Internet Engineering Task Force (IETF)
Request for Comments: 7931
Updates: 7530
Category: Standards Track
ISSN: 2070-1721

D. Noveck, Ed.
HPE
P. Shivam
C. Lever
B. Baker
ORACLE

July 2016

NFSv4.0 Migration: Specification Update

Abstract

The migration feature of NFSv4 allows the transfer of responsibility for a single file system from one server to another without disruption to clients. Recent implementation experience has shown problems in the existing specification for this feature in NFSv4.0. This document identifies the problem areas and provides revised specification text that updates the NFSv4.0 specification in RFC 7530.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7931.

Copyright Notice

Table of Contents

1. Introduction

This Standards Track document corrects the existing definitive specification of the NFSv4.0 protocol described in [RFC7530]. Given this fact, one should take the current document into account when learning about NFSv4.0, particularly if one is concerned with issues that relate to:

- File system migration, particularly when it involves transparent state migration.
- The construction and interpretation of the nfs_client_id4 structure and particularly the requirements on the id string within it, referred to below as a "client ID string".

2. Conventions

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 [RFC2119].

3. Definitions

3.1. Terminology

The following definitions are included to provide an appropriate context for the reader. This section is derived from Section 1.5 of [RFC7530] but has been adapted to the needs of this document.

Boot Instance Id: A boot instance id is an identifier, such as a boot time, allowing two different instances of the same client to be reliably distinguished. A boot instance id is opaque to the server and is often used as the verifier field in the nfs_client_id4 structure, which identifies the client to the server.

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

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

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

Client ID: A client ID is a 64-bit quantity (in the form of a clientid4) used as a unique, shorthand reference to a particular client instance, identified by a client-supplied verifier (in the form of a boot instance id) and client ID string. The server is responsible for supplying the client ID.

File System: A file system is the collection of objects on a server that share the same fsid attribute (see Section 5.8.1.9 of [RFC7530]).

Grace Period: A grace period is an interval of time during which the server will only grant locking requests to reclaim existing locks but not those that create new locks. This gives clients an opportunity to re-establish locking state in response to a potentially disruptive event. The grace period may be general to help deal with server reboot, or it may be specific to a file system to deal with file system migration when transparent state migration is not provided.

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

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

Lock: The term "lock" is used to refer to record (byte-range) locks as well as share reservations unless specifically stated otherwise.

Lock-Owner: Each byte-range lock is associated with a specific lock-owner and an open-owner. The lock-owner consists of a client ID and an opaque owner string. The client presents this to the server to establish the ownership of the byte-range lock as needed.

Open-Owner: Each open file is associated with a specific open-owner, which consists of a client ID and an opaque owner string. The client presents this to the server to establish the ownership of the open as needed.

Server: A server is an entity responsible for coordinating client access to a set of file systems.

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

Trunking: A situation in which multiple physical addresses are connected to the same logical server.

Verifier: A verifier is a quantity, in the form of a verifier4, that allows one party to an interaction to be aware of a reinitialization or other significant change to the state of the other party. In [RFC7530], this term most often designates the verifier field of an nfs_client_id4, in which a boot instance id is placed to allow the server to determine when there has been a client reboot, making it necessary to eliminate locking state associated with the previous instance of the same client.

3.2. Data Type Definitions

This section contains a table that shows where data types referred to in this document are defined.

Item	Section
cb_client4	Section 2.2.11 in [RFC7530]
clientaddr4	Section 2.2.10 in [RFC7530]
clientid4	Section 2.1 in [RFC7530]
lock_owner4	Section 2.2.14 in [RFC7530]
nfs_client_id4	Section 5.2.1 (this document)
open_owner4	Section 2.2.13 in [RFC7530]
verifier4	Section 2.1 in [RFC7530]

4. Background

Implementation experience with transparent state migration has exposed a number of problems with the then existing specifications of this feature in [RFC7530] and predecessors. The symptoms were:

- After migration of a file system, a reboot of the associated client was not appropriately dealt with, in that the state associated with the rebooting client was not promptly freed.

- Situations can arise whereby a given server has multiple leases with the same nfs_client_id4 (consisting of id and verifier fields), when the protocol clearly assumes there can be only one.

- Excessive client implementation complexity since clients have to deal with situations in which a single client can wind up with its locking state with a given server divided among multiple leases each with its own clientid4.

An analysis of these symptoms leads to the conclusion that existing specifications have erred. They assume that locking state, including both state ids and clientid4s, should be transferred as part of transparent state migration. The troubling symptoms arise from the failure to describe how migrating state is to be integrated with existing client definition structures on the destination server.

The need for the server to appropriately merge stateids associated with a common client boot instance encounters a difficult problem. The issue is that the common client practice with regard to the presentation of unique strings specifying client identity makes it essentially impossible for the client to determine whether or not two stateids, originally generated on different servers, are referable to the same client. This practice is allowed and endorsed by the existing NFSv4.0 specification [RFC7530].

However, upon the prototyping of clients implementing an alternative approach, it has been found that there exist servers that do not work well with these new clients. It appears that current circumstances, in which a particular client implementation pattern had been adopted universally, have resulted in some servers not being able to interoperate against alternate client implementation patterns. As a result, we have a situation that requires careful attention to untangling compatibility issues.

This document updates the existing NFSv4.0 specification [RFC7530] as follows:

- It makes clear that NFSv4.0 supports multiple approaches to the construction of client ID strings, including those formerly endorsed by existing NFSV4.0 specifications and those currently being widely deployed.

- It explains how clients can effectively use client ID strings that are presented to multiple servers.

- It addresses the potential compatibility issues that might arise for clients adopting a previously non-favored client ID string construction approach including the existence of servers that have problems with the new approach.

- It gives some guidance regarding the factors that might govern clients' choice of a client ID string construction approach and recommends that clients construct client ID strings in a manner that supports lease merger if they intend to support transparent state migration.

- It specifies how state is to be transparently migrated, including defining how state that arrives at a new server as part of migration is to be merged into existing leases for clients connected to the target server.

- It makes further clarifications and corrections to address cases where the specification text does not take proper account of the issues raised by state migration or where it has been found that the existing text is insufficiently clear. This includes a revised definition of the SETCLIENTID operation in Section 8.4, which replaces Section 16.33 in [RFC7530].

For a more complete explanation of the choices made in addressing these issues, see [INFO-MIGR].

5. Client Identity Definition

This section is a replacement for Sections 9.1.1 and 9.1.2 in [RFC7530]. The replaced sections are named "Client ID" and "Server Release of Client ID", respectively.

It supersedes the replaced sections.

5.1. Differences from Replaced Sections

Because of the need for greater attention to and careful description of this area, this section is much larger than the sections it replaces. The principal changes/additions made by this section are:

- It corrects inconsistencies regarding the possible role or non-role of the client IP address in construction of client ID strings.

- It clearly addresses the need to maintain a non-volatile record across reboots of client ID strings or any changeable values that are used in their construction.

- It provides a more complete description of circumstances leading to clientid4 invalidity and the appropriate recovery actions.
- It presents, as valid alternatives, two approaches to client ID string construction (named "uniform" and "non-uniform") and gives some implementation guidance to help implementers choose one or the other of these.
- It adds a discussion of issues involved for clients in interacting with servers whose behavior is not consistent with use of uniform client ID strings.
- It adds a description of how server behavior might be used by the client to determine when multiple server IP addresses correspond to the same server.

5.2. Client Identity Data Items

The NFSv4 protocol contains a number of protocol entities to identify clients and client-based entities for locking-related purposes:

- The nfs_client_id4 structure, which uniquely identifies a specific client boot instance. That identification is presented to the server by doing a SETCLIENTID operation. The SETCLIENTID operation is described in Section 8.4, which modifies a description in Section 16.33 of [RFC7530].
- The clientid4, which is returned by the server upon completion of a successful SETCLIENTID operation. This id is used by the client to identify itself when doing subsequent locking-related operations. A clientid4 is associated with a particular lease whereby a client instance holds state on a server instance and may become invalid due to client reboot, server reboot, or other circumstances.
- Opaque arrays, which are used together with the clientid4 to designate within-client entities (e.g., processes) as the owners of opens (open-owners) and owners of byte-range locks (lock-owners).

5.2.1. Client Identity Structure

The basis of the client identification infrastructure is encapsulated in the following data structure, which also appears in Section 9.1.1 of [RFC7530]:

```
struct nfs_client_id4 {
        verifier4       verifier;
        opaque          id<NFS4_OPAQUE_LIMIT>;
};
```

The nfs_client_id4 structure uniquely defines a particular client boot instance as follows:

- The id field is a variable-length string that uniquely identifies a specific client. Although it is described here as a string and is often referred to as a "client string", it should be understood that the protocol defines this as opaque data. In particular, those receiving such an id should not assume that it will be in the UTF-8 encoding. Servers MUST NOT reject an nfs_client_id4 simply because the id string does not follow the rules of UTF-8 encoding.

  The encoding and decoding processes for this field (e.g., use of network byte order) need to result in the same internal representation whatever the endianness of the originating and receiving machines.

- The verifier field contains a client boot instance identifier that is used by the server to detect client reboots. Only if the boot instance is different from that which the server has previously recorded in connection with the client (as identified by the id field) does the server cancel the client's leased state. This cancellation occurs once it receives confirmation of the new nfs_clientd4 via SETCLIENTID_CONFIRM. The SETCLIENTID_CONFIRM operation is described in Section 16.34 of [RFC7530].

  In order to prevent the possibility of malicious destruction of the locking state associated with a client, the server MUST NOT cancel a client's leased state if the principal that established the state for a given id string is not the same as the principal issuing the SETCLIENTID.

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

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

- The string should be selected so that subsequent incarnations (e.g., reboots) of the same client cause the client to present the same string. The implementer is cautioned against an approach that requires the string to be recorded in a local file because this precludes the use of the implementation in an environment where there is no local disk and all file access is from an NFSv4 server.

- The string MAY be different for each server network address that the client accesses rather than common to all server network addresses.

  The considerations that might influence a client to use different strings for different network server addresses are explained in Section 5.4.

