT result in the status NFS4ERR_PNFS_IO_HOLE. Instead, the response SHOULD be successful and if the server applies IO_ADVISE hints on any stripe units that

overlap with the specified range, those hints SHOULD be indicated in the response.

14.8. Changes to Operation 51: LAYOUTRETURN

14.8.1. Introduction

In the pNFS description provided in [RFC5661], the client is not capable to relay an error code from the DS to the MDS. In the specification of the Objects-Based Layout protocol [RFC5664], use is made of the opaque lrf_body field of the LAYOUTRETURN argument to do such a relaying of error codes. In this section, we define a new data structure to enable the passing of error codes back to the MDS and provide some guidelines on what both the client and MDS should expect in such circumstances.

There are two broad classes of errors, transient and persistent. The client SHOULD strive to only use this new mechanism to report persistent errors. It MUST be able to deal with transient issues by itself. Also, while the client might consider an issue to be persistent, it MUST be prepared for the MDS to consider such issues to be transient. A prime example of this is if the MDS fences off a client from either a stateid or a filehandle. The client will get an error from the DS and might relay either NFS4ERR_ACCESS or NFS4ERR_BAD_STATEID back to the MDS, with the belief that this is a hard error. If the MDS is informed by the client that there is an error, it can safely ignore that. For it, the mission is accomplished in that the client has returned a layout that the MDS had most likely recalled.

The client might also need to inform the MDS that it cannot reach one or more of the DSes. While the MDS can detect the connectivity of both of these paths:

- MDS to DS

- MDS to client

it cannot determine if the client and DS path is working. As with the case of the DS passing errors to the client, it must be prepared for the MDS to consider such outages as being transitory.

The existing LAYOUTRETURN operation is extended by introducing a new data structure to report errors, layoutreturn_device_error4. Also, layoutreturn_device_error4 is introduced to enable an array of errors to be reported.

14.8.2. ARGUMENT

The ARGUMENT specification of the LAYOUTRETURN operation in section 18.44.1 of [RFC5661] is augmented by the following XDR code [RFC4506]:

```
struct layoutreturn_device_error4 {
        deviceid4       lrde_deviceid;
        nfsstat4        lrde_status;
        nfs_opnum4      lrde_opnum;
};

struct layoutreturn_error_report4 {
        layoutreturn_device_error4      lrer_errors<>;
};
```

14.8.3. RESULT

The RESULT of the LAYOUTRETURN operation is unchanged; see section 18.44.2 of [RFC5661].

14.8.4. DESCRIPTION

The following text is added to the end of the LAYOUTRETURN operation DESCRIPTION in section 18.44.3 of [RFC5661].

When a client uses LAYOUTRETURN with a type of LAYOUTRETURN4_FILE, then if the lrf_body field is NULL, it indicates to the MDS that the client experienced no errors. If lrf_body is non-NULL, then the field references error information which is layout type specific. I.e., the Objects-Based Layout protocol can continue to utilize lrf_body as specified in [RFC5664]. For both Files-Based and Block-Based Layouts, the field references a layoutreturn_device_error4, which contains an array of layoutreturn_device_error4.

Each individual layoutreturn_device_error4 describes a single error associated with a DS, which is identified via lrde_deviceid. The operation which returned the error is identified via lrde_opnum. Finally the NFS error value (nfsstat4) encountered is provided via lrde_status and may consist of the following error codes:

NFS4ERR_NXIO: The client was unable to establish any communication with the DS.

NFS4ERR_*: The client was able to establish communication with the DS and is returning one of the allowed error codes for the operation denoted by lrde_opnum.

14.8.5. IMPLEMENTATION

The following text is added to the end of the LAYOUTRETURN operation IMPLEMENTATION in section 18.4.4 of [RFC5661].

Clients are expected to tolerate transient storage device errors, and hence clients SHOULD NOT use the LAYOUTRETURN error handling for device access problems that may be transient. The methods by which a client decides whether a device access problem is transient vs. persistent are implementation-specific, but may include retrying I/Os to a data server under appropriate conditions.

