NFSv4 T. Haynes
Internet-Draft Editor
Intended status: Standards Track January 04, 2012
Expires: July 7, 2012

NFS Version 4 Minor Version 2
draft-ietf-nfsv4-minorversion2-07.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, Space Reservations, and Support for Sparse Files.

Requirements Language

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

Status of this Memo

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

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

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

This Internet-Draft will expire on July 7, 2012.

Copyright Notice

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 [11] and the second minor version, NFSv4.1, is described in [2]. It follows the guidelines for minor versioning that are listed in Section 11 of [11].

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:

- describe the NFSv4.0 or NFSv4.1 protocols, except where needed to contrast with NFSv4.2.
- modify the specification of the NFSv4.0 or NFSv4.1 protocols.
- 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 [3].

1.3. NFSv4.2 Goals

[[Comment.1: This needs fleshing out! --TH]]

1.4. Overview of NFSv4.2 Features

[[Comment.2: This needs fleshing out! --TH]]

1.4.1. Sparse Files

Two new operations are defined to support the reading of sparse files (READ_PLUS) and the punching of holes to remove backing storage (INITIALIZE).

1.4.2. Application I/O Advise

We propose a new IO_ADVISE operation for NFSv4.2 that clients can use to communicate expected I/O behavior to the server. By communicating 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, servers can 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.

1.5. Differences from NFSv4.1

[[Comment.3: This needs fleshing out! --TH]]

2. NFS Server-side Copy

2.1. Introduction

This section describes a server-side copy feature for the NFS protocol.

The server-side copy feature provides a mechanism for the NFS client to perform a file copy on the server without the data being transmitted back and forth over the network.

Without this feature, an NFS client copies data from one location to another by reading the data from the server over the network, and then writing the data back over the network to the server. Using this server-side copy operation, the client is able to instruct the server to copy the data locally without the data being sent back and forth over the network unnecessarily.

In general, this feature is useful whenever data is copied from one location to another on the server. It is particularly useful when copying the contents of a file from a backup. Backup-versions of a file are copied for a number of reasons, including restoring and cloning data.

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

2.2. 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 [12] [13], the client can use the ONC RPC Federated Filesystem protocol [13] to perform the copy. Specifically the client can determine the source junction's attributes using the FEDFS_LOOKUP_FSN procedure and create a duplicate junction using the FEDFS_CREATE_JUNCTION procedure.

For the inter-server copy protocol, the operations are defined to be compatible with a server-to-server copy protocol in which the destination server reads the file data from the source server. This model in which the file data is pulled from the source by the destination has a number of advantages over a model in which the source pushes the file data to the destination. The advantages of the pull model include:

- The pull model only requires a remote server (i.e., the destination server) to be granted read access. A push model requires a remote server (i.e., the source server) to be granted write access, which is more privileged.

- The pull model allows the destination server to stop reading if it has run out of space. In a push model, the destination server must flow control the source server in this situation.

- The pull model allows the destination server to easily flow control the data stream by adjusting the size of its read

operations. In a push model, the destination server does not have this ability. The source server in a push model is capable of writing chunks larger than the destination server has requested in attributes and session parameters. In theory, the destination server could perform a "short" write in this situation, but this approach is known to behave poorly in practice.

The following operations are provided to support server-side copy:

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

COPY_REVOKE: Also for inter-server copies, the client sends this operation to the source server to revoke permission to copy a file for the given user.

COPY: Used by the client to request a file copy.

COPY_ABORT: Used by the client to abort an asynchronous file copy.

COPY_STATUS: Used by the client to poll the status of an asynchronous file copy.

CB_COPY: Used by the destination server to report the results of an asynchronous file copy to the client.

These operations are described in detail in Section 2.3. This section provides an overview of how these operations are used to perform server-side copies.

2.2.1. Intra-Server Copy

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

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

Figure 1: A synchronous intra-server copy.

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

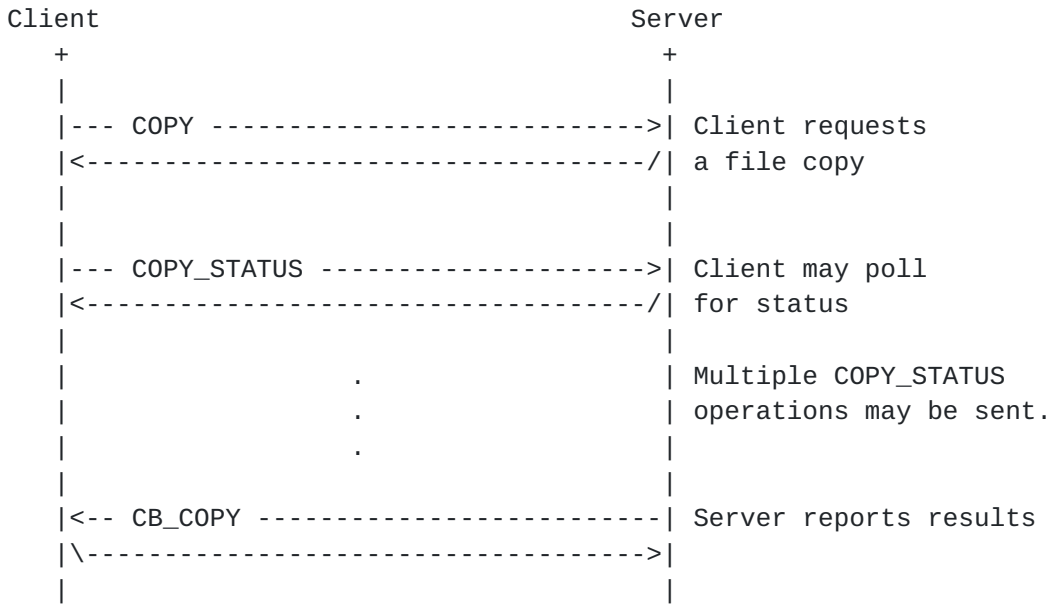

Figure 2: An asynchronous intra-server copy.

2.2.2. Inter-Server Copy

A copy may also be performed between two servers. The copy protocol is designed to accommodate a variety of network topologies. As shown in Figure 3, the client and servers may be connected by multiple networks. In particular, the servers may be connected by a specialized, high speed network (network 192.168.33.0/24 in the diagram) that does not include the client. The protocol allows the

client to setup the copy between the servers (over network 10.11.78.0/24 in the diagram) and for the servers to communicate on the high speed network if they choose to do so.

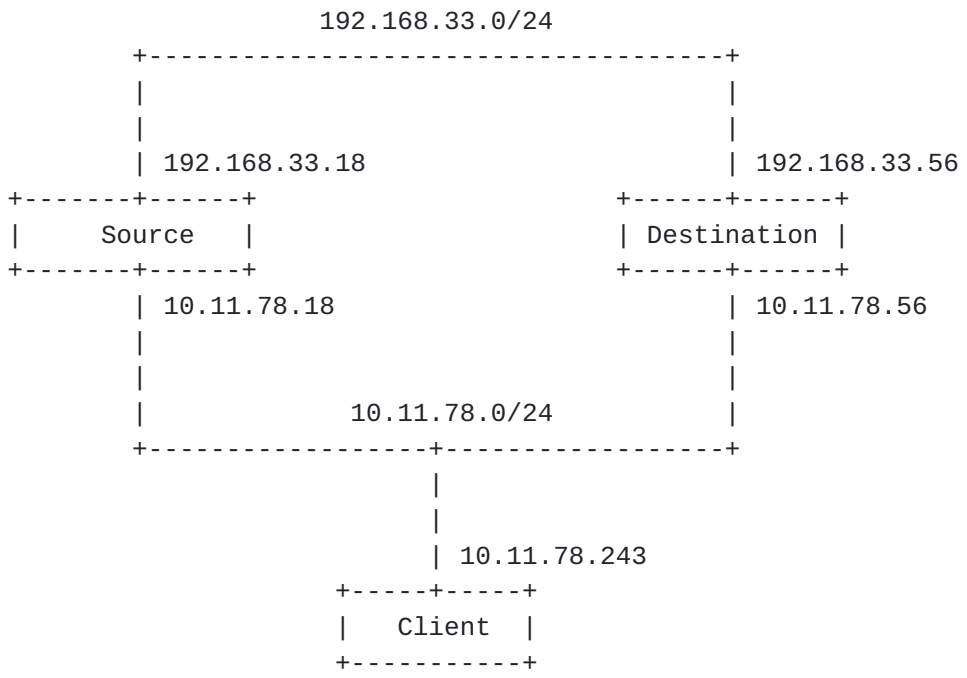

Figure 3: An example inter-server network topology.

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.

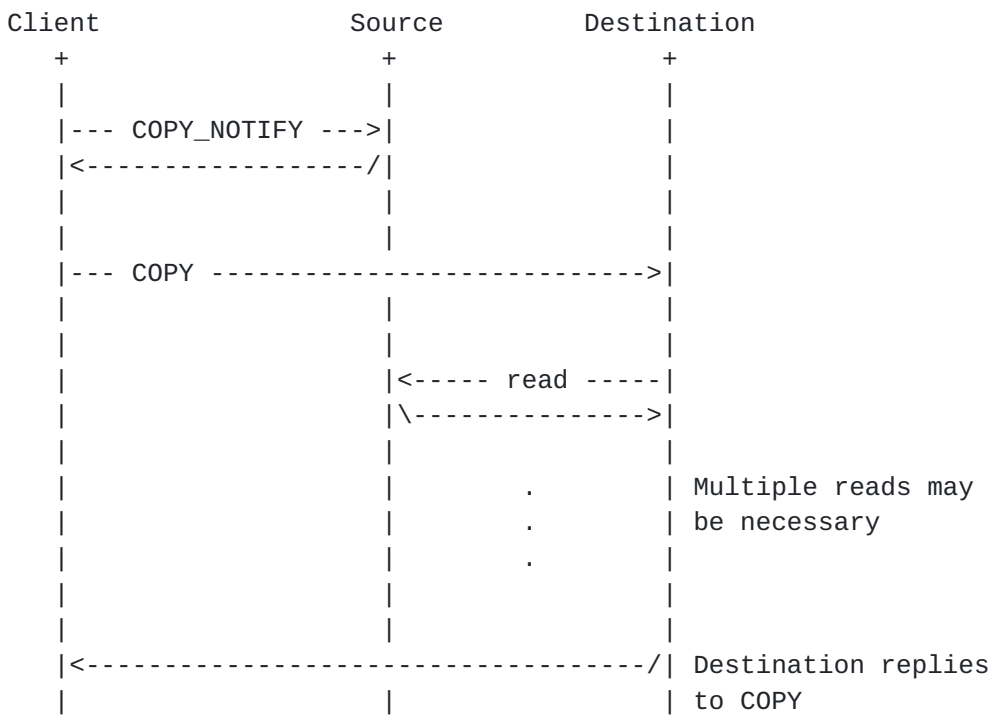

Figure 4: A synchronous inter-server copy.