- The algorithm for generating the string should not assume that the clients' network addresses will remain the same for any set period of time. Even while the client is still running in its current incarnation, changes might occur between client incarnations.

  Changes to the client ID string due to network address changes would result in successive SETCLIENTID operations for the same client appearing as from different clients, interfering with the use of the nfs_client_id4 verifier field to cancel state associated with previous boot instances of the same client.

  The difficulty is more severe if the client address is the only client-based information in the client ID string. In such a case, there is a real risk that after the client gives up the network address, another client, using the same algorithm, would generate a conflicting id string. This would be likely to cause an inappropriate loss of locking state. See Section 5.9 for detailed guidance regarding client ID string construction.

5.2.2. Client Identity Shorthand

Once a SETCLIENTID and SETCLIENTID_CONFIRM sequence has successfully completed, the client uses the shorthand client identifier, of type clientid4, instead of the longer and less compact nfs_client_id4 structure. This shorthand client identifier (a client ID) is assigned by the server and should be chosen so that it will not conflict with a client ID previously assigned by the same server and, to the degree practicable, by other servers as well. This applies across server restarts or reboots.

Establishment of the client ID by a new incarnation of the client also has the effect of immediately breaking any leased state that a previous incarnation of the client might have had on the server, as opposed to forcing the new client incarnation to wait for the leases to expire. Breaking the lease state amounts to the server removing all locks, share reservations, and delegation states not requested using the CLAIM_DELEGATE_PREV claim type associated with a client having the same identity. For a discussion of delegation state recovery, see Section 10.2.1 of [RFC7530].

Note that the SETCLIENTID and SETCLIENTID_CONFIRM operations have a secondary purpose of establishing the information the server needs to make callbacks to the client for the purpose of supporting delegations. The client is able to change this information via SETCLIENTID and SETCLIENTID_CONFIRM within the same incarnation of the client without causing removal of the client's leased state.

Distinct servers MAY assign clientid4s independently, and they will generally do so. Therefore, a client has to be prepared to deal with multiple instances of the same clientid4 value received on distinct IP addresses, denoting separate entities. When trunking of server IP addresses is not a consideration, a client should keep track of <IP-address, clientid4> pairs, so that each pair is distinct. For a discussion of how to address the issue in the face of possible trunking of server IP addresses, see Section 5.4.

Owners of opens and owners of byte-range locks are separate entities and remain separate even if the same opaque arrays are used to designate owners of each. The protocol distinguishes between open-owners (represented by open_owner4 structures) and lock-owners (represented by lock_owner4 structures).

Both sorts of owners consist of a clientid4 and an opaque owner string. For each client, there is a set of distinct owner values used with that client which constitutes the set of known owners of that type, for the given client.

Each open is associated with a specific open-owner while each byte-range lock is associated with a lock-owner and an open-owner, the latter being the open-owner associated with the open file under which the LOCK operation was done.

When a clientid4 is presented to a server and that clientid4 is not valid, the server will reject the request with an error that depends on the reason for clientid4 invalidity. The error NFS4ERR_ADMIN_REVOKED is returned when the invalidation is the result of administrative action. When the clientid4 is unrecognizable, the error NFS4ERR_STALE_CLIENTID or NFS4ERR_EXPIRED may be returned. An unrecognizable clientid4 can occur for a number of reasons:

- A server reboot causing loss of the server's knowledge of the client. (Always returns NFS4ERR_STALE_CLIENTID.)
- Client error sending an incorrect clientid4 or a valid clientid4 to the wrong server. (May return either error.)
- Loss of lease state due to lease expiration. (Always returns NFS4ERR_EXPIRED.)
- Client or server error causing the server to believe that the client has rebooted (i.e., receiving a SETCLIENTID with an nfs_client_id4 that has a matching id string and a non-matching boot instance id as the verifier). (May return either error.)
- Migration of all state under the associated lease causes its non-existence to be recognized on the source server. (Always returns NFS4ERR_STALE_CLIENTID.)
- Merger of state under the associated lease with another lease under a different client ID causes the clientid4 serving as the source of the merge to cease being recognized on its server. (Always returns NFS4ERR_STALE_CLIENTID.)

In the event of a server reboot, loss of lease state due to lease expiration, or administrative revocation of a clientid4, the client must obtain a new clientid4 by use of the SETCLIENTID operation and then proceed to any other necessary recovery for the server reboot case (see Section 9.6.2 in [RFC7530]). In cases of server or client error resulting in a clientid4 becoming unusable, use of SETCLIENTID to establish a new lease is desirable as well.

In cases in which loss of server knowledge of a clientid4 is the result of migration, different recovery procedures are required. See Section 6.1.1 for details. Note that in cases in which there is any uncertainty about which sort of handling is applicable, the

distinguishing characteristic is that in reboot-like cases, the clientid4 and all associated stateids cease to exist while in migration-related cases, the clientid4 ceases to exist while the stateids are still valid.

The client must also employ the SETCLIENTID operation when it receives an NFS4ERR_STALE_STATEID error using a stateid derived from its current clientid4, since this indicates a situation, such as a server reboot that has invalidated the existing clientid4 and associated stateids (see Section 9.1.5 in [RFC7530] for details).

See the detailed descriptions of SETCLIENTID (in Section 8.4) and SETCLIENTID_CONFIRM (in Section 16.34 of [RFC7530]) for a complete specification of these operations.

5.3. Server Release of Client ID

If the server determines that the client holds no associated state for its clientid4, the server may choose to release that clientid4. The server may make this choice for an inactive client so that resources are not consumed by those intermittently active clients. If the client contacts the server after this release, the server must ensure the client receives the appropriate error so that it will use the SETCLIENTID/SETCLIENTID_CONFIRM sequence to establish a new identity. It should be clear that the server must be very hesitant to release a client ID since the resulting work on the client to recover from such an event will be the same burden as if the server had failed and restarted. Typically, a server would not release a client ID unless there had been no activity from that client for many minutes.

Note that if the id string in a SETCLIENTID request is properly constructed, and if the client takes care to use the same principal for each successive use of SETCLIENTID, then, barring an active denial-of-service attack, NFS4ERR_CLID_INUSE should never be returned.

However, client bugs, server bugs, or perhaps a deliberate change of the principal owner of the id string (such as may occur in the case in which a client changes security flavors, and under the new flavor, there is no mapping to the previous owner) will in rare cases result in NFS4ERR_CLID_INUSE.

In situations in which there is an apparent change of principal, when the server gets a SETCLIENTID specifying a client ID string for which the server has a clientid4 that currently has no state, or for which it has state, but where the lease has expired, the server MUST allow the SETCLIENTID rather than returning NFS4ERR_CLID_INUSE. The server

MUST then confirm the new client ID if followed by the appropriate SETCLIENTID_CONFIRM.

5.4. Client ID String Approaches

One particular aspect of the construction of the nfs_client_id4 string has proved recurrently troublesome. The client has a choice of:

- Presenting the same id string to multiple server addresses. This is referred to as the "uniform client ID string approach" and is discussed in Section 5.6.

- Presenting different id strings to multiple server addresses. This is referred to as the "non-uniform client ID string approach" and is discussed in Section 5.5.

Note that implementation considerations, including compatibility with existing servers, may make it desirable for a client to use both approaches, based on configuration information, such as mount options. This issue will be discussed in Section 5.7.

Construction of the client ID string has arisen as a difficult issue because of the way in which the NFS protocols have evolved. It is useful to consider two points in that evolution.

- NFSv3 as a stateless protocol had no need to identify the state shared by a particular client-server pair (see [RFC1813]). Thus, there was no need to consider the question of whether a set of requests come from the same client or whether two server IP addresses are connected to the same server. As the environment was one in which the user supplied the target server IP address as part of incorporating the remote file system in the client's file namespace, there was no occasion to take note of server trunking. Within a stateless protocol, the situation was symmetrical. The client has no server identity information, and the server has no client identity information.

- NFSv4.1 is a stateful protocol with full support for client and server identity determination (see [RFC5661]). This enables the server to be aware when two requests come from the same client (they are on sessions sharing a clientid4) and the client to be aware when two server IP addresses are connected to the same server. Section 2.10.5.1 of [RFC5661] explains how the client is able to assure itself that the connections are to the same logical server.

NFSv4.0 is unfortunately halfway between these two. It introduced new requirements such as the need to identify specific clients and client instances without addressing server identity issues. The two client ID string approaches have arisen in attempts to deal with the changing requirements of the protocol as implementation has proceeded, and features that were not very substantial in early implementations of NFSv4.0 became more substantial as implementation proceeded.

- In the absence of any implementation of features related to fs_locations (replication, referral, and migration), the situation is very similar to that of NFSv3 (see Section 8.1 and the subsections within Section 8.4 of [RFC7530] for discussion of these features). In this case, locking state has been added, but there is no need for concern about the provision of accurate client and server identity determination. This is the situation that gave rise to the non-uniform client ID string approach.

- In the presence of replication and referrals, the client may have occasion to take advantage of knowledge of server trunking information. Even more important, transparent state migration, by transferring state among servers, causes difficulties for the non-uniform client ID string approach, in that the two different client ID strings sent to different IP addresses may wind up being processed by the same logical server, adding confusion.

- A further consideration is that client implementations typically provide NFSv4.1 by augmenting their existing NFSv4.0 implementation, not by providing two separate implementations. Thus, the more NFSv4.0 and NFSv4.1 can work alike, the less complex the clients are. This is a key reason why those implementing NFSv4.0 clients might prefer using the uniform client string model, even if they have chosen not to provide fs_locations-related features in their NFSv4.0 client.

Both approaches have to deal with the asymmetry in client and server identity information between client and server. Each seeks to make the client's and the server's views match. In the process, each encounters some combination of inelegant protocol features and/or implementation difficulties. The choice of which to use is up to the client implementer, and the sections below try to give some useful guidance.

5.5. Non-uniform Client ID String Approach

The non-uniform client ID string approach is an attempt to handle these matters in NFSv4.0 client implementations in as NFSv3-like a way as possible.