When an I/O fails to a storage device, the client SHOULD retry the failed I/O via the MDS. In this situation, before retrying the I/O, the client SHOULD return the layout, or the affected portion thereof, and SHOULD indicate which storage device or devices was problematic. The client needs to do this when the DS is being unresponsive in order to fence off any failed write attempts, and ensure that they do not end up overwriting any later data being written through the MDS. If the client does not do this, the MDS MAY issue a layout recall callback in order to perform the retried I/O.

The client needs to be cognizant that since this error handling is optional in the MDS, the MDS may silently ignore this functionality. Also, as the MDS may consider some issues the client reports to be expected (see Section 14.8.1), the client might find it difficult to detect a MDS which has not implemented error handling via LAYOUTRETURN.

If an MDS is aware that a storage device is proving problematic to a client, the MDS SHOULD NOT include that storage device in any pNFS layouts sent to that client. If the MDS is aware that a storage device is affecting many clients, then the MDS SHOULD NOT include that storage device in any pNFS layouts sent out. If a client asks for a new layout for the file from the MDS, it MUST be prepared for the MDS to return that storage device in the layout. The MDS might not have any choice in using the storage device, i.e., there might only be one possible layout for the system. Also, in the case of existing files, the MDS might have no choice in which storage devices to hand out to clients.

The MDS is not required to indefinitely retain per-client storage device error information. An MDS is also not required to automatically reinstate use of a previously problematic storage device; administrative intervention may be required instead.

14.9. Operation 65: READ_PLUS

14.9.1. ARGUMENT

```
struct READ_PLUS4args {
        /* CURRENT_FH: file */
        stateid4        rpa_stateid;
        offset4         rpa_offset;
        count4          rpa_count;
};
```

14.9.2. RESULT

```
struct data_info4 {
        offset4         di_offset;
        length4         di_length;
        bool            di_allocated;
};

struct data4 {
        offset4         d_offset;
        bool            d_allocated;
        opaque          d_data<>;
};
```

```
union read_plus_content switch (data_content4 rpc_content) {
case NFS4_CONTENT_DATA:
        data4           rpc_data;
case NFS4_CONTENT_APP_DATA_HOLE:
        app_data_hole4  rpc_adh;
case NFS4_CONTENT_HOLE:
        data_info4      rpc_hole;
default:
        void;
};

/*
 * Allow a return of an array of contents.
 */
struct read_plus_res4 {
        bool                    rpr_eof;
        read_plus_content       rpr_contents<>;
};

union READ_PLUS4res switch (nfsstat4 rp_status) {
case NFS4_OK:
        read_plus_res4  rp_resok4;
default:
        void;
};
```

14.9.3. DESCRIPTION

The READ_PLUS operation is based upon the NFSv4.1 READ operation (see Section 18.22 of [RFC5661]) and similarly reads data from the regular file identified by the current filehandle.

The client provides a rpa_offset of where the READ_PLUS is to start and a rpa_count of how many bytes are to be read. A rpa_offset of zero means to read data starting at the beginning of the file. If rpa_offset is greater than or equal to the size of the file, the status NFS4_OK is returned with di_length (the data length) set to zero and eof set to TRUE.

The READ_PLUS result is comprised of an array of rpr_contents, each of which describe a data_content4 type of data (Section 7.1.2). For NFSv4.2, the allowed values are data, ADH, and hole. A server is required to support the data type, but neither ADH nor hole. Both an ADH and a hole must be returned in its entirety - clients must be prepared to get more information than they requested. Both the start and the end of the hole may exceed what was requested. The array contents MUST be contiguous in the file.

If the data to be returned is comprised entirely of zeros, then the server may elect to return that data as a hole. The server differentiates this to the client by setting di_allocated to TRUE in this case. Note that in such a scenario, the server is not required to determine the full extent of the "hole" - it does not need to determine where the zeros start and end. If the server elects to return the hole as data, then it can set the d_allocted to FALSE in the rpc_data to indicate it is a hole.