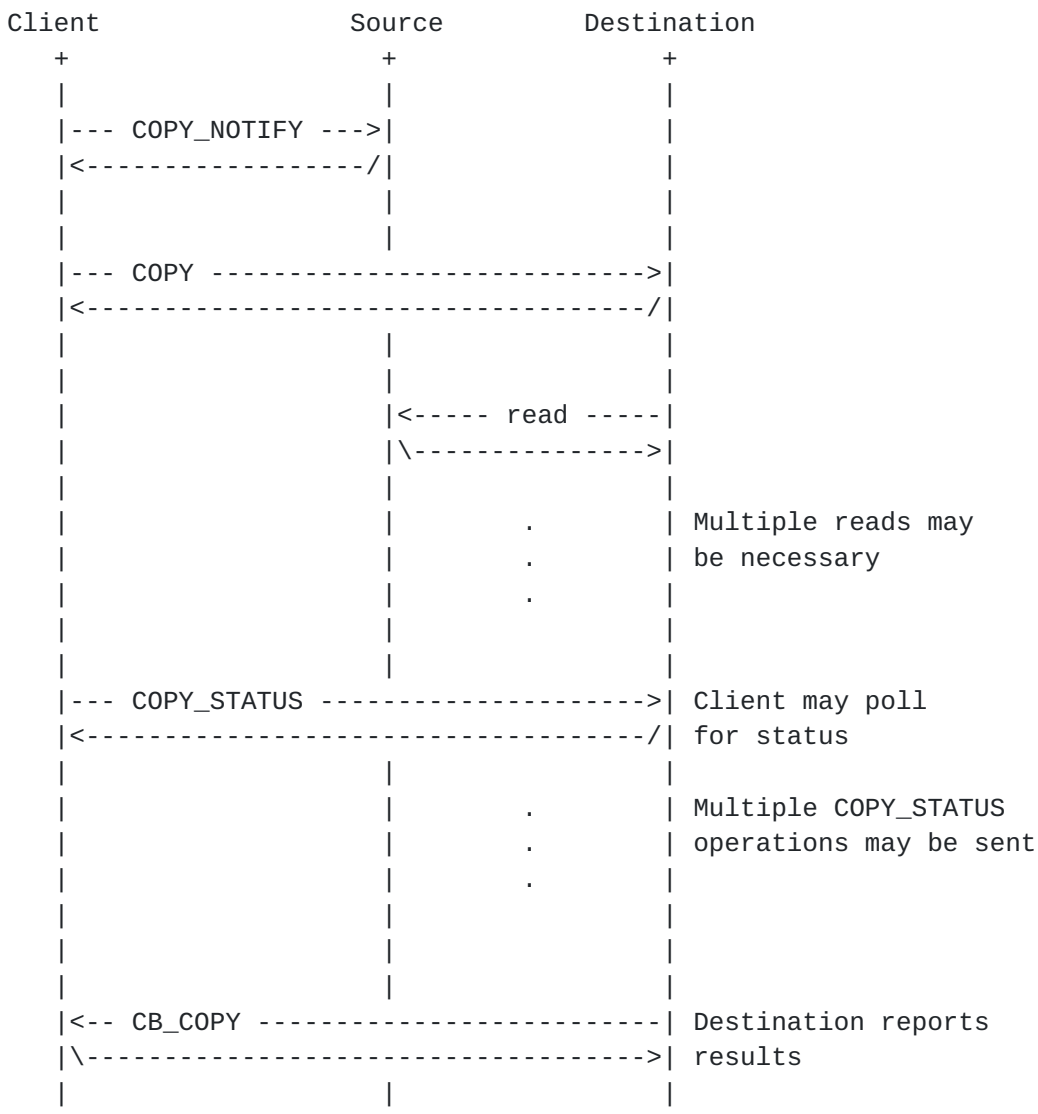

Figure 5: An asynchronous inter-server copy.

2.2.3. Server-to-Server Copy Protocol

During an inter-server copy, the destination server reads the file data from the source server. The source server and destination server are not required to use a specific protocol to transfer the file data. The choice of what protocol to use is ultimately the destination server's decision.

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

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

operations to read data from the source server. Specifically, the destination server may use the NFSv4.x OPEN operation's CLAIM_FH facility to open the file being copied and obtain an open stateid. Using the stateid, the destination server may then use NFSv4.x READ operations to read the file.

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

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

In a heterogeneous environment, using a protocol other than NFSv4.x (e.g,. HTTP [14] or FTP [15]) 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 2.4.1.2.4 and Section 2.4.1.4.

2.3. Operations

In the sections that follow, several operations are defined that together provide the server-side copy feature. These operations are intended to be OPTIONAL operations as defined in section 17 of [2]. The COPY_NOTIFY, COPY_REVOKE, COPY, COPY_ABORT, and COPY_STATUS operations are designed to be sent within an NFSv4 COMPOUND procedure. The CB_COPY operation is designed to be sent within an NFSv4 CB_COMPOUND procedure.

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

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

2.3.1. 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 [4] 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 [2].

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.

2.3.2. 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, COPY_ABORT, COPY_STATUS, and CB_COPY operations.

Section 8.2.4 of [2] specifies that stateids are valid until either (A) the client or server restart or (B) the client returns the resource.

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

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

2.4. Security Considerations

The security considerations pertaining to NFSv4 [11] apply to this document.

The standard security mechanisms provide by NFSv4 [11] may be used to secure the protocol described in this document.

NFSv4 clients and servers supporting the the inter-server copy operations described in this document are REQUIRED to implement [5], including the RPCSEC_GSSv3 privileges copy_from_auth and copy_to_auth. If the server-to-server copy protocol is ONC RPC based, the servers are also REQUIRED to implement the RPCSEC_GSSv3 privilege copy_confirm_auth. These requirements to implement are not requirements to use. NFSv4 clients and servers are RECOMMENDED to use [5] to secure server-side copy operations.

2.4.1. Inter-Server Copy Security

2.4.1.1. Requirements for Secure Inter-Server Copy

Inter-server copy is driven by several requirements:

- 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.

- 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.

2.4.1.2. Inter-Server Copy with RPCSEC_GSSv3

When the client sends a COPY_NOTIFY to the source server to expect the destination to attempt to copy data from the source server, it is expected that this copy is being done on behalf of the principal (called the "user principal") that sent the RPC request that encloses the COMPOUND procedure that contains the COPY_NOTIFY operation. The user principal is identified by the RPC credentials. A mechanism that allows the user principal to authorize the destination server to perform the copy in a manner that lets the source server properly authenticate the destination's copy, and without allowing the destination to exceed its authorization is necessary.

An approach that sends delegated credentials of the client's user principal to the destination server is not used for the following reasons. If the client's user delegated its credentials, the destination would authenticate as the user principal. If the

destination were using the NFSv4 protocol to perform the copy, then the source server would authenticate the destination server as the user principal, and the file copy would securely proceed. However, this approach would allow the destination server to copy other files. The user principal would have to trust the destination server to not do so. This is counter to the requirements, and therefore is not considered. Instead an approach using RPCSEC_GSSv3 [5] privileges is proposed.

One of the stated applications of the proposed RPCSEC_GSSv3 protocol is compound client host and user authentication [+ privilege assertion]. For inter-server file copy, we require compound NFS server host and user authentication [+ privilege assertion]. The distinction between the two is one without meaning.

RPCSEC_GSSv3 introduces the notion of privileges. We define three privileges:

copy_from_auth: A user principal is authorizing a source principal ("nfs@<source>") to allow a destination principal ("nfs@<destination>") to copy a file from the source to the destination. This privilege is established on the source server before the user principal sends a COPY_NOTIFY operation to the source server.

```
struct copy_from_auth_priv {
        secret4             cfap_shared_secret;
        netloc4             cfap_destination;
        /* the NFSv4 user name that the user principal maps to */
        utf8str_mixed       cfap_username;
        /* equal to seq_num of rpc_gss_cred_vers_3_t */
        unsigned int        cfap_seq_num;
};
```

cap_shared_secret is a secret value the user principal generates.

copy_to_auth: A user principal is authorizing a destination principal ("nfs@<destination>") to allow it to copy a file from the source to the destination. This privilege is established on the destination server before the user principal sends a COPY operation to the destination server.

```
struct copy_to_auth_priv {
        /* equal to cfap_shared_secret */
        secret4             ctap_shared_secret;
        netloc4             ctap_source;
        /* the NFSv4 user name that the user principal maps to */
        utf8str_mixed       ctap_username;
        /* equal to seq_num of rpc_gss_cred_vers_3_t */
        unsigned int        ctap_seq_num;
};
```

ctap_shared_secret is a secret value the user principal generated and was used to establish the copy_from_auth privilege with the source principal.

copy_confirm_auth: A destination principal is confirming with the source principal that it is authorized to copy data from the source on behalf of the user principal. When the inter-server copy protocol is NFSv4, or for that matter, any protocol capable of being secured via RPCSEC_GSSv3 (i.e., any ONC RPC protocol), this privilege is established before the file is copied from the source to the destination.

```
struct copy_confirm_auth_priv {
        /* equal to GSS_GetMIC() of cfap_shared_secret */
        opaque              ccap_shared_secret_mic<>;
        /* the NFSv4 user name that the user principal maps to */
        utf8str_mixed       ccap_username;
        /* equal to seq_num of rpc_gss_cred_vers_3_t */
        unsigned int        ccap_seq_num;
};
```

2.4.1.2.1. Establishing a Security Context

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

- The user principal generates a secret it will share with the two servers. This shared secret will be placed in the cfap_shared_secret and ctap_shared_secret fields of the appropriate privilege data types, copy_from_auth_priv and copy_to_auth_priv.