For a client using the non-uniform approach, all internal recording of clientid4 values is to include, whether explicitly or implicitly, the server IP address so that one always has an <IP-address, clientid4> pair. Two such pairs from different servers are always distinct even when the clientid4 values are the same, as they may occasionally be. In this approach, such equality is always treated as simple happenstance.

Making the client ID string different on different server IP addresses results in a situation in which a server has no way of tying together information from the same client, when the client accesses multiple server IP addresses. As a result, it will treat a single client as multiple clients with separate leases for each server network address. Since there is no way in the protocol for the client to determine if two network addresses are connected to the same server, the resulting lack of knowledge is symmetrical and can result in simpler client implementations in which there is a single clientid4/lease per server network address.

Support for migration, particularly with transparent state migration, is more complex in the case of non-uniform client ID strings. For example, migration of a lease can result in multiple leases for the same client accessing the same server addresses, vitiating many of the advantages of this approach. Therefore, client implementations that support migration with transparent state migration are likely to experience difficulties using the non-uniform client ID string approach and should not do so, except where it is necessary for compatibility with existing server implementations (for details of arranging use of multiple client ID string approaches, see Section 5.7).

5.6. Uniform Client ID String Approach

When the client ID string is kept uniform, the server has the basis to have a single clientid4/lease for each distinct client. The problem that has to be addressed is the lack of explicit server identity information, which was made available in NFSv4.1.

When the same client ID string is given to multiple IP addresses, the client can determine whether two IP addresses correspond to a single server, based on the server's behavior. This is the inverse of the strategy adopted for the non-uniform approach in which different

server IP addresses are told about different clients, simply to prevent a server from manifesting behavior that is inconsistent with there being a single server for each IP address, in line with the traditions of NFS. So, to compare:

- In the non-uniform approach, servers are told about different clients because, if the server were to use accurate client identity information, two IP addresses on the same server would behave as if they were talking to the same client, which might prove disconcerting to a client not expecting such behavior.

- In the uniform approach, the servers are told about there being a single client, which is, after all, the truth. Then, when the server uses this information, two IP addresses on the same server will behave as if they are talking to the same client, and this difference in behavior allows the client to infer the server IP address trunking configuration, even though NFSv4.0 does not explicitly provide this information.

  The approach given in the section below shows one example of how this might be done.

The uniform client ID string approach makes it necessary to exercise more care in the definition of the boot instance id sent as the verifier field in an nfs_client_id4:

- In [RFC7530], the client is told to change the verifier field value when reboot occurs, but there is no explicit statement as to the converse, so that any requirement to keep the verifier field constant unless rebooting is only present by implication.

- Many existing clients change the boot instance id every time they destroy and recreate the data structure that tracks an <IP-address, clientid4> pair. This might happen if the last mount of a particular server is removed, and then a fresh mount is created. Also, note that this might result in each <IP-address, clientid4> pair having its own boot instance id that is independent of the others.

- Within the uniform client ID string approach, an nfs_client_id4 designates a globally known client instance, so that the verifier field should change if and only if a new client instance is created, typically as a result of a reboot.

  Clients using the uniform client ID string approach are therefore well advised to use a verifier established only once for each reboot, typically at reboot time.

The following are advantages for the implementation of using the uniform client ID string approach:

- Clients can take advantage of server trunking (and clustering with single-server-equivalent semantics) to increase bandwidth or reliability.

- There are advantages in state management so that, for example, one never has a delegation under one clientid4 revoked because of a reference to the same file from the same client under a different clientid4.

- The uniform client ID string approach allows the server to do any necessary automatic lease merger in connection with transparent state migration, without requiring any client involvement. This consideration is of sufficient weight to cause us to recommend use of the uniform client ID string approach for clients supporting transparent state migration.

The following implementation considerations might cause issues for client implementations.

- This approach is considerably different from the non-uniform approach, which most client implementations have been following. Until substantial implementation experience is obtained with this approach, reluctance to embrace something so new is to be expected.

- Mapping between server network addresses and leases is more complicated in that it is no longer a one-to-one mapping.

Another set of relevant considerations relate to privacy concerns, which users of the client might have in that use of the uniform client ID string approach would enable multiple servers acting in concert to determine when multiple requests received at different times derive from the same NFSv4.0 client. For example, this might enable determination that multiple distinct user identities in fact are likely to correspond to requests made by the same person, even when those requests are directed to different servers.

How to balance these considerations depends on implementation goals.

5.7. Mixing Client ID String Approaches

As noted above, a client that needs to use the uniform client ID string approach (e.g., to support migration) may also need to support existing servers with implementations that do not work properly in this case.

Some examples of such server issues include:

- Some existing NFSv4.0 server implementations of IP address failover depend on clients' use of a non-uniform client ID string approach. In particular, when a server supports both its own IP address and one failed over from a partner server, it may have separate sets of state applicable to the two IP addresses, owned by different servers but residing on a single one.

  In this situation, some servers have relied on clients' use of the non-uniform client ID string approach, as suggested but not mandated by [RFC7530], to keep these sets of state separate, and they will have problems handling clients using the uniform client ID string approach, in that such clients will see changes in trunking relationships whenever server failover and giveback occur.

- Some existing servers incorrectly return NFS4ERR_CLID_INUSE simply because there already exists a clientid4 for the same client, established using a different IP address. This causes difficulty for a multihomed client using the uniform client ID string approach.

  Although this behavior is not correct, such servers still exist, and this specification should give clients guidance about dealing with the situation, as well as making the correct behavior clear.

In order to support use of these sorts of servers, the client can use different client ID string approaches for different mounts, in order to assure that:

- The uniform client ID string approach is used when accessing servers that may return NFS4ERR_MOVED and when the client wishes to enable transparent state migration.

- The non-uniform client ID string approach is used when accessing servers whose implementations make them incompatible with the uniform client ID string approach.

Since the client cannot easily determine which of the above are true, implementations are likely to rely on user-specified mount options to select the appropriate approach to use, in cases in which a client supports simultaneous use of multiple approaches. Choice of a default to use in such cases is up to the client implementation.

In the case in which the same server has multiple mounts, and both approaches are specified for the same server, the client could have multiple clientid4s corresponding to the same server, one for each approach, and would then have to keep these separate.

5.8. Trunking Determination when Using Uniform Client ID Strings

This section provides an example of how trunking determination could be done by a client following the uniform client ID string approach (whether this is used for all mounts or not). Clients need not follow this procedure, but implementers should make sure that the issues dealt with by this procedure are all properly addressed.

It is best to clarify here the various possible purposes of trunking determination and the corresponding requirements as to server behavior. The following points should be noted:

- The primary purpose of the trunking determination algorithm is to make sure that, if the server treats client requests on two IP addresses as part of the same client, the client will not be surprised and encounter disconcerting server behavior, as mentioned in Section 5.6. Such behavior could occur if the client were unaware that all of its client requests for the two IP addresses were being handled as part of a single client talking to a single server.

- A second purpose is to be able to use knowledge of trunking relationships for better performance, etc.

- If a server were to give out distinct clientid4s in response to receiving the same nfs_client_id4 on different network addresses, and acted as if these were separate clients, the primary purpose of trunking determination would be met, as long as the server did not treat them as part of the same client. In this case, the server would be acting, with regard to that client, as if it were two distinct servers. This would interfere with the secondary purpose of trunking determination, but there is nothing the client can do about that.

- Suppose a server were to give such a client two different clientid4s but act as if they were one. That is the only way that the server could behave in a way that would defeat the primary purpose of the trunking determination algorithm.

  Servers MUST NOT behave that way.

For a client using the uniform approach, clientid4 values are treated as important information in determining server trunking patterns.

For two different IP addresses to return the same clientid4 value is a necessary, though not a sufficient condition for them to be considered as connected to the same server. As a result, when two different IP addresses return the same clientid4, the client needs to determine, using the procedure given below or otherwise, whether the IP addresses are connected to the same server. For such clients, all internal recording of clientid4 values needs to include, whether explicitly or implicitly, identification of the server from which the clientid4 was received so that one always has a (server, clientid4) pair. Two such pairs from different servers are always considered distinct even when the clientid4 values are the same, as they may occasionally be.

In order to make this approach work, the client must have certain information accessible for each nfs_client_id4 used by the uniform approach (only one in general). The client needs to maintain a list of all server IP addresses, together with the associated clientid4 values, SETCLIENTID principals, and authentication flavors. As a part of the associated data structures, there should be the ability to mark a server IP structure as having the same server as another and to mark an IP address as currently unresolved. One way to do this is to allow each such entry to point to another with the pointer value being one of:

- A pointer to another entry for an IP address associated with the same server, where that IP address is the first one referenced to access that server.

- A pointer to the current entry if there is no earlier IP address associated with the same server, i.e., where the current IP address is the first one referenced to access that server. The text below refers to such an IP address as the lead IP address for a given server.

- The value NULL if the address's server identity is currently unresolved.

In order to keep the above information current, in the interests of the most effective trunking determination, RENEWs should be periodically done on each server. However, even if this is not done, the primary purpose of the trunking determination algorithm, to prevent confusion due to trunking hidden from the client, will be achieved.

Given this apparatus, when a SETCLIENTID is done and a clientid4 returned, the data structure can be searched for a matching clientid4 and if such is found, further processing can be done to determine whether the clientid4 match is accidental, or the result of trunking.

In this algorithm, when SETCLIENTID is done initially, it will use the common nfs_client_id4 and specify the current target IP address as callback.cb_location within the callback parameters. We call the clientid4 and SETCLIENTID verifier returned by this operation XC and XV, respectively.

This choice of callback parameters is provisional and reflects the client's preferences in the event that the IP address is not trunked with other IP addresses. The algorithm is constructed so that only the appropriate callback parameters, reflecting observed trunking patterns, are actually confirmed.

Note that when the client has done previous SETCLIENTIDs to any IP addresses, with more than one principal or authentication flavor, one has the possibility of receiving NFS4ERR_CLID_INUSE, since it is not yet known which of the connections with existing IP addresses might be trunked with the current one. In the event that the SETCLIENTID fails with NFS4ERR_CLID_INUSE, one must try all other combinations of principals and authentication flavors currently in use, and eventually one will be correct and not return NFS4ERR_CLID_INUSE.