The server may elect to return adjacent elements of the same type. For example, the guard pattern or block size of an ADH might change, which would require adjacent elements of type ADH. Likewise if the server has a range of data comprised entirely of zeros and then a hole, it might want to return two adjacent holes to the client.

If the client specifies a rpa_count value of zero, the READ_PLUS succeeds and returns zero bytes of data. In all situations, 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.

If the client specifies an rpa_offset and rpa_count value that is entirely contained within a hole of the file, then the di_offset and di_length returned must be for the entire hole. This result is considered valid until the file is changed (detected via the change attribute). The server MUST provide the same semantics for the hole as if the client read the region and received zeroes; the implied holes contents lifetime MUST be exactly the same as any other read data.

If the client specifies an rpa_offset and rpa_count value that begins in a non-hole of the file but extends into hole the server should return an array comprised of both data and a hole. The client MUST be prepared for the server to return a short read describing just the data. The client will then issue another READ_PLUS for the remaining bytes, which the server will respond with information about the hole in the file.

Except when special stateids are used, the stateid value for a READ_PLUS 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_PLUS operation, if rpa_offset + rpa_count is equal to the size of the file), or the READ_PLUS operation extends beyond the size of

the file (if rpa_offset + rpa_count is greater than the size of the file), eof is returned as TRUE; otherwise, it is FALSE. A successful READ_PLUS 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_PLUS with a stateid value of all bits equal to zero, the server MAY allow the READ_PLUS to be serviced subject to mandatory byte-range locks or the current share deny modes for the file. For a READ_PLUS with a stateid value of all bits equal to one, the server MAY allow READ_PLUS operations to bypass locking checks at the server.

On success, the current filehandle retains its value.

14.9.4. IMPLEMENTATION

In general, the IMPLEMENTATION notes for READ in Section 18.22.4 of [RFC5661] also apply to READ_PLUS. One delta is that when the owner has a locked byte range, the server MUST return an array of rpr_contents with values inside that range.

14.9.4.1. Additional pNFS Implementation Information

With pNFS, the semantics of using READ_PLUS remains the same. Any data server MAY return a hole or ADH result for a READ_PLUS request that it receives. When a data server chooses to return such a result, it has the option of returning information for the data stored on that data server (as defined by the data layout), but it MUST NOT return results for a byte range that includes data managed by another data server.

A data server should do its best to return as much information about a ADH as is feasible without having to contact the metadata server. If communication with the metadata server is required, then every attempt should be taken to minimize the number of requests.

If mandatory locking is enforced, then the data server must also ensure that to return only information that is within the owner's locked byte range.

14.9.5. READ_PLUS with Sparse Files Example

The following table describes a sparse file. For each byte range, the file contains either non-zero data or a hole. In addition, the server in this example uses a Hole Threshold of 32K.

Byte-Range	Contents
0-15999	Hole
16K-31999	Non-Zero
32K-255999	Hole
256K-287999	Non-Zero
288K-353999	Hole
354K-417999	Non-Zero

Table 5

Under the given circumstances, if a client was to read from the file with a max read size of 64K, the following will be the results for the given READ_PLUS calls. This assumes the client has already opened the file, acquired a valid stateid ('s' in the example), and just needs to issue READ_PLUS requests.

1. READ_PLUS(s, 0, 64K) --> NFS_OK, eof = false, <data[0,32K], hole[32K,224K]>. Since the first hole is less than the server's Hole Threshhold, the first 32K of the file is returned as data and the remaining 32K is returned as a hole which actually extends to 256K.

2. READ_PLUS(s, 32K, 64K) --> NFS_OK, eof = false, <hole[32K,224K]> The requested range was all zeros, and the current hole begins at offset 32K and is 224K in length. Note that the client should not have followed up the previous READ_PLUS request with this one as the hole information from the previous call extended past what the client was requesting.