- An instance of copy_from_auth_priv is filled in with the shared secret, the destination server, and the NFSv4 user id of the user principal. It will be sent with an RPCSEC_GSS3_CREATE procedure,

and so cfap_seq_num is set to the seq_num of the credential of the RPCSEC_GSS3_CREATE procedure. Because cfap_shared_secret is a secret, after XDR encoding copy_from_auth_priv, GSS_Wrap() (with privacy) is invoked on copy_from_auth_priv. The RPCSEC_GSS3_CREATE procedure's arguments are:

```
struct {
   rpc_gss3_gss_binding    *compound_binding;
   rpc_gss3_chan_binding   *chan_binding_mic;
   rpc_gss3_assertion      assertions<>;
   rpc_gss3_extension      extensions<>;
} rpc_gss3_create_args;
```

The string "copy_from_auth" is placed in assertions[0].privs. The output of GSS_Wrap() is placed in extensions[0].data. The field extensions[0].critical is set to TRUE. The source server calls GSS_Unwrap() on the privilege, and verifies that the seq_num matches the credential. It then verifies that the NFSv4 user id being asserted matches the source server's mapping of the user principal. If it does, the privilege is established on the source server as: <"copy_from_auth", user id, destination>. The successful reply to RPCSEC_GSS3_CREATE has:

```
struct {
   opaque                  handle<>;
   rpc_gss3_chan_binding   *chan_binding_mic;
   rpc_gss3_assertion      granted_assertions<>;
   rpc_gss3_assertion      server_assertions<>;
   rpc_gss3_extension      extensions<>;
} rpc_gss3_create_res;
```

The field "handle" is the RPCSEC_GSSv3 handle that the client will use on COPY_NOTIFY requests involving the source and destination server. granted_assertions[0].privs will be equal to "copy_from_auth". The server will return a GSS_Wrap() of copy_to_auth_priv.

- An instance of copy_to_auth_priv is filled in with the shared secret, the source server, and the NFSv4 user id. It will be sent with an RPCSEC_GSS3_CREATE procedure, and so ctap_seq_num is set to the seq_num of the credential of the RPCSEC_GSS3_CREATE procedure. Because ctap_shared_secret is a secret, after XDR encoding copy_to_auth_priv, GSS_Wrap() is invoked on copy_to_auth_priv. The RPCSEC_GSS3_CREATE procedure's arguments

are:

```
struct {
   rpc_gss3_gss_binding    *compound_binding;
   rpc_gss3_chan_binding   *chan_binding_mic;
   rpc_gss3_assertion      assertions<>;
   rpc_gss3_extension      extensions<>;
} rpc_gss3_create_args;
```

The string "copy_to_auth" is placed in assertions[0].privs. The output of GSS_Wrap() is placed in extensions[0].data. The field extensions[0].critical is set to TRUE. After unwrapping, verifying the seq_num, and the user principal to NFSv4 user ID mapping, the destination establishes a privilege of <"copy_to_auth", user id, source>. The successful reply to RPCSEC_GSS3_CREATE has:

```
struct {
   opaque                  handle<>;
   rpc_gss3_chan_binding   *chan_binding_mic;
   rpc_gss3_assertion      granted_assertions<>;
   rpc_gss3_assertion      server_assertions<>;
   rpc_gss3_extension      extensions<>;
} rpc_gss3_create_res;
```

The field "handle" is the RPCSEC_GSSv3 handle that the client will use on COPY requests involving the source and destination server. The field granted_assertions[0].privs will be equal to "copy_to_auth". The server will return a GSS_Wrap() of copy_to_auth_priv.

2.4.1.2.2. 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 the name of the destination server specified in copy_from_auth_priv. Otherwise, COPY_NOTIFY will fail with NFS4ERR_ACCESS. The source server verifies that the privilege <"copy_from_auth", user id, destination> exists, and annotates it with the source filehandle, if the user principal has read access to the source file, and if administrative policies give the user principal and the NFS client read access to the source file (i.e., if the ACCESS operation would grant read access). Otherwise, COPY_NOTIFY will fail with NFS4ERR_ACCESS.

When the client sends a COPY request to the destination server, it uses the privileged "copy_to_auth" RPCSEC_GSSv3 handle. ca_source_server in COPY MUST be the same as the name of the source server specified in copy_to_auth_priv. Otherwise, COPY will fail with NFS4ERR_ACCESS. The destination server verifies that the privilege <"copy_to_auth", user id, source> exists, and annotates it with the source and destination filehandles. If the client has failed to establish the "copy_to_auth" policy it will reject the request with NFS4ERR_PARTNER_NO_AUTH.

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

2.4.1.2.3. 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, using nfs@<destination> as the initiator principal, and nfs@<source> as the target principal.

The value of the field ccap_shared_secret_mic is a GSS_VerifyMIC() of the shared secret passed in the copy_to_auth privilege. The field ccap_username is the mapping of the user principal to an NFSv4 user name ("user"@"domain" form), and MUST be the same as ctap_username and cfap_username. The field ccap_seq_num is the seq_num of the RPCSEC_GSSv3 credential used for the RPCSEC_GSS3_CREATE procedure the destination will send to the source server to establish the privilege.

The source server verifies the privilege, and establishes a <"copy_confirm_auth", user id, destination> privilege. If the source server fails to verify the privilege, the COPY operation will be rejected with NFS4ERR_PARTNER_NO_AUTH. All subsequent ONC RPC requests sent from the destination to copy data from the source to the destination will use the RPCSEC_GSSv3 handle returned by the source's RPCSEC_GSS3_CREATE response.

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

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

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

If the attempt to establish a "copy_confirm_auth" privilege fails, then when the user principal sends a COPY request to destination, the destination server will reject it with NFS4ERR_PARTNER_NO_AUTH.

2.4.1.2.4. Securing Non ONC RPC Server-to-Server Copy Protocols

If the destination won't be using ONC RPC to copy the data, then the source and destination are using an unspecified copy protocol. The destination could use the shared secret and the NFSv4 user id to prove to the source server that the user principal has authorized the copy.

For protocols that authenticate user names with passwords (e.g., HTTP [14] and FTP [15]), the nfsv4 user id could be used as the user name, and an ASCII hexadecimal representation of the RPCSEC_GSSv3 shared secret could be used as the user password or as input into non-password authentication methods like CHAP [16].

2.4.1.3. Inter-Server Copy via ONC RPC but without RPCSEC_GSSv3

ONC RPC security flavors other than RPCSEC_GSSv3 MAY be used with the server-side copy offload operations described in this document. In particular, host-based ONC RPC security flavors such as AUTH_NONE and AUTH_SYS MAY be used. If a host-based security flavor is used, a minimal level of protection for the server-to-server copy protocol is possible.

In the absence of strong security mechanisms such as RPCSEC_GSSv3, the challenge is how the source server and destination server identify themselves to each other, especially in the presence of multi-homed source and destination servers. In a multi-homed environment, the destination server might not contact the source server from the same network address specified by the client in the COPY_NOTIFY. This can be overcome using the procedure described below.

When the client sends the source server the COPY_NOTIFY operation, the source server may reply to the client with a list of target addresses, names, and/or URLs and assign them to the unique 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 10.11.78.56 with the URLs:

```
nfs://10.11.78.18//_COPY/FvhH1OKbu8VrxvV1erdjvR7N/10.11.78.56/_FH/
0x12345

nfs://192.168.33.18//_COPY/FvhH1OKbu8VrxvV1erdjvR7N/10.11.78.56/
_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 192.168.33.0/24 network is a high speed network and the destination server decides to transfer the file over this network. If the destination contacts the source server from 192.168.33.56 over this network using NFSv4.1, it does the following:

```
COMPOUND { PUTROOTFH, LOOKUP "_COPY" ; LOOKUP
   "FvhH1OKbu8VrxvV1erdjvR7N" ; LOOKUP "10.11.78.56"; 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.

2.4.1.4. Inter-Server Copy without ONC RPC and RPCSEC_GSSv3

The same techniques as Section 2.4.1.3, using unique URLs for each destination server, can be used for other protocols (e.g., HTTP [14] and FTP [15]) as well.

3. Sparse Files

3.1. 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.

This section introduces a new operation READ_PLUS (Section 12.10) which supports all the features of READ but includes an extension to support sparse pattern files. 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.

3.2. 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.

3.3. Determining the next hole/data

Solaris and ZFS support an extension to lseek(2) that allows applications to discover holes in a file. The values, SEEK_HOLE and SEEK_DATA, allow clients to seek to the next hole or beginning of data, respectively.

4. Space Reservation

4.1. Introduction

This section describes a set of operations that allow applications such as hypervisors to reserve space for a file, report the amount of actual disk space a file occupies and freeup the backing space of a file when it is not required. In virtualized environments, virtual disk files are often stored on NFS mounted volumes. Since virtual disk files represent the hard disks of virtual machines, hypervisors often have to guarantee certain properties for the file.

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

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

If the space_reserved attribute is set on a file, it is guaranteed that writes that do not grow the file will not fail with NFSERR_NOSPC.

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

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

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

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

space_reserved This attribute specifies whether the blocks backing the file have been preallocated.

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

INITIALIZED This operation zeroes and/or deallocates 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, is helpful in solving this problem. space_freed is the number of blocks that are allocated to the given file that would be freed on its deletion. In the example, both A and B would report space_freed as 4 * BLOCK_SIZE and space_used as 10 * BLOCK_SIZE. If A is deleted, B will report space_freed as 10 * BLOCK_SIZE as the deletion of B would result in the deallocation of all 10 blocks.

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

5. Support for Application IO Hints

5.1. Introduction

Applications currently have several options for communicating I/O access patterns 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. Therefore, here we put forth a proposal for the NFSv4.2 protocol to allow applications to communicate their expected behavior to the server.

By communicating expected access pattern, e.g., sequential or random, and data re-use behavior, e.g., data range will be read multiple times and should be cached, the server will be able to better understand what optimizations it should implement for access to a file. For example, if a application indicates it will never read the data more than once, then the file system can avoid polluting the data cache and not cache the data.

The first application that can issue client I/O hints is the posix_fadvise operation. For example, on Linux, when an application uses posix_fadvise to specify a file will be read sequentially, Linux doubles the readahead buffer size.

Another instance where applications provide an indication of their desired I/O behavior is the use of direct I/O. By specifying direct I/O, clients will no longer cache data, but this information is not passed to the server, which will continue caching data.

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

This section adds a new IO_ADVISE operation 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.