Note that at this point, no SETCLIENTID_CONFIRM has yet been done. This is because the SETCLIENTID just done has either established a new clientid4 on a previously unknown server or changed the callback parameters on a clientid4 associated with some already known server. Given it is undesirable to confirm something that should not happen, what is to be done next depends on information about existing clientid4s.

- If no matching clientid4 is found, the IP address X and clientid4 XC are added to the list and considered as having no existing known IP addresses trunked with it. The IP address is marked as a lead IP address for a new server. A SETCLIENTID_CONFIRM is done using XC and XV.

- If a matching clientid4 is found that is marked unresolved, processing on the new IP address is suspended. In order to simplify processing, there can only be one unresolved IP address for any given clientid4.

- If one or more matching clientid4s are found, none of which are marked unresolved, the new IP address X is entered and marked unresolved. A SETCLIENTID_CONFIRM is done to X using XC and XV.

When, as a result of encountering the last of the three cases shown above, an unresolved IP address exists, further processing is required. After applying the steps below to each of the lead IP addresses with a matching clientid4, the address will have been

resolved: It may have been determined to be part of an already known server as a new IP address to be added to an existing set of IP addresses for that server. Otherwise, it will be recognized as a new server. At the point at which this determination is made, the unresolved indication is cleared and any suspended SETCLIENTID processing is restarted.

For each lead IP address IPn with a clientid4 matching XC, the following steps are done. Because the Remote Procedure Call (RPC) to do a SETCLIENTID could take considerable time, it is desirable for the client to perform these operations in parallel. Note that because the clientid4 is a 64-bit value, the number of such IP addresses that would need to be tested is expected to be quite small, even when the client is interacting with many NFSv4.0 servers. Thus, while parallel processing is desirable, it is not necessary.

- If the principal for IPn does not match that for X, the IP address is skipped, since it is impossible for IPn and X to be trunked in these circumstances. If the principal does match but the authentication flavor does not, the authentication flavor already used should be used for address X as well. This will avoid any possibility that NFS4ERR_CLID_INUSE will be returned for the SETCLIENTID and SETCLIENTID_CONFIRM to be done below, as long as the server(s) at IP addresses IPn and X is correctly implemented.

- A SETCLIENTID is done to update the callback parameters to reflect the possibility that X will be marked as associated with the server whose lead IP address is IPn. The specific callback parameters chosen, in terms of cb_client4 and callback_ident, are up to the client and should reflect its preferences as to callback handling for the common clientid4, in the event that X and IPn are trunked together. When a SETCLIENTID is done on IP address IPn, a setclientid_confirm value (in the form of a verifier4) is returned, which will be referred to as SCn.

  Note that the NFSv4.0 specification requires the server to make sure that such verifiers are very unlikely to be regenerated. Given that it is already highly unlikely that the clientid4 XC is duplicated by distinct servers, the probability that SCn is duplicated as well has to be considered vanishingly small. Note also that the callback update procedure can be repeated multiple times to reduce the probability of further spurious matches.

- The setclientid_confirm value SCn is saved for later use in confirming the SETCLIENTID done to IPn.

Once the SCn values are gathered up by the procedure above, they are each tested by being used as the verifier for a SETCLIENTID_CONFIRM operation directed to the original IP address X, whose trunking relationships are to be determined. These RPC operations may be done in parallel.

There are a number of things that should be noted at this point.

- The SETCLIENTID operations done on the various IPn addresses in the procedure above will never be confirmed by SETCLIENTID_CONFIRM operations directed to the various IPn addresses. If these callback updates are to be confirmed, they will be confirmed by SETCLIENTID_CONFIRM operations directed at the original IP address X, which can only happen if SCn was generated by an IPn that was trunked with X, allowing the SETCLIENTID to be successfully confirmed and allowing us to infer the existence of that trunking relationship.

- The number of successful SETCLIENTID_CONFIRM operations done should never be more than one. If both SCn and SCm are accepted by X, then it indicates that both IPn and IPm are trunked with X, but that is only possible if IPn and IPm are trunked together. Since these two addresses were earlier recognized as not trunked together, this should be impossible, if the servers in question are implemented correctly.

Further processing depends on the success or failure of the various SETCLIENTD_CONFIRM operations done in the step above.

- If the setclientid_confirm value generated by a particular IPn is accepted on X, then X and IPn are recognized as connected to the same server, and the entry for X is marked as associated with IPn.

- If none of the confirm operations are accepted, then X is recognized as a distinct server. Its callback parameters will remain as the ones established by the original SETCLIENTID.

In either of the cases, the entry is considered resolved and processing can be restarted for IP addresses whose clientid4 matched XC but whose resolution had been deferred.

The procedure described above must be performed so as to exclude the possibility that multiple SETCLIENTIDs done to different server IP addresses and returning the same clientid4 might "race" in such a fashion that there is no explicit determination of whether they correspond to the same server. The following possibilities for serialization are all valid, and implementers may choose among them

based on a tradeoff between performance and complexity. They are listed in order of increasing parallelism:

- An NFSv4.0 client might serialize all instances of SETCLIENTID/SETCLIENTID_CONFIRM processing, either directly or by serializing mount operations involving use of NFSv4.0. While doing so will prevent the races mentioned above, this degree of serialization can cause performance issues when there is a high volume of mount operations.

- One might instead serialize the period of processing that begins when the clientid4 received from the server is processed and ends when all trunking determination for that server is completed. This prevents the races mentioned above, without adding to delay except when trunking determination is common.

- One might avoid much of the serialization implied above, by allowing trunking determination for distinct clientid4 values to happen in parallel, with serialization of trunking determination happening independently for each distinct clientid4 value.

The procedure above has made no explicit mention of the possibility that server reboot can occur at any time. To address this possibility, the client should make sure the following steps are taken:

- When a SETCLIENTID_CONFIRM is rejected by a given IPn, the client should be aware of the possibility that the rejection is due to XC (rather than XV) being invalid. This situation can be addressed by doing a RENEW specifying XC directed to the IP address X. If that operation succeeds, then the rejection is to be acted on normally since either XV is invalid on IPn or XC has become invalid on IPn while it is valid on X, showing that IPn and X are not trunked. If, on the other hand, XC is not valid on X, then the trunking detection process should be restarted once a new client ID is established on X.

- In the event of a reboot detected on any server-lead IP, the set of IP addresses associated with the server should not change, and state should be re-established for the lease as a whole, using all available connected server IP addresses. It is prudent to verify connectivity by doing a RENEW using the new clientid4 on each such server address before using it, however.

Another situation not discussed explicitly above is the possibility that a SETCLIENTID done to one of the IPn addresses might take so long that it is necessary to time out the operation, to prevent unacceptably delaying the MOUNT operation. One simple possibility is

to simply fail the MOUNT at this point. Because the average number of IP addresses that might have to be tested is quite small, this will not greatly increase the probability of MOUNT failure. Other possible approaches are:

- If the IPn has sufficient state in existence, the existing stateids and sequence values might be validated by being used on IP address X. In the event of success, X and IPn should be considered trunked together.

  What constitutes "sufficient" state in this context is an implementation decision that is affected by the implementer's willingness to fail the MOUNT in an uncertain case and the strength of the state verification procedure implemented.

- If IPn has no locking state in existence, X could be recorded as a lead IP address on a provisional basis, subject to trunking being tested again, once IPn starts becoming responsive. To avoid confusion between IPn and X, and the need to merge distinct state corpora for X and IPn at a later point, this retest of trunking should occur after RENEWs on IPn are responded to and before establishing any new state for either IPn as a separate server or for IPn considered as a server address trunked with X.

- The client locking-related code could be made more tolerant of what would otherwise be considered anomalous results due to an unrecognized trunking relationship. The client could use the appearance of behavior explainable by a previously unknown trunking relationship as the cue to consider the addresses as trunked.

  This choice has a lot of complexity associated with it, and it is likely that few implementations will use it. When the set of locking state on IPn is small (e.g., a single stateid) but not empty, most client implementations are likely to either fail the MOUNT or implement a more stringent verification procedure using the existing stateid on IPn as a basis to generate further state as raw material for the trunking verification process.

5.9. Client ID String Construction Details

This section gives more detailed guidance on client ID string construction. The guidance in this section will cover cases in which either the uniform or the non-uniform approach to the client ID string is used.

Note that among the items suggested for inclusion, there are many that may conceivably change. In order for the client ID string to remain valid in such circumstances, the client SHOULD either:

- Use a saved copy of such value rather than the changeable value itself, or
- Save the constructed client ID string rather than constructing it anew at SETCLIENTID time, based on unchangeable parameters and saved copies of changeable data items.

A file is not always a valid choice to store such information, given the existence of diskless clients. In such situations, whatever facilities exist for a client to store configuration information such as boot arguments should be used.

Given the considerations listed in Section 5.2.1, an id string would be one that includes as its basis:

- An identifier uniquely associated with the node on which the client is running.
- For a user-level NFSv4.0 client, it should contain additional information to distinguish the client from a kernel-based client and from other user-level clients running on the same node, such as a universally unique identifier (UUID).
- Where the non-uniform approach is to be used, the IP address of the server.
- Additional information that tends to be unique, such as one or more of:
  - The timestamp of when the NFSv4 software was first installed on the client (though this is subject to the previously mentioned caution about using information that is stored in a file, because the file might only be accessible over NFSv4).
  - A true random number, generally established once and saved.

With regard to the identifier associated with the node on which the client is running, the following possibilities are likely candidates.

- The client machine's serial number.
- The client's IP address. Note that this SHOULD be treated as a changeable value.

- A Media Access Control (MAC) address. Note that this also should be considered a changeable value because of the possibility of configuration changes.

Privacy concerns may be an issue if some of the items above (e.g., machine serial number and MAC address) are used. When it is necessary to use such items to ensure uniqueness, application of a one-way hash function is desirable. When the non-uniform approach is used, that hash function should be applied to all of the components chosen as a unit rather than to particular individual elements.

6. Locking and Multi-Server Namespace

This section contains a replacement for Section 9.14 of [RFC7530], "Migration, Replication, and State".