3. READ_PLUS(s, 256K, 64K) --> NFS_OK, eof = false, <data[256K, 288K], hole[288K, 354K]>. Returns an array of the 32K data and the hole which extends to 354K.

4. READ_PLUS(s, 354K, 64K) --> NFS_OK, eof = true, <data[354K, 418K]>. Returns the final 64K of data and informs the client there is no more data in the file.

14.10. Operation 66: SEEK

SEEK is an operation that allows a client to determine the location of the next data_content4 in a file. It allows an implementation of the emerging extension to lseek(2) to allow clients to determine SEEK_HOLE and SEEK_DATA.

14.10.1. ARGUMENT

```
struct SEEK4args {
        /* CURRENT_FH: file */
        stateid4        sa_stateid;
        offset4         sa_offset;
        data_content4   sa_what;
};
```

14.10.2. RESULT

```
union seek_content switch (data_content4 content) {
case NFS4_CONTENT_DATA:
        data_info4      sc_data;
case NFS4_CONTENT_APP_DATA_HOLE:
        app_data_hole4  sc_adh;
case NFS4_CONTENT_HOLE:
        data_info4      sc_hole;
default:
        void;
};

struct seek_res4 {
        bool                    sr_eof;
        seek_content            sr_contents;
};

union SEEK4res switch (nfsstat4 status) {
case NFS4_OK:
        seek_res4       resok4;
default:
        void;
};
```

14.10.3. DESCRIPTION

From the given sa_offset, find the next data_content4 of type sa_what in the file. For either a hole or ADH, this must return the data_content4 in its entirety. For data, it must not return the actual data.

SEEK must follow the same rules for stateids as READ_PLUS (Section 14.9.3).

If the server could not find a corresponding sa_what, then the status would still be NFS4_OK, but sr_eof would be TRUE. The sr_contents would contain a zero-ed out content of the appropriate type.

14.11. Operation 64: WRITE_HOLE

14.11.1. ARGUMENT

```
struct data_info4 {
        offset4         di_offset;
        length4         di_length;
        bool            di_allocated;
};


struct WRITE_HOLE4args {
        /* CURRENT_FH: file */
        stateid4        wh_stateid;
        stable_how4     wh_stable;
        data_info4      wh_hole;
};
```

14.11.2. RESULT

```
struct write_response4 {
        stateid4        wr_callback_id<1>;
        length4         wr_count;
        stable_how4     wr_committed;
        verifier4       wr_writeverf;
};


union WRITE_HOLE4res switch (nfsstat4 wh_status) {
case NFS4_OK:
        write_response4         wh_resok4;
default:
        void;
};
```

14.11.3. DESCRIPTION

The WRITE_HOLE operation is an extension of the NFSv4.1 WRITE operation (see Section 18.2 of [RFC5661]) and writes holes to the regular file identified by the current filehandle. The server MAY write fewer bytes than requested by the client.

A successful WRITE_HOLE will construct a reply for wr_count, wr_committed, and wr_writeverf as per the NFSv4.1 WRITE operation results. If wr_callback_id is set, it indicates an asynchronous reply (see Section 14.11.3.2).

WRITE_HOLE has to support all of the errors which are returned by WRITE plus NFS4ERR_NOTSUPP, i.e., it is an OPTIONAL operation. If the client supports WRITE_HOLE, it MUST support CB_OFFLOAD.

14.11.3.1. Hole punching

Whenever a client wishes to zero the blocks backing a particular region in the file, it calls the WRITE_HOLE operation with the current filehandle set to the filehandle of the file in question, and the equivalent of start offset and length in bytes of the region set in wh_hole.di_offset and wh_hole.di_length respectively. If the wh_hole.di_allocated is set to TRUE, then the blocks will be zeroed and if it is set to FALSE, then they will be deallocated. All further reads to this region MUST return zeros until overwritten. The filehandle specified must be that of a regular file.

Situations may arise where di_offset and/or di_offset + di_length will not be aligned to a boundary for which the server does allocations/deallocations. For most file systems, this is the block size of the file system. In such a case, the server can deallocate as many bytes as it can in the region. The blocks that cannot be deallocated MUST be zeroed. Except for the block deallocation and maximum hole punching capability, a WRITE_HOLE operation is to be treated similar to a write of zeroes.