5.2. POSIX Requirements

The first key requirement of the IO_ADVISE operation is to support the posix_fadvise function [6], which is supported in Linux and many other operating systems. Examples and guidance on how to use posix_fadvise to improve performance can be found here [17]. posix_fadvise is defined as follows,

```
int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

The posix_fadvise() function shall advise the implementation on the expected behavior of the application with respect to the data in the file associated with the open file descriptor, fd, starting at offset and continuing for len bytes. The specified range need not currently exist in the file. If len is zero, all data following offset is specified. The implementation may use this information to optimize handling of the specified data. The posix_fadvise() function shall have no effect on the semantics of other operations on the specified data, although it may affect the performance of other operations.

The advice to be applied to the data is specified by the advice parameter and may be one of the following values:

POSIX_FADV_NORMAL - Specifies that the application has no advice to give on its behavior with respect to the specified data. It is the default characteristic if no advice is given for an open file.

POSIX_FADV_SEQUENTIAL - Specifies that the application expects to access the specified data sequentially from lower offsets to higher offsets.

POSIX_FADV_RANDOM - Specifies that the application expects to access the specified data in a random order.

POSIX_FADV_WILLNEED - Specifies that the application expects to access the specified data in the near future.

POSIX_FADV_DONTNEED - Specifies that the application expects that it will not access the specified data in the near future.

POSIX_FADV_NOREUSE - Specifies that the application expects to access the specified data once and then not reuse it thereafter.

Upon successful completion, posix_fadvise() shall return zero; otherwise, an error number shall be returned to indicate the error.

5.3. Additional Requirements

Many use cases exist for sending application I/O hints to the server that cannot utilize the POSIX supported interface. This is because some applications may benefit from additional hints not specified by posix_fadvise, and some applications may not use POSIX altogether.

One use case is "Opportunistic Prefetch", which allows a stateid holder to tell the server that it is possible that it will access the specified data in the near future. This is similar to POSIX_FADV_WILLNEED, but the client is unsure it will in fact read the specified data, so the server should only prefetch the data if it can be done at a marginal cost. For example, when a server receives this hint, it could prefetch only the indirect blocks for a file instead of all the data. This would still improve performance if the client does read the data, but with less pressure on server memory.

An example use case for this hint is a database that reads in a single record that points to additional records in either other areas of the same file or different files located on the same or different server. While it is likely that the application may access the additional records, it is far from guaranteed. Therefore, the database may issue an opportunistic prefetch (instead of POSIX_FADV_WILLNEED) for the data in the other files pointed to by the record.

Another use case is "Direct I/O", which allows a stated holder to inform the server that it does not wish to cache data. Today, for applications that only intend to read data once, the use of direct I/O disables client caching, but does not affect server caching. By caching data that will not be re-read, the server is polluting its cache and possibly causing useful cached data to be evicted. By informing the server of its expected I/O access, this situation can be avoid. Direct I/O can be used in Linux and AIX via the open() O_DIRECT parameter, in Solaris via the directio() function, and in Windows via the CreateFile() FILE_FLAG_NO_BUFFERING flag.

Another use case is "Backward Sequential Read", which allows a stated holder to inform the server that it intends to read the specified data backwards, i.e., back the end to the beginning. This is different than POSIX_FADV_SEQUENTIAL, whose implied intention was that data will be read from beginning to end. This hint allows

servers to prefetch data at the end of the range first, and then prefetch data sequentially in a backwards manner to the start of the data range. One example of an application that can make use of this hint is video editing.

5.4. Security Considerations

None.

5.5. IANA Considerations

The IO_ADVISE_type4 will be extended through an IANA registry.

6. Application Data Block 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 [18]). 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. An ADB Number (ADBN), which allows the application to determine which data block is being referenced. The ADBN is a logical block number and 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 [18] and detect corruption in data blocks [19]. 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. When the application is initializing a file to have the ADB structure, it could compress the ADBs to just the information to necessary to later reconstruct the header portion of

the ADB when the contents are read back. Using sparse file techniques, the disk blocks described by would not be allocated. Unlike sparse file techniques, there would be a small cost to store the compressed header data.

In this section, we are going to define a generic framework for an ADB, present one approach to detecting corruption in a given ADB implementation, and describe the model for how the client and server can support efficient initialization of ADBs, reading of ADB holes, punching holes in ADBs, and space reservation. Further, we need to be able to extend this model to applications which do not support ADBs, but wish to be able to handle sparse files, hole punching, and space reservation.

6.1. Generic Framework

We want the representation of the ADB 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 ADB. One might store the ADBN at the start of the block and then have a guard pattern to detect corruption [20]. The next might store the ADBN at an offset of 100 bytes within the block and have no guard pattern at all. The point is that 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 ADB.

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.

6.1.1. Data Block Representation

```
struct app_data_block4 {
        offset4         adb_offset;
        length4         adb_block_size;
        length4         adb_block_count;
        length4         adb_reloff_blocknum;
        count4          adb_block_num;
        length4         adb_reloff_pattern;
        opaque          adb_pattern<>;
};
```

The app_data_block4 structure captures the abstraction presented for the ADB. The additional fields present are to allow the transmission of adb_block_count ADBs at one time. We also use adb_block_num to convey the ADBN of the first block in the sequence. Each ADB will contain the same adb_pattern string.

As both adb_block_num and adb_pattern are optional, if either adb_reloff_pattern or adb_reloff_blocknum is set to NFS4_UINT64_MAX, then the corresponding field is not set in any of the ADB.

6.1.2. Data Content

```
/*
 * Use an enum such that we can extend new types.
 */
enum data_content4 {
        NFS4_CONTENT_DATA = 0,
        NFS4_CONTENT_APP_BLOCK = 1,
        NFS4_CONTENT_HOLE = 2
};
```

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

6.2. pNFS Considerations

While this document does not mandate how sparse ADBs are recorded on the server, it does make the assumption that such information is not in the file. I.e., the information is metadata. As such, the INITIALIZE operation is defined to be not supported by the DS - it must be issued to the MDS. But since the client must not assume a priori whether a read is sparse or not, the READ_PLUS operation MUST be supported by both the DS and the MDS. I.e., the client might impose on the MDS to asynchronously read the data from the DS.

Furthermore, each DS MUST not report to a client either a sparse ADB or data which belongs to another DS. One implication of this requirement is that the app_data_block4's adb_block_size MUST be either be the stripe width or the stripe width must be an even multiple of it.

The second implication here is that the DS must be able to use the Control Protocol to determine from the MDS where the sparse ADBs occur. [[Comment.4: Need to discuss what happens if after the file is being written to and an INITIALIZE occurs? --TH]] Perhaps instead of the DS pulling from the MDS, the MDS pushes to the DS? Thus an INITIALIZE causes a new push? [[Comment.5: Still need to consider

race cases of the DS getting a WRITE and the MDS getting an INITIALIZE. --TH]]

6.3. An Example of Detecting Corruption

In this section, we define an ADB 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 ADB 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 [21] 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 ADB - it need not make the transfer layer aware of the fact that there is a checksum (see [19] for an example of checksums used to detect corruption in application data blocks).

Corruption in each ADB can be detected thusly:

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

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

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

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

- 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 ADB. 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.

6.4. Example of READ_PLUS

The hypothetical application presented in Section 6.3 can be used to illustrate how READ_PLUS would return an array of results. A file is created and initialized with 100 4k ADBs in the FREE state:

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

Further, assume the application writes a single ADB 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 ADB, some data, and a final ADB:

```
ADB {0, 4, 0, 0, 8, 0xfeedface}
data 4k
```

```
ADB {20k, 4k, 59, 0, 6, 0xfeedface}
```

6.5. Zero Filled Holes

As applications are free to define the structure of an ADB, it is trivial to define an ADB which supports zero filled holes. Such a case would encompass the traditional definitions of a sparse file and hole punching. For example, to punch a 64k hole, starting at 100M, into an existing file which has no ADB structure:

```
INITIALIZE {100M, 64k, 1, NFS4_UINT64_MAX,
            0, NFS4_UINT64_MAX, 0x0}
```

7. Labeled NFS

7.1. 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. 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.

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 a method for the server to notify the client that the attribute changed on an open file on the server. If the file is closed, then during the open attempt, the client will gather the new attribute value. The server MUST not communicate the new value of the attribute, the client MUST query it. This

requirement stems from the need for the client to provide sufficient access rights to the attribute.

The final change necessary is a modification to the RPC layer used in NFSv4 in the form of a new version of the RPCSEC_GSS [7] framework. In order for an NFSv4 server to apply MAC checks it must obtain additional information from the client. Several methods were explored for performing this and it was decided that the best approach was to incorporate the ability to make security attribute assertions through the RPC mechanism. RPCSECGSSv3 [5] outlines a method to assert additional security information such as security labels on gss context creation and have that data bound to all RPC requests that make use of that context.

7.2. 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.

Domain of Interpretation (DOI): represents an administrative security boundary, where all systems within the DOI have semantically coherent labeling. That is, a security attribute must always mean exactly the same thing anywhere within the DOI.

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

Multi-Level Security (MLS): is a traditional model where objects are given a sensitivity level (Unclassified, Secret, Top Secret, etc) and a category set [22].

7.3. 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 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 [23] 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 will be used to satisfy this requirement.

7.3.1. Interpreting FATTR4_SEC_LABEL

The XDR [24] necessary to implement Labeled NFSv4 is presented below:

```
const FATTR4_SEC_LABEL   = 81;

typedef uint32_t  policy4;
```

Figure 6

```
struct labelformat_spec4 {
        policy4 lfs_lfs;
        policy4 lfs_pi;
};

struct sec_label_attr_info {
        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 [23]. 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.

7.3.2. Delegations

In the event that a security attribute is changed on the server while a client holds a delegation on the file, the client should follow the existing protocol with respect to attribute changes. It should flush all changes back to the server and relinquish the delegation.

7.3.3. 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 7.6 gives a specific example of how the security attribute is handled under a particular MAC model.

7.3.4. 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 aware it must always provide the initial security attribute upon file creation. In the event that the server is the only MAC aware entity in the system it should ignore the security attribute specified by the client and instead make the determination itself. A more in depth explanation can be found in Section 7.6.

7.3.5. Existing Objects

Note that under the MAC model, all objects must have labels. Therefore, if an existing server is upgraded to include LNFS support, then it is the responsibility of the security system to define the behavior for existing objects. For example, if the security system is LFS 0, which means the server just stores and returns labels, then existing files should return labels which are set to an empty value.

7.3.6. Label Changes

As per the requirements, when a file's security label is modified, the server must notify all clients which have the file opened of the change in label. It does so with CB_ATTR_CHANGED. There are preconditions to making an attribute change imposed by NFSv4 and the security system might want to impose others. In the process of meeting these preconditions, the server may chose to either serve the request in whole or return NFS4ERR_DELAY to the SETATTR operation.

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 7.3.2 must be followed.

As the server is always presented with the subject label from the client, it does not necessarily need to communicate the fact that the label has changed to the client. In the cases where the change outright denies the client access, the client will be able to quickly determine that there is a new label in effect. It is in cases where the client may share the same object between multiple subjects or a security system which is not strictly hierarchical that the CB_ATTR_CHANGED callback is very useful. It allows the server to inform the clients that the cached security attribute is now stale.

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 smart client. So if client A changes a security label on a file, then the server MUST inform all clients that have the file opened that the label has changed via CB_ATTR_CHANGED. Then the clients MUST retrieve the new label and MUST enforce access via the new attribute values.

[[Comment.6: Describe a LFS of 0, which will be the means to indicate such a deployment. In the current LFR, 0 is marked as reserved. If we use it, then we define the default LFS to be used by a LNFS aware server. I.e., it lets smart clients work together in the face of a dumb server. Note that will supporting this system is optional, it will make for a very good debugging mode during development. I.e., even if a server does not deploy with another security system, this mode gets your foot in the door. --TH]]

7.4. pNFS Considerations

This section examines the issues in deploying LNFS in a pNFS community of servers.