The replacement is in Section 6.1 and supersedes the replaced section.

The changes made can be briefly summarized as follows:

- Adding text to address the case of stateid conflict on migration.
- Specifying that when leases are moved, as a result of file system migration, they are to be merged with leases on the destination server that are connected to the same client.
- Adding text that deals with the case of a clientid4 being changed on state transfer as a result of conflict with an existing clientid4.
- Adding a section describing how information associated with open-owners and lock-owners is to be managed with regard to migration.
- The description of handling of the NFS4ERR_LEASE_MOVED has been rewritten for greater clarity.

6.1. Lock State and File System Transitions

File systems may transition to a different server in several circumstances:

- Responsibility for handling a given file system is transferred to a new server via migration.
- A client may choose to use an alternate server (e.g., in response to server unresponsiveness) in the context of file system replication.

In such cases, the appropriate handling of state shared between the client and server (i.e., locks, leases, stateids, and client IDs) is as described below. The handling differs between migration and replication.

If a server replica or a server immigrating a file system agrees to, or is expected to, accept opaque values from the client that originated from another server, then it is a wise implementation practice for the servers to encode the "opaque" values in network byte order (i.e., in a big-endian format). When doing so, servers acting as replicas or immigrating file systems will be able to parse values like stateids, directory cookies, filehandles, etc., even if their native byte order is different from that of other servers cooperating in the replication and migration of the file system.

6.1.1. Migration and State

In the case of migration, the servers involved in the migration of a file system should transfer all server state associated with the migrating file system from source to the destination server. If state is transferred, this MUST be done in a way that is transparent to the client. This state transfer will ease the client's transition when a file system migration occurs. If the servers are successful in transferring all state, the client will continue to use stateids assigned by the original server. Therefore, the new server must recognize these stateids as valid and treat them as representing the same locks as they did on the source server.

In this context, the phrase "the same locks" means that:

- They are associated with the same file.
- They represent the same types of locks, whether opens, delegations, advisory byte-range locks, or mandatory byte-range locks.
- They have the same lock particulars, including such things as access modes, deny modes, and byte ranges.
- They are associated with the same owner string(s).

If transferring stateids from server to server would result in a conflict for an existing stateid for the destination server with the existing client, transparent state migration MUST NOT happen for that client. Servers participating in using transparent state migration should coordinate their stateid assignment policies to make this situation unlikely or impossible. The means by which this might be done, like all of the inter-server interactions for migration, are

not specified by the NFS version 4.0 protocol (neither in [RFC7530] nor this update).

A client may determine the disposition of migrated state by using a stateid associated with the migrated state on the new server.

- If the stateid is not valid and an error NFS4ERR_BAD_STATEID is received, either transparent state migration has not occurred or the state was purged due to a mismatch in the verifier (i.e., the boot instance id).

- If the stateid is valid, transparent state migration has occurred.

Since responsibility for an entire file system is transferred with a migration event, there is no possibility that conflicts will arise on the destination server as a result of the transfer of locks.

The servers may choose not to transfer the state information upon migration. However, this choice is discouraged, except where specific issues such as stateid conflicts make it necessary. When a server implements migration and it does not transfer state information, it MUST provide a file-system-specific grace period, to allow clients to reclaim locks associated with files in the migrated file system. If it did not do so, clients would have to re-obtain locks, with no assurance that a conflicting lock was not granted after the file system was migrated and before the lock was re-obtained.

In the case of migration without state transfer, when the client presents state information from the original server (e.g., in a RENEW operation or a READ operation of zero length), the client must be prepared to receive either NFS4ERR_STALE_CLIENTID or NFS4ERR_BAD_STATEID from the new server. The client should then recover its state information as it normally would in response to a server failure. The new server must take care to allow for the recovery of state information as it would in the event of server restart.

In those situations in which state has not been transferred, as shown by a return of NFS4ERR_BAD_STATEID, the client may attempt to reclaim locks in order to take advantage of cases in which the destination server has set up a file-system-specific grace period in support of the migration.

6.1.1.1. Migration and Client IDs

The handling of clientid4 values is similar to that for stateids. However, there are some differences that derive from the fact that a clientid4 is an object that spans multiple file systems while a stateid is inherently limited to a single file system.

The clientid4 and nfs_client_id4 information (id string and boot instance id) will be transferred with the rest of the state information, and the destination server should use that information to determine appropriate clientid4 handling. Although the destination server may make state stored under an existing lease available under the clientid4 used on the source server, the client should not assume that this is always so. In particular,

- If there is an existing lease with an nfs_client_id4 that matches a migrated lease (same id string and verifier), the server SHOULD merge the two, making the union of the sets of stateids available under the clientid4 for the existing lease. As part of the lease merger, the expiration time of the lease will reflect renewal done within either of the ancestor leases (and so will reflect the latest of the renewals).

- If there is an existing lease with an nfs_client_id4 that partially matches a migrated lease (same id string and a different (boot) verifier), the server MUST eliminate one of the two, possibly invalidating one of the ancestor clientid4s. Since boot instance ids are not ordered, the later lease renewal time will prevail.

- If the destination server already has the transferred clientid4 in use for another purpose, it is free to substitute a different clientid4 and associate that with the transferred nfs_client_id4.

When leases are not merged, the transfer of state should result in creation of a confirmed client record with empty callback information but matching the {v, x, c} with v and x derived from the transferred client information and c chosen by the destination server. For a description of this notation, see Section 8.4.5

In such cases, the client SHOULD re-establish new callback information with the new server as soon as possible, according to sequences described in sections "Operation 35: SETCLIENTID -- Negotiate Client ID" and "Operation 36: SETCLIENTID_CONFIRM -- Confirm Client ID". This ensures that server operations are not delayed due to an inability to recall delegations and prevents the

unwanted revocation of existing delegations. The client can determine the new clientid4 (the value c) from the response to SETCLIENTID.

The client can use its own information about leases with the destination server to see if lease merger should have happened. When there is any ambiguity, the client MAY use the above procedure to set the proper callback information and find out, as part of the process, the correct value of its clientid4 with respect to the server in question.

6.1.1.2. Migration and State Owner Information

In addition to stateids, the locks they represent, and client identity information, servers also need to transfer information related to the current status of open-owners and lock-owners.

This information includes:

- The sequence number of the last operation associated with the particular owner.
- Sufficient information regarding the results of the last operation to allow reissued operations to be correctly responded to.

When individual open-owners and lock-owners have only been used in connection with a particular file system, the server SHOULD transfer this information together with the lock state. The owner ceases to exist on the source server and is reconstituted on the destination server. This will happen in the case of clients that have been written to isolate each owner to a specific file system, but it may happen for other clients as well.

Note that when servers take this approach for all owners whose state is limited to the particular file system being migrated, doing so will not cause difficulties for clients not adhering to an approach in which owners are isolated to particular file systems. As long as the client recognizes the loss of transferred state, the protocol allows the owner in question to disappear, and the client may have to deal with an owner confirmation request that would not have occurred in the absence of the migration.

When migration occurs and the source server discovers an owner whose state includes the migrated file system but other file systems as well, it cannot transfer the associated owner state. Instead, the

existing owner state stays in place, but propagation of owner state is done as specified below:

- When the current seqid for an owner represents an operation associated with the file system being migrated, owner status SHOULD be propagated to the destination file system.

- When the current seqid for an owner does not represent an operation associated with the file system being migrated, owner status MAY be propagated to the destination file system.

- When the owner in question has never been used for an operation involving the migrated file system, the owner information SHOULD NOT be propagated to the destination file system.

Note that a server may obey all of the conditions above without the overhead of keeping track of a set of file systems that any particular owner has been associated with. Consider a situation in which the source server has decided to keep lock-related state associated with a file system fixed, preparatory to propagating it to the destination file system. If a client is free to create new locks associated with existing owners on other file systems, the owner information may be propagated to the destination file system, even though, at the time the file system migration is recognized by the client to have occurred, the last operation associated with the owner may not be associated with the migrating file system.

When a source server propagates owner-related state associated with owners that span multiple file systems, it will propagate the owner sequence value to the destination server, while retaining it on the source server, as long as there exists state associated with the owner. When owner information is propagated in this way, source and destination servers start with the same owner sequence value that is then updated independently, as the client makes owner-related requests to the servers. Note that each server will have some period in which the associated sequence value for an owner is identical to the one transferred as part of migration. At those times, when a server receives a request with a matching owner sequence value, it MUST NOT respond with the associated stored response if the associated file system is not, when the reissued request is received, part of the set of file systems handled by that server.

One sort of case may require more complex handling. When multiple file systems are migrated, in sequence, to a specific destination server, an owner may be migrated to a destination server, on which it was already present, leading to the issue of how the resident owner information and that being newly migrated are to be reconciled.

If file system migration encounters a situation where owner information needs to be merged, it MAY decline to transfer such state, even if it chooses to handle other cases in which locks for a given owner are spread among multiple file systems.

As a way of understanding the situations that need to be addressed when owner information needs to be merged, consider the following scenario:

- There is client C and two servers, X and Y. There are two clientid4s designating C, which are referred to as CX and CY.
- Initially, server X supports file systems F1, F2, F3, and F4. These will be migrated, one at a time, to server Y.
- While these migrations are proceeding, the client makes locking requests for file systems F1 through F4 on behalf of owner O (either a lock-owner or an open-owner), with each request going to X or Y depending on where the relevant file system is being supported at the time the request is made.
- Once the first migration event occurs, client C will maintain two instances for owner O, one for each server.
- It is always possible that C may make a request of server X relating to owner O, and before receiving a response, it finds the target file system has moved to Y and needs to reissue the request to server Y.
- At the same time, C may make a request of server Y relating to owner O, and this too may encounter a lost-response situation.

As a result of such merger situations, the server will need to provide support for dealing with retransmission of owner-sequenced requests that diverge from the typical model in which there is support for retransmission of replies only for a request whose sequence value exactly matches the last one sent. In some situations, there may be two requests, each of which had the last sequence when it was issued. As a result of migration and owner merger, one of those will no longer be the last by sequence.