The server is not required to complete deallocating the blocks specified in the operation before returning. The server SHOULD return an asynchronous result if it can determine the operation will be long running (see Section 14.11.3.2).

If used to hole punch, WRITE_HOLE will result in the space_used attribute being decreased by the number of bytes that were deallocated. The space_freed attribute may or may not decrease, depending on the support and whether the blocks backing the specified range were shared or not. The size attribute will remain unchanged.

The WRITE_HOLE operation MUST NOT change the space reservation guarantee of the file. While the server can deallocate the blocks specified by di_offset and di_length, future writes to this region MUST NOT fail with NFSERR_NOSPC.

14.11.3.2. Asynchronous Transactions

Hole punching may lead to server determining to service the operation asynchronously. If it decides to do so, it sets the stateid in wr_callback_id to be that of the wh_stateid. If it does not set the wr_callback_id, then the result is synchronous.

When the client determines that the reply will be given asynchronously, it should not assume anything about the contents of what it wrote until it is informed by the server that the operation is complete. It can use OFFLOAD_STATUS (Section 14.5) to monitor the operation and OFFLOAD_ABORT (Section 14.2) to cancel the operation. An example of a asynchronous WRITE_HOLE is shown in Figure 6. Note that as with the COPY operation, WRITE_HOLE must provide a stateid for tracking the asynchronous operation.

```
  Client                                  Server
     +                                      +
     |                                      |
     |--- OPEN ---------------------------->| Client opens
     |<------------------------------------/| the file
     |                                      |
     |--- WRITE_HOLE ---------------------->| Client punches
     |<------------------------------------/| a hole
     |                                      |
     |                                      |
     |--- OFFLOAD_STATUS ------------------>| Client may poll
     |<------------------------------------/| for status
     |                                      |
     |                  .                   | Multiple OFFLOAD_STATUS
     |                  .                   | operations may be sent.
     |                  .                   |
     |                                      |
     |<-- CB_OFFLOAD -----------------------| Server reports results
     |\------------------------------------>|
     |                                      |
     |--- CLOSE --------------------------->| Client closes
     |<------------------------------------/| the file
     |                                      |
     |                                      |
```

Figure 6: An asynchronous WRITE_HOLE.

When CB_OFFLOAD informs the client of the successful WRITE_HOLE, the write_response4 embedded in the operation will provide the necessary information that a synchronous WRITE_HOLE would have provided.

Regardless of whether the operation is asynchronous or synchronous, it MUST still support the COMMIT operation semantics as outlined in Section 18.3 of [RFC5661]. I.e., COMMIT works on one or more WRITE operations and the WRITE_HOLE operation can appear as several WRITE operations to the server. The client can use locking operations to control the behavior on the server with respect to long running asynchronous write operations.

14.12. Operation 68: WRITE_SAME

14.12.1. ARGUMENT

```
struct data_info4 {
        offset4         di_offset;
        length4         di_length;
        bool            di_allocated;
};


struct WRITE_SAME4args {
        /* CURRENT_FH: file */
        stateid4        ws_stateid;
        stable_how4     ws_stable;
        app_data_hole4  ws_adh;
};
```

14.12.2. RESULT

```
struct write_response4 {
        stateid4        wr_callback_id<1>;
        length4         wr_count;
        stable_how4     wr_committed;
        verifier4       wr_writeverf;
};
```

```
union WRITE_SAME4res switch (nfsstat4 ws_status) {
case NFS4_OK:
        write_response4         ws_resok4;
default:
        void;
};
```

14.12.3. DESCRIPTION

The WRITE_SAME operation is an extension of the NFSv4.1 WRITE operation (see Section 18.2 of [RFC5661]) and writes data to the regular file identified by the current filehandle. The server MAY write fewer bytes than requested by the client.