7.4.1. MAC Label Checks

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 [2] 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.

If a file's FATTR4_SEC_LABEL is changed, then the MDS should utilize CB_ATTR_CHANGED to inform the client of that fact. If the MDS is maintaining

7.5. Discovery of Server LNFS Support

The server can easily determine that a client supports LNFS when it queries for the FATTR4_SEC_LABEL label for an object. Note that it cannot assume that the presence of RPCSEC_GSSv3 indicates LNFS support. The client might need to discover which LFS the server supports.

A server which supports LNFS MUST allow a client with any subject label to retrieve the FATTR4_SEC_LABEL attribute for the root filehandle, ROOTFH. The following compound must always succeed as far as a MAC label check is concerned:

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.

7.6. MAC Security NFS Modes of Operation

A system using Labeled NFS may operate in three 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 two modes are variations on each other and are called "smart client" and "smart server" modes. In these modes one end of the connection is not implementing a MAC model and because of this these operating modes offer less protection than full mode.

7.6.1. Full Mode

Full mode environments consist of MAC aware 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 7.3. The security attribute of the subject making the request is transported at the RPC layer using the mechanism described in RPCSECGSSv3 [5].

7.6.1.1. 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 and DOIs. 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 and DOI then the request SHOULD be denied. Another recourse is to allow the host to provide a fallback mapping for unknown security attributes.

7.6.1.2. 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 extracts the security attribute conveyed in the RPC request. The server then uses this security attribute and the attribute 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.

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.

7.6.2. Smart Client Mode

Smart client environments consist of NFSv4 servers that are not MAC aware but NFSv4 clients that are. Clients in this environment are may consist of groups implementing different MAC models policies. The system requires that all clients in the environment be responsible for access control checks. Due to the amount of trust placed in the clients this mode is only to be used in a trusted environment.

7.6.2.1. Initial Labeling and Translation

Just like in full mode the client is responsible for determining the initial label upon object creation. The server in smart client mode does not implement a MAC model, however, it may provide the ability to restrict the creation and labeling of object with certain labels based on different criteria as described in Section 7.6.1.2.

In a smart client environment a group of clients operate in a single DOI. This removes the need for the clients to maintain a set of DOI translations. Servers should provide a method to allow different groups of clients to access the server at the same time. However it should not let two groups of clients operating in different DOIs to access the same files.

7.6.2.2. Policy Enforcement

In smart client mode access control decisions are made by the clients. When a client accesses an object it obtains the security attribute of the object from the server and combines it with the security attribute of the process making the request to make an access control decision. This check is in addition to the DAC checks provided by NFSv4 so this may fail based on the DAC criteria even if the MAC policy grants access. As the policy check is located on the client an access control denial should take the form that is native to the platform.

7.6.3. Smart Server Mode

Smart server environments consist of NFSv4 servers that are MAC aware and one or more MAC unaware clients. 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.

7.6.3.1. Initial Labeling and Translation

In smart server mode all labeling and access control decisions are performed by the NFSv4 server. In this environment the NFSv4 clients are not MAC aware so they cannot provide input into the access control decision. This requires the server to determine the initial labeling of objects. Normally the subject to use in this calculation would originate from the client. Instead the NFSv4 server may choose to assign the subject security attribute based on their authentication credentials and/or associated network attributes (e.g., IP address, network interface).

In smart server mode security attributes are contained solely within the NFSv4 server. This means that all security attributes used in the system remain within a single LFS and DOI. Since security attributes will not cross DOIs or change format there is no need to provide any translation functionality above that which is needed internally by the MAC model.

7.6.3.2. Policy Enforcement

All access control decisions in smart server mode are made by the server. The server will assign the subject a security attribute based on some criteria (e.g., IP address, network interface). Using the newly calculated security attribute and the security attribute of the object being requested the MAC model makes the access control check and returns NFS4ERR_ACCESS on a denial and NFS4_OK on success. This check is done transparently to the client so if the MAC permission check fails the client may be unaware of the reason for the permission failure. When operating in this mode administrators attempting to debug permission failures should be aware to check the MAC policy running on the server in addition to the DAC settings.

7.7. Security Considerations

This entire document 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-LNFS client from mixing entries in the directory cache.

8. Sharing change attribute implementation details with NFSv4 clients

8.1. Introduction

Although both the NFSv4 [11] and NFSv4.1 protocol [2], define the change attribute as being mandatory to implement, there is little in the way of guidance. The only feature that is mandated by them 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 corresponds 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 serialised 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.

8.2. Definition of the 'change_attr_type' per-file system attribute

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

Name	Id	Data Type	Acc
change_attr_type	XX	enum change_attr_typeinfo	R

The solution enables the NFS server to provide additional information about how it expects the change attribute value to evolve after the file data or metadata has changed. 'change_attr_type' is defined as a new recommended attribute, and takes values from enum change_attr_typeinfo as follows:

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 [11] in terms of the time_metadata attribute.

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

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

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

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

9. Security Considerations

10. File Attributes

10.1. Attribute Definitions

10.1.1. Attribute 77: space_reserved

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

If the client tries to grow a file which has the space_reserved attribute set, the server must guarantee that there is disk space to accommodate every byte in the file with the new size before it can

return success. If the server cannot guarantee this, it must return NFS4ERR_NOSPC.

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

The value of space_reserved can be obtained at any time through GETATTR.

In order to avoid ambiguity, the space_reserve bit cannot be set along with the size bit in SETATTR. Increasing the size of a file with space_reserve set will fail if space reservation cannot be guaranteed for the new size. If the file size is decreased, space reservation is only guaranteed for the new size and the extra blocks backing the file can be released.

10.1.2. 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.

11. 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 MUST NOT implement. 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 [2]..

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

REQ REQUIRED to implement

REC RECOMMEND to implement

OPT OPTIONAL to implement

MNI MUST NOT implement

For the NFSv4.2 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

ADB Application Data Blocks

Operations

Operation	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)
COPY_ABORT	OPT	COPY (REQ)
COPY_NOTIFY	OPT	COPY (REQ)
COPY_REVOKE	OPT	COPY (REQ)
COPY_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	
INITIALIZE	OPT	ADB (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	OPT	
READDIR	REQ	
READLINK	OPT	
READ_PLUS	OPT	ADB (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	

Callback Operations

Operation	REQ, REC, OPT, or MNI	Feature (REQ, REC, or OPT)
CB_COPY	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)

12. NFSv4.2 Operations

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

12.1.1. ARGUMENT

```
const COPY4_GUARDED     = 0x00000001;
const COPY4_METADATA    = 0x00000002;

struct COPY4args {
        /* SAVED_FH: source file */
        /* CURRENT_FH: destination file or */
        /*             directory           */
        offset4         ca_src_offset;
        offset4         ca_dst_offset;
        length4         ca_count;
        uint32_t        ca_flags;
        component4      ca_destination;
        netloc4         ca_source_server<>;
};
```

12.1.2. RESULT

```
union COPY4res switch (nfsstat4 cr_status) {
        case NFS4_OK:
                stateid4        cr_callback_id<1>;
        default:
                length4         cr_bytes_copied;
};
```

12.1.3. 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 combination of CURRENT_FH and ca_destination.

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

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

```
PUTFH source-fh
SAVEFH
```

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

In order to avoid this problem, the minor version incorporating the COPY operations will need to make a few small changes in the handling of existing operations. If a server supports the server-to-server COPY feature, a PUTFH followed by a SAVEFH MUST NOT return NFS4ERR_STALE for either operation. These restrictions do not pose substantial difficulties for servers. The CURRENT_FH and SAVED_FH may be validated in the context of the operation referencing them and an NFS4ERR_STALE error returned for an invalid file handle at that point.

The CURRENT_FH and ca_destination together specify the destination of the copy operation. If ca_destination is of 0 (zero) length, then CURRENT_FH specifies the target file. In this case, CURRENT_FH MUST be a regular file and not a directory. If ca_destination is not of 0 (zero) length, the ca_destination argument specifies the file name to which the data will be copied within the directory identified by CURRENT_FH. In this case, CURRENT_FH MUST be a directory and not a regular file.

If the file named by ca_destination does not exist and the operation completes successfully, the file will be visible in the file system namespace. If the file does not exist and the operation fails, the file MAY be visible in the file system namespace depending on when the failure occurs and on the implementation of the NFS server receiving the COPY operation. If the ca_destination name cannot be created in the destination file system (due to file name restrictions, such as case or length), the operation MUST fail.

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

(zero) bytes, the amount of data copied is implementation dependent (clients may guard against this case by specifying a non-zero count value or preventing modification of the source file as mentioned above).

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

If the destination file is created as a result of this command, the destination file's size will be equal to the number of bytes successfully copied. If the destination file already existed, the destination file's size may increase as a result of this operation (e.g. if ca_dst_offset plus ca_count is greater than the destination's initial size).

If the ca_source_server list is specified, then this is an inter-server copy operation and the source file is on a remote server. The client is expected to have previously issued a successful COPY_NOTIFY request to the remote source server. The ca_source_server list SHOULD be the same as the COPY_NOTIFY response's cnr_source_server list. If the client includes the entries from the COPY_NOTIFY response's cnr_source_server list in the ca_source_server list, the source server can indicate a specific copy protocol for the destination server to use by returning a URL, which specifies both a protocol service and server name. Server-to-server copy protocol considerations are described in Section 2.2.3 and Section 2.4.1.

The ca_flags argument allows the copy operation to be customized in the following ways using the guarded flag (COPY4_GUARDED) and the metadata flag (COPY4_METADATA).

If the guarded flag is set and the destination exists on the server, this operation will fail with NFS4ERR_EXIST.

If the guarded flag is not set and the destination exists on the server, the behavior is implementation dependent.

If the metadata flag is set and the client is requesting a whole file copy (i.e., ca_count is 0 (zero)), a subset of the destination file's attributes MUST be the same as the source file's corresponding attributes and a subset of the destination file's attributes SHOULD be the same as the source file's corresponding attributes. The attributes in the MUST and SHOULD copy subsets will be defined for each NFS version.

For NFSv4.1, Table 1 and Table 2 list the REQUIRED and RECOMMENDED attributes respectively. A "MUST" in the "Copy to destination file?" column indicates that the attribute is part of the MUST copy set. A "SHOULD" in the "Copy to destination file?" column indicates that the attribute is part of the SHOULD copy set.