When servers do support such merger of owner information on the destination server, the following rules are to be adhered to:

- When an owner sequence value is propagated to a destination server where it already exists, the resulting sequence value is to be the greater of the one present on the destination server and the one being propagated as part of migration.

- In the event that an owner sequence value on a server represents a request applying to a file system currently present on the server, it is not to be rendered invalid simply because that sequence value is changed as a result of owner information propagation as part of file system migration. Instead, it is retained until it can be deduced that the client in question has received the reply.

As a result of the operation of these rules, there are three ways in which there can be more reply data than what is typically present, i.e., data for a single request per owner whose sequence is the last one received, where the next sequence to be used is one beyond that.

- When the owner sequence value for a migrating file system is greater than the corresponding value on the destination server, the last request for the owner in effect at the destination server needs to be retained, even though it is no longer one less than the next sequence to be received.

- When the owner sequence value for a migrating file system is less than the corresponding value on the destination server, the sequence number for last request for the owner in effect on the migrating file system needs to be retained, even though it is no longer than one less the next sequence to be received.

- When the owner sequence value for a migrating file system is equal to the corresponding value on the destination server, one has two different "last" requests that both must be retained. The next sequence value to be used is one beyond the sequence value shared by these two requests.

Here are some guidelines as to when servers can drop such additional reply data, which is created as part of owner information migration.

- The server SHOULD NOT drop this information simply because it receives a new sequence value for the owner in question, since that request may have been issued before the client was aware of the migration event.

- The server SHOULD drop this information if it receives a new sequence value for the owner in question, and the request relates to the same file system.

- The server SHOULD drop the part of this information that relates to non-migrated file systems if it receives a new sequence value for the owner in question, and the request relates to a non-migrated file system.

- The server MAY drop this information when it receives a new sequence value for the owner in question for a considerable period of time (more than one or two lease periods) after the migration occurs.

6.1.2. Replication and State

Since client switch-over in the case of replication is not under server control, the handling of state is different. In this case, leases, stateids, and client IDs do not have validity across a transition from one server to another. The client must re-establish its locks on the new server. This can be compared to the re-establishment of locks by means of reclaim-type requests after a server reboot. The difference is that the server has no provision to distinguish requests reclaiming locks from those obtaining new locks or to defer the latter. Thus, a client re-establishing a lock on the new server (by means of a LOCK or OPEN request) may have the requests denied due to a conflicting lock. Since replication is intended for read-only use of file systems, such denial of locks should not pose large difficulties in practice. When an attempt to re-establish a lock on a new server is denied, the client should treat the situation as if its original lock had been revoked.

6.1.3. Notification of Migrated Lease

A file system can be migrated to another server while a client that has state related to that file system is not actively submitting requests to it. In this case, the migration is reported to the client during lease renewal. Lease renewal can occur either explicitly via a RENEW operation or implicitly when the client performs a lease-renewing operation on another file system on that server.

In order for the client to schedule renewal of leases that may have been relocated to the new server, the client must find out about lease relocation before those leases expire. Similarly, when migration occurs but there has not been transparent state migration, the client needs to find out about the change soon enough to be able to reclaim the lock within the destination server's grace period. To accomplish this, all operations that implicitly renew leases for a client (such as OPEN, CLOSE, READ, WRITE, RENEW, LOCK, and others) will return the error NFS4ERR_LEASE_MOVED if responsibility for any of the leases to be renewed has been transferred to a new server. Note that when the transfer of responsibility leaves remaining state for that lease on the source server, the lease is renewed just as it would have been in the NFS4ERR_OK case, despite returning the error. The transfer of responsibility happens when the server receives a GETATTR(fs_locations) from the client for each file system for which

a lease has been moved to a new server. Normally, it does this after receiving an NFS4ERR_MOVED for an access to the file system, but the server is not required to verify that this happens in order to terminate the return of NFS4ERR_LEASE_MOVED. By convention, the compounds containing GETATTR(fs_locations) SHOULD include an appended RENEW operation to permit the server to identify the client getting the information.

Note that the NFS4ERR_LEASE_MOVED error is required only when responsibility for at least one stateid has been affected. In the case of a null lease, where the only associated state is a clientid4, an NFS4ERR_LEASE_MOVED error SHOULD NOT be generated.

Upon receiving the NFS4ERR_LEASE_MOVED error, a client that supports file system migration MUST perform the necessary GETATTR operation for each of the file systems containing state that have been migrated, so it gives the server evidence that it is aware of the migration of the file system. Once the client has done this for all migrated file systems on which the client holds state, the server MUST resume normal handling of stateful requests from that client.

One way in which clients can do this efficiently in the presence of large numbers of file systems is described below. This approach divides the process into two phases: one devoted to finding the migrated file systems, and the second devoted to doing the necessary GETATTRs.

The client can find the migrated file systems by building and issuing one or more COMPOUND requests, each consisting of a set of PUTFH/GETFH pairs, each pair using a filehandle in one of the file systems in question. All such COMPOUND requests can be done in parallel. The successful completion of such a request indicates that none of the file systems interrogated have been migrated while termination with NFS4ERR_MOVED indicates that the file system getting the error has migrated while those interrogated before it in the same COMPOUND have not. Those whose interrogation follows the error remain in an uncertain state and can be interrogated by restarting the requests from after the point at which NFS4ERR_MOVED was returned or by issuing a new set of COMPOUND requests for the file systems that remain in an uncertain state.

Once the migrated file systems have been found, all that is needed is for the client to give evidence to the server that it is aware of the migrated status of file systems found by this process, by interrogating the fs_locations attribute for a filehandle within each of the migrated file systems. The client can do this by building and issuing one or more COMPOUND requests, each of which consists of a set of PUTFH operations, each followed by a GETATTR of the

fs_locations attribute. A RENEW is necessary to enable the operations to be associated with the lease returning NFS4ERR_LEASE_MOVED. Once the client has done this for all migrated file systems on which the client holds state, the server will resume normal handling of stateful requests from that client.

In order to support legacy clients that do not handle the NFS4ERR_LEASE_MOVED error correctly, the server SHOULD time out after a wait of at least two lease periods, at which time it will resume normal handling of stateful requests from all clients. If a client attempts to access the migrated files, the server MUST reply with NFS4ERR_MOVED. In this situation, it is likely that the client would find its lease expired, although a server may use "courtesy" locks (as described in Section 9.6.3.1 of [RFC7530]) to mitigate the issue.

When the client receives an NFS4ERR_MOVED error, the client can follow the normal process to obtain the destination server information (through the fs_locations attribute) and perform renewal of those leases on the new server. If the server has not had state transferred to it transparently, the client will receive either NFS4ERR_STALE_CLIENTID or NFS4ERR_STALE_STATEID from the new server, as described above. The client can then recover state information as it does in the event of server failure.

Aside from recovering from a migration, there are other reasons a client may wish to retrieve fs_locations information from a server. When a server becomes unresponsive, for example, a client may use cached fs_locations data to discover an alternate server hosting the same file system data. A client may periodically request fs_locations data from a server in order to keep its cache of fs_locations data fresh.

Since a GETATTR(fs_locations) operation would be used for refreshing cached fs_locations data, a server could mistake such a request as indicating recognition of an NFS4ERR_LEASE_MOVED condition. Therefore, a compound that is not intended to signal that a client has recognized a migrated lease SHOULD be prefixed with a guard operation that fails with NFS4ERR_MOVED if the filehandle being queried is no longer present on the server. The guard can be as simple as a GETFH operation.

Though unlikely, it is possible that the target of such a compound could be migrated in the time after the guard operation is executed on the server but before the GETATTR(fs_locations) operation is encountered. When a client issues a GETATTR(fs_locations) operation as part of a compound not intended to signal recognition of a migrated lease, it SHOULD be prepared to process fs_locations data in the reply that shows the current location of the file system is gone.

6.1.4. Migration and the lease_time Attribute

In order that the client may appropriately manage its leases in the case of migration, the destination server must establish proper values for the lease_time attribute.

When state is transferred transparently, that state should include the correct value of the lease_time attribute. The lease_time attribute on the destination server must never be less than that on the source since this would result in premature expiration of leases granted by the source server. Upon migration in which state is transferred transparently, the client is under no obligation to refetch the lease_time attribute and may continue to use the value previously fetched (on the source server).

In the case in which lease merger occurs as part of state transfer, the lease_time attribute of the destination lease remains in effect. The client can simply renew that lease with its existing lease_time attribute. State in the source lease is renewed at the time of transfer so that it cannot expire, as long as the destination lease is appropriately renewed.

If state has not been transferred transparently (i.e., the client needs to reclaim or re-obtain its locks), the client should fetch the value of lease_time on the new (i.e., destination) server, and use it for subsequent locking requests. However, the server must respect a grace period at least as long as the lease_time on the source server, in order to ensure that clients have ample time to reclaim their locks before potentially conflicting non-reclaimed locks are granted. The means by which the new server obtains the value of lease_time on the old server is left to the server implementations. It is not specified by the NFS version 4.0 protocol.

7. Server Implementation Considerations

This section provides suggestions to help server implementers deal with issues involved in the transparent transfer of file-system-related data between servers. Servers are not obliged to follow these suggestions but should be sure that their approach to the issues handle all the potential problems addressed below.

7.1. Relation of Locking State Transfer to Other Aspects of File System Motion

In many cases, state transfer will be part of a larger function wherein the contents of a file system are transferred from server to server. Although specifics will vary with the implementation, the relation between the transfer of persistent file data and metadata

and the transfer of state will typically be described by one of the cases below.

- In some implementations, access to the on-disk contents of a file system can be transferred from server to server by making the storage devices on which the file system resides physically accessible from multiple servers, and transferring the right and responsibility for handling that file system from server to server.

  In such implementations, the transfer of locking state happens on its own, as described in Section 7.2. The transfer of physical access to the file system happens after the locking state is transferred and before any subsequent access to the file system. In cases where such transfer is not instantaneous, there will be a period in which all operations on the file system are held off, either by having the operations themselves return NFS4ERR_DELAY or, where this is not allowed, by using the techniques described below in Section 7.2.