The WRITE_SAME argument is comprised of an array of rpr_contents, each of which describe a data_content4 type of data (Section 7.1.2). For NFSv4.2, the allowed values are data, ADH, and hole. The array contents MUST be contiguous in the file. A successful WRITE_SAME will construct a reply for wr_count, wr_committed, and wr_writeverf as per the NFSv4.1 WRITE operation results. If wr_callback_id is set, it indicates an asynchronous reply (see Section 14.12.3.2).

WRITE_SAME has to support all of the errors which are returned by WRITE plus NFS4ERR_NOTSUPP, i.e., it is an OPTIONAL operation. If the client supports WRITE_SAME, it MUST support CB_OFFLOAD.

14.12.3.1. ADHs

If the server supports ADHs, then it MUST support the WRITE_SAME operation. The server has no concept of the structure imposed by the application. It is only when the application writes to a section of the file does order get imposed. In order to detect corruption even before the application utilizes the file, the application will want to initialize a range of ADHs using WRITE_SAME.

For ADHs, when the client invokes the WRITE_SAME operation, it has two desired results:

1. The structure described by the app_data_block4 be imposed on the file.

2. The contents described by the app_data_block4 be sparse.

If the server supports the WRITE_SAME operation, it still might not support sparse files. So if it receives the WRITE_SAME operation, then it MUST populate the contents of the file with the initialized

ADHs. The server SHOULD return an asynchronous result if it can determine the operation will be long running (see Section 14.12.3.2).

If the data was already initialized, there are two interesting scenarios:

1. The data blocks are allocated.

2. Initializing in the middle of an existing ADH.

If the data blocks were already allocated, then the WRITE_SAME is a hole punch operation. If WRITE_SAME supports sparse files, then the data blocks are to be deallocated. If not, then the data blocks are to be rewritten in the indicated ADH format.

Since the server has no knowledge of ADHs, it should not report misaligned creation of ADHs. Even while it can detect them, it cannot disallow them, as the application might be in the process of changing the size of the ADHs. Thus the server must be prepared to handle an WRITE_SAME into an existing ADH.

This document does not mandate the manner in which the server stores ADHs sparsely for a file. However, if an WRITE_SAME arrives that will force a new ADH to start inside an existing ADH then the server will have three ADHs instead of two. It will have one up to the new one for the WRITE_SAME, one for the WRITE_SAME, and one for after the WRITE_SAME. Note that depending on server specific policies for block allocation, there may also be some physical blocks allocated to align the boundaries.

14.12.3.2. Asynchronous Transactions

ADH initialization may lead to server determining to service the operation asynchronously. If it decides to do so, it sets the stateid in wr_callback_id to be that of the ws_stateid. If it does not set the wr_callback_id, then the result is synchronous.

When the client determines that the reply will be given asynchronously, it should not assume anything about the contents of what it wrote until it is informed by the server that the operation is complete. It can use OFFLOAD_STATUS (Section 14.5) to monitor the operation and OFFLOAD_ABORT (Section 14.2) to cancel the operation. An example of a asynchronous WRITE_SAME is shown in Figure 7. Note that as with the COPY operation, WRITE_SAME must provide a stateid for tracking the asynchronous operation.

```
     Client                                  Server
        +                                       +
        |                                       |
        |--- OPEN ---------------------------->| Client opens
        |<------------------------------------/| the file
        |                                       |
        |--- WRITE_SAME ---------------------->| Client initializes
        |<------------------------------------/| an ADH
        |                                       |
        |                                       |
        |--- OFFLOAD_STATUS ------------------>| Client may poll
        |<------------------------------------/| for status
        |                                       |
        |                   .                   | Multiple OFFLOAD_STATUS
        |                   .                   | operations may be sent.
        |                   .                   |
        |                                       |
        |<-- CB_OFFLOAD -----------------------| Server reports results
        |\------------------------------------>|
        |                                       |
        |--- CLOSE --------------------------->| Client closes
        |<------------------------------------/| the file
        |                                       |
        |                                       |
```

Figure 7: An asynchronous WRITE_SAME.