Name	Id	Copy to destination file?
supported_attrs	0	no
type	1	MUST
fh_expire_type	2	no
change	3	SHOULD
size	4	MUST
link_support	5	no
symlink_support	6	no
named_attr	7	no
fsid	8	no
unique_handles	9	no
lease_time	10	no
rdattr_error	11	no
filehandle	19	no
suppattr_exclcreat	75	no

Table 1

Name	Id	Copy to destination file?
acl	12	MUST
aclsupport	13	no
archive	14	no
cansettime	15	no
case_insensitive	16	no
case_preserving	17	no
change_policy	60	no
chown_restricted	18	MUST
dacl	58	MUST
dir_notif_delay	56	no
dirent_notif_delay	57	no
fileid	20	no
files_avail	21	no
files_free	22	no
files_total	23	no
fs_charset_cap	76	no
fs_layout_type	62	no
fs_locations	24	no

fs_locations_info	67	no
fs_status	61	no
hidden	25	MUST
homogeneous	26	no
layout_alignment	66	no
layout_blksize	65	no
layout_hint	63	no
layout_type	64	no
maxfilesize	27	no
maxlink	28	no
maxname	29	no
maxread	30	no
maxwrite	31	no
mdsthreshold	68	no
mimetype	32	MUST
mode	33	MUST
mode_set_masked	74	no
mounted_on_fileid	55	no
no_trunc	34	no
numlinks	35	no
owner	36	MUST
owner_group	37	MUST
quota_avail_hard	38	no
quota_avail_soft	39	no
quota_used	40	no
rawdev	41	no
retentevt_get	71	MUST
retentevt_set	72	no
retention_get	69	MUST
retention_hold	73	MUST
retention_set	70	no
sacl	59	MUST
space_avail	42	no
space_free	43	no
space_freed	78	no
space_reserved	77	MUST
space_total	44	no
space_used	45	no
system	46	MUST
time_access	47	MUST
time_access_set	48	no
time_backup	49	no
time_create	50	MUST
time_delta	51	no
time_metadata	52	SHOULD
time_modify	53	MUST
time_modify_set	54	no

Table 2

[NOTE: The source file's attribute values will take precedence over any attribute values inherited by the destination file.]

In the case of an inter-server copy or an intra-server copy between file systems, the attributes supported for the source file and destination file could be different. By definition,the REQUIRED attributes will be supported in all cases. If the metadata flag is set and the source file has a RECOMMENDED attribute that is not supported for the destination file, the copy MUST fail with NFS4ERR_ATTRNOTSUPP.

Any attribute supported by the destination server that is not set on the source file SHOULD be left unset.

Metadata attributes not exposed via the NFS protocol SHOULD be copied to the destination file where appropriate.

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

If the metadata flag is not set and the client is requesting a whole file copy (i.e., ca_count is 0 (zero)), the destination file's metadata is implementation dependent.

If the client is requesting a partial file copy (i.e., ca_count is not 0 (zero)), the client SHOULD NOT set the metadata flag and the server MUST ignore the metadata flag.

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

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

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

If the cr_callback_id is returned, this indicates that the operation

was initiated and a CB_COPY callback will deliver the final results of the operation. The cr_callback_id stateid is termed a copy stateid in this context. The server is given the option of returning the results in a callback because the data may require a relatively long period of time to copy.

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

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

In the event of a failure the state of the destination file is implementation dependent. The COPY operation may fail for the following reasons (this is a partial list).

NFS4ERR_MOVED: The file system which contains the source file, or the destination file or directory is not present. The client can determine the correct location and reissue the operation with the correct location.

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the NFS server receiving this request.

NFS4ERR_PARTNER_NOTSUPP: The remote server does not support the server-to-server copy offload protocol.

NFS4ERR_OFFLOAD_DENIED: 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.

NFS4ERR_PARTNER_NO_AUTH: The remote server does not authorize a server-to-server copy offload operation. This may be due to the client's failure to send the COPY_NOTIFY operation to the remote server, the remote server receiving a server-to-server copy offload request after the copy lease time expired, or for some other permission problem.

NFS4ERR_FBIG: The copy operation would have caused the file to grow beyond the server's limit.

NFS4ERR_NOTDIR: The CURRENT_FH is a file and ca_destination has non-zero length.

NFS4ERR_WRONG_TYPE: The SAVED_FH is not a regular file.

NFS4ERR_ISDIR: The CURRENT_FH is a directory and ca_destination has zero length.

NFS4ERR_INVAL: The source offset or offset plus count are greater than or equal to the size of the source file.

NFS4ERR_DELAY: The server does not have the resources to perform the copy operation at the current time. The client should retry the operation sometime in the future.

NFS4ERR_METADATA_NOTSUPP: The destination file cannot support the same metadata as the source file.

NFS4ERR_WRONGSEC: The security mechanism being used by the client does not match the server's security policy.

12.2. Operation 60: COPY_ABORT - Cancel a server-side copy

12.2.1. ARGUMENT

```
struct COPY_ABORT4args {
        /* CURRENT_FH: desination file */
        stateid4        caa_stateid;
};
```

12.2.2. RESULT

```
struct COPY_ABORT4res {
        nfsstat4        car_status;
};
```

12.2.3. DESCRIPTION

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

some other reason.

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

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

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

The COPY_ABORT operation may fail for the following reasons (this is a partial list):

NFS4ERR_NOTSUPP: The abort operation is not supported by the NFS server receiving this request.

NFS4ERR_RETRY: The abort failed, but a retry at some time in the future MAY succeed.

NFS4ERR_COMPLETE_ALREADY: The abort failed, and a callback will deliver the results of the copy operation.

NFS4ERR_SERVERFAULT: An error occurred on the server that does not map to a specific error code.

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

12.3.1. ARGUMENT

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

12.3.2. 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;
};
```

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

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

- 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.

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

The cnr_lease_time is chosen by the source server. A cnr_lease_time of 0 (zero) indicates an infinite lease. To renew the copy lease time the client should resend the same copy notification request to the source server.

To avoid the need for synchronized clocks, copy lease times are granted by the server as a time delta. However, there is a requirement that the client and server clocks do not drift

excessively over the duration of the lease. There is also the issue of propagation delay across the network which could easily be several hundred milliseconds as well as the possibility that requests will be lost and need to be retransmitted.

To take propagation delay into account, the client should subtract it from copy lease times (e.g., if the client estimates the one-way propagation delay as 200 milliseconds, then it can assume that the lease is already 200 milliseconds old when it gets it). In addition, it will take another 200 milliseconds to get a response back to the server. So the client must send a lease renewal or send the copy offload request to the cna_destination_server at least 400 milliseconds before the copy lease would expire. If the propagation delay varies over the life of the lease (e.g., the client is on a mobile host), the client will need to continuously subtract the increase in propagation delay from the copy lease times.

The server's copy lease period configuration should take into account the network distance of the clients that will be accessing the server's resources. It is expected that the lease period will take into account the network propagation delays and other network delay factors for the client population. Since the protocol does not allow for an automatic method to determine an appropriate copy lease period, the server's administrator may have to tune the copy lease period.

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

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

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

The COPY_NOTIFY operation may fail for the following reasons (this is a partial list):

NFS4ERR_MOVED: The file system which contains the source file is not present on the source server. The client can determine the correct location and reissue the operation with the correct location.

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the NFS server receiving this request.

NFS4ERR_WRONGSEC: The security mechanism being used by the client does not match the server's security policy.

12.4. Operation 62: COPY_REVOKE - Revoke a destination server's copy privileges

12.4.1. ARGUMENT

```
struct COPY_REVOKE4args {
        /* CURRENT_FH: source file */
        netloc4         cra_destination_server;
};
```

12.4.2. RESULT

```
struct COPY_REVOKE4res {
        nfsstat4        crr_status;
};
```

12.4.3. 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 cra_destination_server from reading the file specified by CURRENT_FH on behalf of given user. If the cra_destination_server has already begun copying the file, a successful return from this operation indicates that further access will be prevented.

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

The COPY_REVOKE operation is useful in situations in which the source server granted a very long or infinite lease on the destination server's ability to read the source file and all copy operations on the source file have been completed.

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

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

The COPY_REVOKE operation may fail for the following reasons (this is a partial list):

NFS4ERR_MOVED: The file system which contains the source file is not present on the source server. The client can determine the correct location and reissue the operation with the correct location.

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the NFS server receiving this request.

12.5. Operation 63: COPY_STATUS - Poll for status of a server-side copy

12.5.1. ARGUMENT

```
struct COPY_STATUS4args {
        /* CURRENT_FH: destination file */
        stateid4        csa_stateid;
};
```

12.5.2. RESULT

```
struct COPY_STATUS4resok {
        length4         csr_bytes_copied;
        nfsstat4        csr_complete<1>;
};

union COPY_STATUS4res switch (nfsstat4 csr_status) {
        case NFS4_OK:
                COPY_STATUS4resok       resok4;
        default:
                void;
};
```

12.5.3. DESCRIPTION

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

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

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

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

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

The COPY_STATUS operation may fail for the following reasons (this is a partial list):

NFS4ERR_NOTSUPP: The copy status operation is not supported by the NFS server receiving this request.

NFS4ERR_BAD_STATEID: The stateid is not valid (see Section 2.3.2 below).

NFS4ERR_EXPIRED: The stateid has expired (see Copy Offload Stateid section below).

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

12.6.1. ARGUMENT

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

12.6.2. RESULT

Unchanged

12.6.3. 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 abort/complete all outstanding operations.

12.6.4. 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. If the client ID is created with this capability then the following will occur:

- The server will not reply to DESTROY_SESSION 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.
- When DESTROY_CLIENTID is invoked, if there are sessions (both idle and non-idle), opens, locks, delegations, layouts, and/or wants (Section 18.49) associated with the client ID are removed. Pending operations will be completed or aborted before the sessions, opens, locks, delegations, layouts, and/or wants are deleted.
- 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.

12.7. Operation 64: INITIALIZE

This operation can be used to initialize the structure imposed by an application onto a file and to punch a hole into a file.

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 ADBs. It uses the INITIALIZE operation to do so.

12.7.1. ARGUMENT

```
/*
 * We use data_content4 in case we wish to
 * extend new types later. Note that we
 * are explicitly disallowing data.
 */
union initialize_arg4 switch (data_content4 content) {
case NFS4_CONTENT_APP_BLOCK:
        app_data_block4 ia_adb;
case NFS4_CONTENT_HOLE:
        data_info4      ia_hole;
default:
        void;
};

struct INITIALIZE4args {
        /* CURRENT_FH: file */
        stateid4        ia_stateid;
        stable_how4     ia_stable;
        initialize_arg4 ia_data<>;
};
```

12.7.2. RESULT