- In other implementations, file system data and metadata must be copied from the server where they have existed to the destination server. Because of the typical amounts of data involved, it is generally not practical to hold off access to the file system while this transfer is going on. Normal access to the file system, including modifying operations, will generally happen while the transfer is going on.

  Eventually, the file system copying process will complete. At this point, there will be two valid copies of the file system, one on each of the source and destination servers. Servers may maintain that state of affairs by making sure that each modification to file system data is done on both the source and destination servers.

  Although the transfer of locking state can begin before the above state of affairs is reached, servers will often wait until it is arrived at to begin transfer of locking state. Once the transfer of locking state is completed, as described in the section below, clients may be notified of the migration event and access the destination file system on the destination server.

- Another case in which file system data and metadata must be copied from server to server involves a variant of the pattern above. In cases in which a single file system moves between or among a small set of servers, it will transition to a server on which a previous instantiation of that same file system existed before. In such cases, it is often more efficient to update the previous file

system instance to reflect changes made while the active file system was residing elsewhere rather than copying the file system data anew.

In such cases, the copying of file system data and metadata is replaced by a process that validates each visible file system object, copying new objects and updating those that have changed since the file system was last present on the destination server. Although this process is generally shorter than a complete copy, it is generally long enough that it is not practical to hold off access to the file system while this update is going on.

Eventually, the file system updating process will complete. At this point, there will be two valid copies of the file system, one on each of the source and destination servers. Servers may maintain that state of affairs just as is done in the previous case. Similarly, the transfer of locking state, once it is complete, allows the clients to be notified of the migration event and access the destination file system on the destination server.

7.2. Preventing Locking State Modification during Transfer

When transferring locking state from the source to a destination server, there will be occasions when the source server will need to prevent operations that modify the state being transferred. For example, if the locking state at time T is sent to the destination server, any state change that occurs on the source server after that time but before the file system transfer is made effective will mean that the state on the destination server will differ from that on the source server, which matches what the client would expect to see.

In general, a server can prevent some set of server-maintained data from changing by returning NFS4ERR_DELAY on operations that attempt to change that data. In the case of locking state for NFSv4.0, there are two specific issues that might interfere:

- Returning NFS4ERR_DELAY will not prevent state from changing in that owner-based sequence values will still change, even though NFS4ERR_DELAY is returned. For example, OPEN and LOCK will change state (in the form of owner seqid values) even when they return NFS4ERR_DELAY.

- Some operations that modify locking state are not allowed to return NFS4ERR_DELAY (i.e., OPEN_CONFIRM, RELEASE_LOCKOWNER, and RENEW).

Note that the first problem and most instances of the second can be addressed by returning NFS4ERR_DELAY on the operations that establish a filehandle within the target as one of the filehandles associated with the request, i.e., as either the current or saved filehandle. This would require returning NFS4ERR_DELAY under the following circumstances:

- On a PUTFH that specifies a filehandle within the target file system.
- On a LOOKUP or LOOKUPP that crosses into the target file system.

As a result of doing this, OPEN_CONFIRM is dealt with, leaving only RELEASE_LOCKOWNER and RENEW still to be dealt with.

Note that if the server establishes and maintains a situation in which no request has, as either the current or saved filehandle, a filehandle within the target file system, no special handling of SAVEFH or RESTOREFH is required. Thus, the fact that these operations cannot return NFS4ERR_DELAY is not a problem since neither will establish a filehandle in the target file system as the current filehandle.

If the server is to establish the situation described above, it may have to take special note of long-running requests that started before state migration. Part of any solution to this issue will involve distinguishing two separate points in time at which handling for the target file system will change. Let us distinguish:

- A time T after which the previously mentioned operations will return NFS4ERR_DELAY.
- A later time T' at which the server can consider file system locking state fixed, making it possible for it to be sent to the destination server.

For a server to decide on T', it must ensure that requests started before T cannot change target file system locking state, given that all those started after T are dealt with by returning NFS4ERR_DELAY upon setting filehandles within the target file system. Among the ways of doing this are:

- Keeping track of the earliest request started that is still in execution (for example, by keeping a list of active requests ordered by request start time). Requests that started before and are still in progress at time T may potentially affect the locking state; once the starting time of the earliest-started active request is later than T, the starting time of the first such

request can be chosen as T' by the server since any request in progress after T' started after time T. Accordingly, it would not have been allowed to change locking state for the migrating file system and would have returned NFS4ERR_DELAY had it tried to make a change.

- Keeping track of the count of requests started before time T that have a filehandle within the target file system as either the current or saved filehandle. The server can then define T' to be the first time after T at which the count is zero.

The set of operations that change locking state include two that cannot be dealt with by the above approach, because they are not specific to a particular file system and do not use a current filehandle as an implicit parameter.

- RENEW can be dealt with by applying the renewal to state for non-transitioning file systems. The effect of renewal for the transitioning file system can be ignored, as long as the servers make sure that the lease on the destination server has an expiration time that is no earlier than the latest renewal done on the source server. This can be easily accomplished by making the lease expiration on the destination server equal to the time in which the state transfer was completed plus the lease period.

- RELEASE_LOCKOWNER can be handled by propagating the fact of the lock-owner deletion (e.g., by using an RPC) to the destination server. Such a propagation RPC can be done as part of the operation, or the existence of the deletion can be recorded locally and propagation of owner deletions to the destination server done as a batch later. In either case, the actual deletions on the destination server have to be delayed until all of the other state information has been transferred.

  Alternatively, RELEASE_LOCKOWNER can be dealt with by returning NFS4ERR_DELAY. In order to avoid compatibility issues for clients not prepared to accept NFS4ERR_DELAY in response to RELEASE_LOCKOWNER, care must be exercised. (See Section 8.3 for details.)

The approach outlined above, wherein NFS4ERR_DELAY is returned based primarily on the use of current and saved filehandles in the file system, prevents all reference to the transitioning file system rather than limiting the delayed operations to those that change locking state on the transitioning file system. Because of this, servers may choose to limit the time during which this broad approach is used by adopting a layered approach to the issue.

- During the preparatory phase, operations that change, create, or destroy locks or modify the valid set of stateids will return NFS4ERR_DELAY. During this phase, owner-associated seqids may change, and the identity of the file system associated with the last request for a given owner may change as well. Also, RELEASE_LOCKOWNER operations may be processed without returning NFS4ERR_DELAY as long as the fact of the lock-owner deletion is recorded locally for later transmission.

- During the restrictive phase, operations that change locking state for the file system in transition are prevented by returning NFS4ERR_DELAY on any attempt to make a filehandle within that file system either the current or saved filehandle for a request. RELEASE_LOCKOWNER operations may return NFS4ERR_DELAY, but if they are processed, the lock-owner deletion needs to be communicated immediately to the destination server.

A possible sequence would be the following.

- The server enters the preparatory phase for the transitioning file system.

- At this point, locking state, including stateids, locks, and owner strings, is transferred to the destination server. The seqids associated with owners are either not transferred or transferred on a provisional basis, subject to later change.

- After the above has been transferred, the server may enter the restrictive phase for the file system.

- At this point, the updated seqid values may be sent to the destination server.

  Reporting regarding pending owner deletions (as a result of RELEASE_LOCKOWNER operations) can be communicated at the same time.

- Once it is known that all of this information has been transferred to the destination server, and there are no pending RELEASE_LOCKOWNER notifications outstanding, the source server may treat the file system transition as having occurred and return NFS4ERR_MOVED when an attempt is made to access it.

8. Additional Changes

This section contains a number of items that relate to the changes in the section above, but which, for one reason or another, exist in different portions of the specification to be updated.

8.1. Summary of Additional Changes from Previous Documents

Summarized here are all the remaining changes, not included in the two main sections.

- New definition of the error NFS4ERR_CLID_INUSE, appearing in Section 8.2. This replaces the definition in Section 13.1.10.1 in [RFC7530].
- A revision of the error definitions section to allow RELEASE_LOCKOWNER to return NFS4ERR_DELAY, with appropriate constraints to assure interoperability with clients not expecting this error to be returned. These changes are discussed in Section 8.2 and modify the error tables in Sections 13.2 and 13.4 in [RFC7530].
- A revised description of SETCLIENTID, appearing in Section 8.4. This brings the description into sync with the rest of the specification regarding NFS4ERR_CLID_INUSE. The revised description replaces the one in Section 16.33 of [RFC7530].
- Some security-related changes appear in Sections 8.5 and 8.6. The Security Considerations section of this document (Section 9) describes the effect on the corresponding section (Section 19) in [RFC7530].

8.2. NFS4ERR_CLID_INUSE Definition

The definition of this error is now as follows:

> The SETCLIENTID operation has found that the id string within the specified nfs_client_id4 was previously presented with a different principal and that client instance currently holds an active lease. A server MAY return this error if the same principal is used, but a change in authentication flavor gives good reason to reject the new SETCLIENTID operation as not bona fide.

8.3. NFS4ERR_DELAY Return from RELEASE_LOCKOWNER

The existing error tables should be considered modified to allow NFS4ERR_DELAY to be returned by RELEASE_LOCKOWNER. However, the scope of this addition is limited and is not to be considered as making this error return generally acceptable.

It needs to be made clear that servers may not return this error to clients not prepared to support file system migration. Such clients may be following the error specifications in [RFC7530] and so might not expect NFS4ERR_DELAY to be returned on RELEASE_LOCKOWNER.

The following constraint applies to this additional error return, as if it were a note appearing together with the newly allowed error code:

> In order to make server state fixed for a file system being migrated, a server MAY return NFS4ERR_DELAY in response to a RELEASE_LOCKOWNER that will affect locking state being propagated to a destination server. The source server MUST NOT do so unless it is likely that it will later return NFS4ERR_MOVED for the file system in question.
>
> In the context of lock-owner release, the set of file systems, such that server state being made fixed can result in NFS4ERR_DELAY, must include the file system on which the operation associated with the current lock-owner seqid was performed.
>
> In addition, this set may include other file systems on which an operation associated with an earlier seqid for the current lock-owner seqid was performed, since servers will have to deal with the issue of an owner being used in succession for multiple file systems.
>
> Thus, if a client is prepared to receive NFS4ERR_MOVED after creating state associated with a given file system, it also needs to be prepared to receive NFS4ERR_DELAY in response to RELEASE_LOCKOWNER, if it has used that owner in connection with a file on that file system.