When CB_OFFLOAD informs the client of the successful WRITE_SAME, the write_response4 embedded in the operation will provide the necessary information that a synchronous WRITE_SAME would have provided.

Regardless of whether the operation is asynchronous or synchronous, it MUST still support the COMMIT operation semantics as outlined in Section 18.3 of [RFC5661]. I.e., COMMIT works on one or more WRITE operations and the WRITE_SAME operation can appear as several WRITE operations to the server. The client can use locking operations to control the behavior on the server with respect to long running asynchronous write operations.

15. NFSv4.2 Callback Operations

15.1. Operation 15: CB_OFFLOAD - Report results of an asynchronous operation

15.1.1. ARGUMENT

```
struct write_response4 {
        stateid4       wr_callback_id<1>;
        length4        wr_count;
        stable_how4    wr_committed;
        verifier4      wr_writeverf;
};

union offload_info4 switch (nfsstat4 coa_status) {
case NFS4_OK:
        write_response4 coa_resok4;
default:
        length4        coa_bytes_copied;
};

struct CB_OFFLOAD4args {
        nfs_fh4        coa_fh;
        stateid4       coa_stateid;
        offload_info4  coa_offload_info;
};
```

15.1.2. RESULT

```
struct CB_OFFLOAD4res {
        nfsstat4       cor_status;
};
```

15.1.3. DESCRIPTION

CB_OFFLOAD is used to report to the client the results of an asynchronous operation, e.g., Server-side Copy or a hole punch. The coa_fh and coa_stateid identify the transaction and the coa_status indicates success or failure. The coa_resok4.wr_callback_id MUST NOT be set. If the transaction failed, then the coa_bytes_copied contains the number of bytes copied before the failure occurred. The coa_bytes_copied value indicates the number of bytes copied but not which specific bytes have been copied.

If the client supports either

1. the COPY operation

2. either the WRITE_HOLE or WRITE_SAME operations

then the client is REQUIRED to support the CB_OFFLOAD operation.

There is a potential race between the reply to the original transaction on the forechannel and the CB_OFFLOAD callback on the backchannel. Sections 2.10.6.3 and 20.9.3 of [RFC5661] describe how to handle this type of issue.

15.1.3.1. Server-side Copy

CB_OFFLOAD is used for both intra- and inter-server asynchronous copies. This operation is sent by the destination server to the client in a CB_COMPOUND request. Upon success, the coa_resok4.wr_count presents the total number of bytes copied.

15.1.3.2. WRITE_HOLE and WRITE_SAME

CB_OFFLOAD is used to report the completion of either a hole punch or an ADH initialization. Upon success, the coa_resok4 will contain the same information that the corresponding synchronous WRITE_HOLE or WRITE_SAME would have returned.

16. IANA Considerations

This section uses terms that are defined in [RFC5226].

17. References

17.1. Normative References

[NFSv42xdr]
Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 External Data Representation Standard (XDR) Description", April 2014.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

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

[RFC5664] Halevy, B., Welch, B., and J. Zelenka, "Object-Based Parallel NFS (pNFS) Operations", RFC 5664, January 2010.

[posix_fadvise]
The Open Group, "Section 'posix_fadvise()' of System Interfaces of The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition", 2004.

[rpcsec_gssv3]
Adamson, W. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", October 2013.

17.2. Informative References

[Ashdown08]
Ashdown, L., "Chapter 15, Validating Database Files and Backups, of Oracle Database Backup and Recovery User's Guide 11g Release 1 (11.1)", August 2008.

[Baira08] Bairavasundaram, L., Goodson, G., Schroeder, B., Arpaci-Dusseau, A., and R. Arpaci-Dusseau, "An Analysis of Data Corruption in the Storage Stack", Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08) , 2008.

[FEDFS-ADMIN]
Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M. Naik, "Administration Protocol for Federated Filesystems", draft-ietf-nfsv4-federated-fs-admin (Work In Progress), 2010.