```
struct INITIALIZE4resok {
        count4          ir_count;
        stable_how4     ir_committed;
        verifier4       ir_writeverf;
        data_content4   ir_sparse;
};

union INITIALIZE4res switch (nfsstat4 status) {
case NFS4_OK:
        INITIALIZE4resok        resok4;
default:
        void;
};
```

12.7.3. DESCRIPTION

When the client invokes the INITIALIZE 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 INITIALIZE operation, it still might not support sparse files. So if it receives the INITIALIZE operation, then it MUST populate the contents of the file with the initialized ADBs. In other words, if the server supports INITIALIZE, then it supports the concept of ADBs. [[Comment.7: Do we want to support an asynchronous INITIALIZE? Do we have to? --TH]] [[Comment.8: Need to document union arm error code. --TH]]

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 ADB.

If the data blocks were already allocated, then the INITIALIZE is a hole punch operation. If INITIALIZE supports sparse files, then the data blocks are to be deallocated. If not, then the data blocks are to be rewritten in the indicated ADB format. [[Comment.9: Need to document interaction between space reservation and hole punching? --TH]]

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

This document does not mandate the manner in which the server stores ADBs sparsely for a file. It does assume that if ADBs are stored sparsely, then the server can detect when an INITIALIZE arrives that will force a new ADB to start inside an existing ADB. For example, assume that ADBi has a adb_block_size of 4k and that an INITIALIZE starts 1k inside ADBi. The server should [[Comment.10: Need to flesh this out. --TH]]

12.7.3.1. Hole punching

Whenever a client wishes to deallocate the blocks backing a particular region in the file, it calls the INITIALIZE operation with the current filehandle set to the filehandle of the file in question, start offset and length in bytes of the region set in hpa_offset and hpa_count respectively. All further reads to this region MUST return

zeros until overwritten. The filehandle specified must be that of a regular file.

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

The server is not required to complete deallocating the blocks specified in the operation before returning. It is acceptable to have the deallocation be deferred. In fact, INITIALIZE is merely a hint; it is valid for a server to return success without ever doing anything towards deallocating the blocks backing the region specified. However, any future reads to the region MUST return zeroes.

If used to hole punch, INITIALIZE 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 INITIALIZE operation MUST NOT change the space reservation guarantee of the file. While the server can deallocate the blocks specified by hpa_offset and hpa_count, future writes to this region MUST NOT fail with NFSERR_NOSPC.

The INITIALIZE operation may fail for the following reasons (this is a partial list):

NFS4ERR_NOTSUPP The Hole punch operations are not supported by the NFS server receiving this request.

NFS4ERR_DIR The current filehandle is of type NF4DIR.

NFS4ERR_SYMLINK The current filehandle is of type NF4LNK.

NFS4ERR_WRONG_TYPE The current filehandle does not designate an ordinary file.

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

This section introduces a new operation, named IO_ADVISE, which allows NFS clients to communicate application I/O access pattern hints to the NFS server. This new operation will allow hints to be sent to the server when applications use posix_fadvise, direct I/O, or at any other point at which the client finds useful.

12.8.1. 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
};

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

12.8.2. RESULT

```
struct IO_ADVISE4resok {
        bitmap4 ior_hints;
};

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

12.8.3. DESCRIPTION

The IO_ADVISE operation sends an I/O access pattern hint to the server for the owner of stated 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 possible hints:

IO_ADVISE4_NORMAL Specifies that the application has no advice to give on its behavior with respect to the specified data. It is the default characteristic if no advice is given.

IO_ADVISE4_SEQUENTIAL Specifies that the stated holder expects to access the specified data sequentially from lower offsets to higher offsets.

IO_ADVISE4_SEQUENTIAL BACKWARDS Specifies that the stated holder expects to access the specified data sequentially from higher offsets to lower offsets.

IO_ADVISE4_RANDOM Specifies that the stated holder expects to access the specified data in a random order.

IO_ADVISE4_WILLNEED Specifies that the stated holder expects to access the specified data in the near future.

IO_ADVISE4_WILLNEED_OPPORTUNISTIC Specifies that the stated holder 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 Specifies that the stated holder expects that it will not access the specified data in the near future.

IO_ADVISE_NOREUSE Specifies that the stated holder expects to access the specified data once and then not reuse it thereafter.

IO_ADVISE4_READ Specifies that the stated holder expects to read the specified data in the near future.

IO_ADVISE4_WRITE Specifies that the stated holder expects to write the specified data in the near future.

The server will return success if the operation is properly formed,

otherwise the server will return an error. The server MUST NOT return an error if it does not recognize or does not support the requested advice. This is also true even 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 this case, the server MUST return success and a ior_hints value that indicates the hint it intends to optimize. For contradictory hints, this may mean simply returning IO_ADVISE4_NORMAL for example.

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 stated and byte range.

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

12.8.4. IMPLEMENTATION

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

The most obvious is in direct response to an applications 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.

Another useful point would be when an application indicates it is using direct I/O. Direct I/O may be specified at file open, in which case a IO_ADVISE may be included in the same compound as the OPEN operation with the IO_ADVISE4_NOREUSE flag set. Direct I/O may also

be specified separately, in which case a IO_ADVISE operation can be sent to the server separately. As above, IO_ADVISE4_WRITE and IO_ADVISE4_READ may be set based upon the type of file access specified when the file was opened.

12.8.5. 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, then if the client implements IO_ADVISE, then if it wants the DS to honor IO_ADVISE, the client MUST send the operation to the MDS, and the server will communicate the advice back each DS. If the client sends IO_ADVISE to the DS, then the server MAY return NFS4ERR_NOTSUPP.

If NFL42_UFLG_IO_ADVISE_THRU_MDS is not set, then this indicates to client that if wants to inform the server via IO_ADVISE of the client's intended use of the file, then the client SHOULD send an IO_ADVISE to each DS. 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.

12.8.5.1. 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.

12.8.6. Number of Supported File Segments

In theory IO_ADVISE allows a client and server to support multiple file segments, meaning that different, possibly overlapping, byte ranges of the same open file reference will support different hints. This is not practical, and in general the server will support just one set of hints, and these will apply to the entire file. However, there are some hints that very ephemeral, and are essentially amount to one time instructions to the NFS server, which will be forgotten momentarily after IO_ADVISE is executed.

The following hints will always apply to the entire file, regardless of the specified byte range:

- IO_ADVISE4_NORMAL
- IO_ADVISE4_SEQUENTIAL
- IO_ADVISE4_SEQUENTIAL_BACKWARDS
- IO_ADVISE4_RANDOM

The following hints will always apply to specified byte range, and will treated as one time instructions:

- IO_ADVISE4_WILLNEED
- IO_ADVISE4_WILLNEED_OPPORTUNISTIC
- IO_ADVISE4_DONTNEED
- IO_ADVISE4_NOREUSE

The following hints are modifiers to all other hints, and will apply to the entire file and/or to a one time instruction on the specified byte range:

- IO_ADVISE4_READ
- IO_ADVISE4_WRITE

12.8.7. Possible Additional Hint - IO_ADVISE4_RECENTLY_USED

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

A use case for this hint is that of the NFS client or application restart. In the event of restart, the app's/client's cache will be cold and it will need to fill it from the server. If the server is maintaining a list (LRU most likely) of byte ranges tagged with IO_ADVISE4_RECENTLY_USED, then the server could have stored the data in these ranges into a storage medium that is less expensive than DRAM, and faster than random access magnetic or optical media, such as flash. This allows the end to end application to storage system to co-operate to meet a service level agreement/objective contracted to the end user by the IT provider.

On the other side, this is effectively a hint regarding multi-level caching, and it may be more useful to specify a more formal multi-level caching system. In addition, the action to be taken by the server file system with this hint, and hence its usefulness, is unclear. For example, as most clients already cache data that they know is important, having this data cached twice may be unnecessary. In fact, substantial performance improvements have been demonstrated by making caches more exclusive between each other [25], not the other way around. This means that there is a strong argument to be made that servers should immediately purge the described cached data upon receiving this hint. Other work showed that even infinite sized secondary caches can be largely ineffective [26], but this of course is subject to the workload.

12.9. Changes to Operation 51: LAYOUTRETURN

12.9.1. Introduction

In the pNFS description provided in [2], the client is not enabled to relay an error code from the DS to the MDS. In the specification of the Objects-Based Layout protocol [8], 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 persistent. 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_STALE_STATEID back to the MDS, with the belief that this is a hard error. The MDS on the other hand, is waiting for the client to report such an error. For it, the mission is accomplished in that the client has returned a layout that the MDS had most likley recalled.

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.

12.9.2. ARGUMENT

The ARGUMENT specification of the LAYOUTRETURN operation in section 18.44.1 of [2] is augmented by the following XDR code [24]:

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

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

12.9.3. RESULT

The RESULT of the LAYOUTRETURN operation is unchanged; see section 18.44.2 of [2].

12.9.4. DESCRIPTION

The following text is added to the end of the LAYOUTRETURN operation DESCRIPTION in section 18.44.3 of [2].

When a client used 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 [8]. For both Files-Based Layouts, the field references a layoutreturn_device_error4, which contains an array of layoutreturn_device_error4.

Each individual layoutreturn_device_error4 descibes a single error associated with a DS, which is identfied 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:

NFS4_OKAY: No issues were found for this device.

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.

12.9.5. IMPLEMENTATION

The following text is added to the end of the LAYOUTRETURN operation IMPLEMENTATION in section 18.4.4 of [2].

A client that expects to use pNFS for a mounted filesystem SHOULD check for pNFS support at mount time. This check SHOULD be performed by sending a GETDEVICELIST operation, followed by layout-type-specific checks for accessibility of each storage device returned by GETDEVICELIST. If the NFS server does not support pNFS, the GETDEVICELIST operation will be rejected with an NFS4ERR_NOTSUPP error; in this situation it is up to the client to determine whether it is acceptable to proceed with NFS-only access.

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

server under appropriate conditions.

When an I/O fails 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. 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 12.9.1), 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. Clients must still be aware that the MDS might not have any choice in using the storage device, i.e., there might only be one possible layout for the system.

Another interesting complication is that for existing files, the MDS might have no choice in which storage devices to hand out to clients. The MDS might try to restripe a file across a different storage device, but clients need to be aware that not all implementations have restriping support.

An MDS SHOULD react to a client return of layouts with errors by not using the problematic storage devices in layouts for that client, but 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.

A client MAY perform I/O via the MDS even when the client holds a layout that covers the I/O; servers MUST support this client behavior, and MAY recall layouts as needed to complete I/Os.

12.10. Operation 65: READ_PLUS

READ_PLUS is a new read operation which allows NFS clients to avoid reading holes in a sparse file and to efficiently transfer ADBs. READ_PLUS is guaranteed to perform no worse than READ, and can dramatically improve performance with sparse files.

READ_PLUS supports all the features of the existing NFSv4.1 READ operation [2] and adds a simple yet significant extension to the format of its response. The change allows the client to avoid returning data for portions of the file which are either initialized and contain no backing store or if the result would appear to be so. 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 unitialized data wastes computational and network resources, thus reducing performance. READ_PLUS uses a new result structure that tells the client that the result is all zeroes AND the byte-range of the hole in which the request was made.

If the client sends a READ operation, it is explicitly stating that it is neither supporting sparse files or ADBs. So if a READ occurs on a sparse ADB or file, then the server must expand such data to be raw bytes. If a READ occurs in the middle of a hole or ADB, the server can only send back bytes starting from that offset.

Such an operation is inefficient for transfer of sparse sections of the file. As such, READ is marked as OBSOLETE in NFSv4.2. Instead, a client should issue READ_PLUS. Note that as the client has no a priori knowledge of whether an ADB is present or not, it should always use READ_PLUS.

12.10.1. ARGUMENT

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

12.10.2. RESULT

```
union read_plus_content switch (data_content4 content) {
case NFS4_CONTENT_DATA:
        opaque          rpc_data<>;
case NFS4_CONTENT_APP_BLOCK:
        app_data_block4 rpc_block;
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 status) {
case NFS4_OK:
        read_plus_res4  resok4;
default:
        void;
};
```

12.10.3. DESCRIPTION

The READ_PLUS operation is based upon the NFSv4.1 READ operation [2], 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. READ_PLUS is subject to access permissions checking.

The READ_PLUS result is comprised of an array of rpr_contents, each of which describe a data_content4 type of data. For NFSv4.2, the allowed values are data, ADB, and hole. A server is required to support the data type, but not ADB nor hole. Both an ADB and a hole

must be returned in its entirety - clients must be prepared to get more information than they requested.

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.

The server may elect to return adjacent elements of the same type. For example, the guard pattern or block size of an ADB might change, which would require adjacent elements of type ADB. 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, again subject to access permissions checking. 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 reurn 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.

12.10.4. IMPLEMENTATION

If the server returns a short read, then the client should send another READ_PLUS to get the remaining data. A server may return less data than requested under several circumstances. The file may have been truncated by another client or perhaps on the server itself, changing the file size from what the requesting client believes to be the case. This would reduce the actual amount of data available to the client. It is possible that the server reduced the transfer size and so return a short read result. Server resource exhaustion may also occur in a short read.

If mandatory byte-range locking is in effect for the file, and if the byte-range corresponding to the data to be read from the file is WRITE_LT locked by an owner not associated with the stateid, the server will return the NFS4ERR_LOCKED error. The client should try to get the appropriate READ_LT via the LOCK operation before re-attempting the READ_PLUS. When the READ_PLUS completes, the client should release the byte-range lock via LOCKU. In addition, the server MUST return an array of rpr_contents with values of that are within the owner's locked byte range.

If another client has an OPEN_DELEGATE_WRITE delegation for the file being read, the delegation must be recalled, and the operation cannot proceed until that delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while the delegation remains outstanding. Normally, delegations will not be recalled as a result of a READ_PLUS

operation since the recall will occur as a result of an earlier OPEN. However, since it is possible for a READ_PLUS to be done with a special stateid, the server needs to check for this case even though the client should have done an OPEN previously.

12.10.4.1. Additional pNFS Implementation Information

[[Comment.11: We need to go over this section. --TH]] With pNFS, the semantics of using READ_PLUS remains the same. Any data server MAY return a READ_HOLE result for a READ_PLUS request that it receives.

When a data server chooses to return a READ_HOLE result, it has the option of returning hole information for the data stored on that data server (as defined by the data layout), but it MUST not return a nfs_readplusreshole structure with a byte range that includes data managed by another data server.

1. Data servers that cannot determine hole information SHOULD return HOLE_NOINFO.

2. Data servers that can obtain hole information for the parts of the file stored on that data server, the data server SHOULD return HOLE_INFO and the byte range of the hole stored on that data server.

A data server should do its best to return as much information about a hole 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 for a Hole that is within the owner's locked byte range.

12.10.5. 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 3

Under the given circumstances, if a client was to read the file from beginning to end with a max read size of 64K, the following will be the result. 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. [[Comment.12: Change the results to match array results. --TH]]

1. READ_PLUS(s, 0, 64K) --> NFS_OK, eof = false, data<>[32K]. Return a short read, as the last half of the request was all zeroes. Note that the first hole is read back as all zeros as it is below the Hole Threshhold.

2. READ_PLUS(s, 32K, 64K) --> NFS_OK, readplusrestype4 = READ_HOLE, nfs_readplusreshole(HOLE_INFO)(32K, 224K). The requested range was all zeros, and the current hole begins at offset 32K and is 224K in length.

3. READ_PLUS(s, 256K, 64K) --> NFS_OK, readplusrestype4 = READ_OK, eof = false, data<>[32K]. Return a short read, as the last half of the request was all zeroes.

4. READ_PLUS(s, 288K, 64K) --> NFS_OK, readplusrestype4 = READ_HOLE, nfs_readplusreshole(HOLE_INFO)(288K, 66K).

5. READ_PLUS(s, 354K, 64K) --> NFS_OK, readplusrestype4 = READ_OK, eof = true, data<>[64K].

12.11. Operation 66: SEEK

SEEK is an operation that allows a client to determine the location of the next data_content4 in a file.

12.11.1. ARGUMENT

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

12.11.2. RESULT

```
union seek_content switch (data_content4 content) {
case NFS4_CONTENT_DATA:
        data_info4      sc_data;
case NFS4_CONTENT_APP_BLOCK:
        app_data_block4 sc_block;
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;
};
```

12.11.3. DESCRIPTION

From the given sa_offset, find the next data_content4 of type sa_what in the file. For either a hole or ADB, 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 12.10.3).

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.

13. NFSv4.2 Callback Operations

13.1. Procedure 16: CB_ATTR_CHANGED - Notify Client that the File's Attributes Changed

13.1.1. ARGUMENTS

```
struct CB_ATTR_CHANGED4args {
        nfs_fh4         acca_fh;
        bitmap4         acca_critical;
        bitmap4         acca_info;
};
```

13.1.2. RESULTS

```
struct CB_ATTR_CHANGED4res {
        nfsstat4        accr_status;
};
```

13.1.3. DESCRIPTION

The CB_ATTR_CHANGED callback operation is used by the server to indicate to the client that the file's attributes have been modified on the server. The server does not convey how the attributes have changed, just that they have been modified. The server can inform the client about both critical and informational attribute changes in the bitmask arguments. The client SHOULD query the server about all attributes set in acca_critical. For all changes reflected in acca_info, the client can decide whether or not it wants to poll the server.

The CB_ATTR_CHANGED callback operation with the FATTR4_SEC_LABEL set in acca_critical is the method used by the server to indicate that the MAC label for the file referenced by acca_fh has changed. In many ways, the server does not care about the result returned by the client.

13.2. Operation 15: CB_COPY - Report results of a server-side copy

13.2.1. ARGUMENT

```
union copy_info4 switch (nfsstat4 cca_status) {
        case NFS4_OK:
                void;
        default:
                length4         cca_bytes_copied;
};

struct CB_COPY4args {
        nfs_fh4         cca_fh;
        stateid4        cca_stateid;
        copy_info4      cca_copy_info;
};
```

13.2.2. RESULT

```
struct CB_COPY4res {
        nfsstat4        ccr_status;
};
```

13.2.3. DESCRIPTION

CB_COPY is used for both intra- and inter-server asynchronous copies. The CB_COPY callback informs the client of the result of an asynchronous server-side copy. This operation is sent by the destination server to the client in a CB_COMPOUND request. The copy is identified by the filehandle and stateid arguments. The result is indicated by the status field. If the copy failed, cca_bytes_copied contains the number of bytes copied before the failure occurred. The cca_bytes_copied value indicates the number of bytes copied but not which specific bytes have been copied.

In the absence of an established backchannel, the server cannot signal the completion of the COPY via a CB_COPY callback. The loss of a callback channel would be indicated by the server setting the SEQ4_STATUS_CB_PATH_DOWN flag in the sr_status_flags field of the SEQUENCE operation. The client must re-establish the callback channel to receive the status of the COPY operation. Prolonged loss of the callback channel could result in the server dropping the COPY operation state and invalidating the copy stateid.

If the client supports the COPY operation, the client is REQUIRED to support the CB_COPY operation.

The CB_COPY operation may fail for the following reasons (this is a partial list):

NFS4ERR_NOTSUPP: The copy offload operation is not supported by the NFS client receiving this request.

14. IANA Considerations

This section uses terms that are defined in [27].

15. References

15.1. Normative References

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

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

[3] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 External Data Representation Standard (XDR) Description", March 2011.

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

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

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

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

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

[9] Shepler, S., Eisler, M., and D. Noveck, "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, January 2010.

[10] Black, D., Glasgow, J., and S. Fridella, "Parallel NFS (pNFS) Block/Volume Layout", RFC 5663, January 2010.

15.2. Informative References

[11] Haynes, T. and D. Noveck, "Network File System (NFS) version 4 Protocol", draft-ietf-nfsv4-rfc3530bis-09 (Work In Progress), March 2011.

[12] 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.

[13] 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.

[14] 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.

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

[16] Simpson, W., "PPP Challenge Handshake Authentication Protocol (CHAP)", RFC 1994, August 1996.

[17] VanDeBogart, S., Frost, C., and E. Kohler, "Reducing Seek Overhead with Application-Directed Prefetching", Proceedings of USENIX Annual Technical Conference , June 2009.

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

[19] 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.

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

[21] 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.

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

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

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

[25] Wong, T. and J. Wilkes, "My cache or yours? Making storage more exclusive", Proceedings of the USENIX Annual Technical Conference , 2002.

[26] Muntz, D. and P. Honeyman, "Multi-level Caching in Distributed File Systems", Proceedings of USENIX Annual Technical Conference , 1992.

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

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

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

[30] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, August 1995.

[31] Eisler, M., "NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5", RFC 2623, June 1999.

[32] Callaghan, B., "NFS URL Scheme", RFC 2224, October 1997.

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

[34] Reynolds, J., "Assigned Numbers: RFC 1700 is Replaced by an On-line Database", RFC 3232, January 2002.

[35] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996.

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

Appendix A. Acknowledgments

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.

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 reviwers 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, Sorrin Faibish, Nico Williams, and David Black also contributed in the final push to get this accepted.

Appendix B. RFC Editor Notes

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

[RFC Editor: prior to publishing this document as an RFC, please replace all occurrences of RFCTBD10 with RFCxxxx where xxxx is the RFC number of this document]

Author's Address

Thomas Haynes
NetApp
9110 E 66th St
Tulsa, OK 74133
USA

Phone: +1 918 307 1415
Email: thomas@netapp.com
URI: http://www.tulsalabs.com