[FEDFS-NSDB]
Lentini, J., Everhart, C., Ellard, D., Tewari, R., and M. Naik, "NSDB Protocol for Federated Filesystems", draft-ietf-nfsv4-federated-fs-protocol (Work In Progress), 2010.

[Haynes13]
Haynes, T., "Requirements for Labeled NFS", draft-ietf-nfsv4-labreqs-04 (work in progress), 2013.

[I-D.ietf-nfsv4-rfc3530bis]
Haynes, T. and D. Noveck, "Network File System (NFS) version 4 Protocol", draft-ietf-nfsv4-rfc3530bis-25 (Work In Progress), February 2013.

[IESG08] ISEG, "IESG Processing of RFC Errata for the IETF Stream", 2008.

[MLS] "Section 46.6. Multi-Level Security (MLS) of Deployment Guide: Deployment, configuration and administration of Red Hat Enterprise Linux 5, Edition 6", 2011.

[McDougall07]
McDougall, R. and J. Mauro, "Section 11.4.3, Detecting Memory Corruption of Solaris Internals", 2007.

[Quigley11]
Quigley, D. and J. Lu, "Registry Specification for MAC Security Label Formats", draft-quigley-label-format-registry (work in progress), 2011.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

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

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[Strohm11]
Strohm, R., "Chapter 2, Data Blocks, Extents, and Segments, of Oracle Database Concepts 11g Release 1 (11.1)", January 2011.

Appendix A. Acknowledgments

Tom Haynes would like to thank NetApp, Inc. for its funding of his time on this project.

For the pNFS Access Permissions Check, the original draft was by Sorin Faibish, David Black, Mike Eisler, and Jason Glasgow. The work was influenced by discussions with Benny Halevy and Bruce Fields. A review was done by Tom Haynes.

For the Sharing change attribute implementation details with NFSv4 clients, the original draft was by Trond Myklebust.

For the NFS Server-side Copy, the original draft was by James Lentini, Mike Eisler, Deepak Kenchammana, Anshul Madan, and Rahul Iyer. Tom Talpey co-authored an unpublished version of that document. It was also was reviewed by a number of individuals: Pranoop Erasani, Tom Haynes, Arthur Lent, Trond Myklebust, Dave Noveck, Theresa Lingutla-Raj, Manjunath Shankararao, Satyam Vaghani, and Nico Williams. Anna Schumaker's early prototyping experience helped us avoid some traps.

For the NFS space reservation operations, the original draft was by Mike Eisler, James Lentini, Manjunath Shankararao, and Rahul Iyer.

For the sparse file support, the original draft was by Dean Hildebrand and Marc Eshel. Valuable input and advice was received from Sorin Faibish, Bruce Fields, Benny Halevy, Trond Myklebust, and Richard Scheffenegger.

For the Application IO Hints, the original draft was by Dean Hildebrand, Mike Eisler, Trond Myklebust, and Sam Falkner. Some early reviewers included Benny Halevy and Pranoop Erasani.

For Labeled NFS, the original draft was by David Quigley, James Morris, Jarret Lu, and Tom Haynes. Peter Staubach, Trond Myklebust, Stephen Smalley, Sorin Faibish, Nico Williams, and David Black also contributed in the final push to get this accepted.

During the review process, Talia Reyes-Ortiz helped the sessions run smoothly. While many people contributed here and there, the core reviewers were Andy Adamson, Pranoop Erasani, Bruce Fields, Chuck Lever, Trond Myklebust, David Noveck, Peter Staubach, and Mike Kupfer.

Appendix B. RFC Editor Notes

[RFC Editor: please remove this section prior to publishing this document as an RFC]

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of NFSv42xdr with RFCxxxx where xxxx is the RFC number of the companion XDR document]

Author's Address

Thomas Haynes (editor)
PrimaryData, Inc.
4300 El Camino Real Ste 100
Los Altos, CA 94022
USA

Phone: +1 408 215 1519
Email: thomas.haynes@primarydata.com