8.4. Operation 35: SETCLIENTID -- Negotiate Client ID

8.4.1. SYNOPSIS

```
client, callback, callback_ident -> clientid, setclientid_confirm
```

8.4.2. ARGUMENT

```
struct SETCLIENTID4args {
        nfs_client_id4  client;
        cb_client4      callback;
        uint32_t        callback_ident;
};
```

8.4.3. RESULT

```
struct SETCLIENTID4resok {
        clientid4       clientid;
        verifier4       setclientid_confirm;
};

union SETCLIENTID4res switch (nfsstat4 status) {
 case NFS4_OK:
         SETCLIENTID4resok      resok4;
 case NFS4ERR_CLID_INUSE:
         clientaddr4    client_using;
default:
         void;
};
```

8.4.4. DESCRIPTION

The client uses the SETCLIENTID operation to notify the server of its intention to use a particular client identifier, callback, and callback_ident for subsequent requests that entail creating lock, share reservation, and delegation state on the server. Upon successful completion, the server will return a shorthand client ID that, if confirmed via a separate step, will be used in subsequent file locking and file open requests. Confirmation of the client ID must be done via the SETCLIENTID_CONFIRM operation to return the client ID and setclientid_confirm values, as verifiers, to the server. The reason why two verifiers are necessary is that it is possible to use SETCLIENTID and SETCLIENTID_CONFIRM to modify the callback and callback_ident information but not the shorthand client ID. In that event, the setclientid_confirm value is effectively the only verifier.

The callback information provided in this operation will be used if the client is provided an open delegation at a future point. Therefore, the client must correctly reflect the program and port numbers for the callback program at the time SETCLIENTID is used.

The callback_ident value is used by the server on the callback. The client can leverage the callback_ident to eliminate the need for more than one callback RPC program number, while still being able to determine which server is initiating the callback.

8.4.5. IMPLEMENTATION

To specify the implementation of SETCLIENTID, the following notations are used.

Let:

x be the value of the client.id subfield of the SETCLIENTID4args structure.

v be the value of the client.verifier subfield of the SETCLIENTID4args structure.

c be the value of the client ID field returned in the SETCLIENTID4resok structure.

k represent the value combination of the callback and callback_ident fields of the SETCLIENTID4args structure.

s be the setclientid_confirm value returned in the SETCLIENTID4resok structure.

{ v, x, c, k, s } be a quintuple for a client record. A client record is confirmed if there has been a SETCLIENTID_CONFIRM operation to confirm it. Otherwise, it is unconfirmed. An unconfirmed record is established by a SETCLIENTID call.

8.4.5.1. IMPLEMENTATION (Preparatory Phase)

Since SETCLIENTID is a non-idempotent operation, our treatment assumes use of a duplicate request cache (DRC). For a discussion of the DRC, see Section 9.1.7 of [RFC7530].

When the server gets a SETCLIENTID { v, x, k } request, it first does a number of preliminary checks as listed below before proceeding to the main part of SETCLIENTID processing.

- It first looks up the request in the DRC. If there is a hit, it returns the result cached in the DRC. The server does NOT remove client state (locks, shares, delegations) nor does it modify any recorded callback and callback_ident information for client { x }. The server now proceeds to the main part of SETCLIENTID.

- Otherwise (i.e., in the case of any DRC miss), the server takes the client ID string x and searches for confirmed client records for x that the server may have recorded from previous SETCLIENTID calls. If there are no such records, or if all such records have

a recorded principal that matches that of the current request's principal, then the preparatory phase proceeds as follows.

* If there is a confirmed client record with a matching client ID string and a non-matching principal, the server checks the current state of the associated lease. If there is no associated state for the lease, or the lease has expired, the server proceeds to the main part of SETCLIENTID.

* Otherwise, the server is being asked to do a SETCLIENTID for a client by a non-matching principal while there is active state. In this case, the server rejects the SETCLIENTID request returning an NFS4ERR_CLID_INUSE error, since use of a single client with multiple principals is not allowed. Note that even though the previously used clientaddr4 is returned with this error, the use of the same id string with multiple clientaddr4s is not prohibited, while its use with multiple principals is prohibited.

8.4.5.2. IMPLEMENTATION (Main Phase)

If the SETCLIENTID has not been dealt with by DRC processing, and has not been rejected with an NFS4ERR_CLID_INUSE error, then the main part of SETCLIENTID processing proceeds, as described below.

o The server checks if it has recorded a confirmed record for { v, x, c, l, s }, where l may or may not equal k. If so, and since the id verifier v of the request matches that which is confirmed and recorded, the server treats this as a probable callback information update and records an unconfirmed { v, x, c, k, t } and leaves the confirmed { v, x, c, l, s } in place, such that t != s. It does not matter if k equals l or not. Any pre-existing unconfirmed { v, x, c, *, * } is removed.

  The server returns { c, t }. It is indeed returning the old clientid4 value c, because the client apparently only wants to update callback value k to value l. It's possible this request is one from the Byzantine router that has stale callback information, but this is not a problem. The callback information update is only confirmed if followed up by a SETCLIENTID_CONFIRM { c, t }.

  The server awaits confirmation of k via SETCLIENTID_CONFIRM { c, t }.

  The server does NOT remove client (lock/share/delegation) state for x.

- The server has previously recorded a confirmed { u, x, c, l, s } record such that v != u, l may or may not equal k, and has not recorded any unconfirmed { *, x, *, *, * } record for x. The server records an unconfirmed { v, x, d, k, t } (d != c, t != s).

  The server returns { d, t }.

  The server awaits confirmation of { d, k } via SETCLIENTID_CONFIRM { d, t }.

  The server does NOT remove client (lock/share/delegation) state for x.

- The server has previously recorded a confirmed { u, x, c, l, s } record such that v != u, l may or may not equal k, and recorded an unconfirmed { w, x, d, m, t } record such that c != d, t != s, m may or may not equal k, m may or may not equal l, and k may or may not equal l. Whether w == v or w != v makes no difference. The server simply removes the unconfirmed { w, x, d, m, t } record and replaces it with an unconfirmed { v, x, e, k, r } record, such that e != d, e != c, r != t, r != s.

  The server returns { e, r }.

  The server awaits confirmation of { e, k } via SETCLIENTID_CONFIRM { e, r }.

  The server does NOT remove client (lock/share/delegation) state for x.

- The server has no confirmed { *, x, *, *, * } for x. It may or may not have recorded an unconfirmed { u, x, c, l, s }, where l may or may not equal k, and u may or may not equal v. Any unconfirmed record { u, x, c, l, * }, regardless whether u == v or l == k, is replaced with an unconfirmed record { v, x, d, k, t } where d != c, t != s.

  The server returns { d, t }.

  The server awaits confirmation of { d, k } via SETCLIENTID_CONFIRM { d, t }. The server does NOT remove client (lock/share/delegation) state for x.

The server generates the clientid and setclientid_confirm values and must take care to ensure that these values are extremely unlikely to ever be regenerated.

8.5. Security Considerations for Inter-server Information Transfer

Although the means by which the source and destination server communicate is not specified by NFSv4.0, the following security-related considerations for inter-server communication should be noted.

- Communication between source and destination servers needs to be carried out in a secure manner, with protection against deliberate modification of data in transit provided by using either a private network or a security mechanism that ensures integrity. In many cases, privacy will also be required, requiring a strengthened security mechanism if a private network is not used.

- Effective implementation of the file system migration function requires that a trust relationship exist between source and destination servers. The details of that trust relationship depend on the specifics of the inter-server transfer protocol, which is outside the scope of this specification.

- The source server may communicate to the destination server security-related information in order to allow it to more rigorously validate clients' identity. For example, the destination server might reject a SETCLIENTID done with a different principal or with a different IP address than was done previously by the client on the source server. However, the destination server MUST NOT use this information to allow any operation to be performed by the client that would not be allowed otherwise.

8.6. Security Considerations Revision

The penultimate paragraph of Section 19 of [RFC7530] should be revised to read as follows:

> Because the operations SETCLIENTID/SETCLIENTID_CONFIRM are responsible for the release of client state, it is imperative that the principal used for these operations be checked against and match the previous use of these operations. In addition, use of integrity protection is desirable on the SETCLIENTID operation, to prevent an attack whereby a change in the boot instance id (verifier) forces an undesired loss of client state. See Section 5 for further discussion.

9. Security Considerations

The security considerations of [RFC7530] remain appropriate with the exception of the modification to the penultimate paragraph specified in Section 8.6 of this document and the addition of the material in Section 8.5.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <http://www.rfc-editor.org/info/rfc7530>.

10.2. Informative References

[INFO-MIGR] Noveck, D., Ed., Shivam, P., Lever, C., and B. Baker, "NFSv4 migration: Implementation experience and spec issues to resolve", Work in Progress, draft-ietf-nfsv4-migration-issues-09, February 2016.

[RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <http://www.rfc-editor.org/info/rfc1813>.

[RFC5661] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 5661, DOI 10.17487/RFC5661, January 2010, <http://www.rfc-editor.org/info/rfc5661>.

Acknowledgements

The editor and authors of this document gratefully acknowledge the contributions of Trond Myklebust of Primary Data and Robert Thurlow of Oracle. We also thank Tom Haynes of Primary Data and Spencer Shepler of Microsoft for their guidance and suggestions.

Special thanks go to members of the Oracle Solaris NFS team, especially Rick Mesta and James Wahlig, for their work implementing an NFSv4.0 migration prototype and identifying many of the issues addressed here.

Authors' Addresses

David Noveck (editor)
Hewlett Packard Enterprise
165 Dascomb Road
Andover, MA 01810
United States of America

Phone: +1 978 474 2011
Email: davenoveck@gmail.com

Piyush Shivam
Oracle Corporation
5300 Riata Park Ct.
Austin, TX 78727
United States of America

Phone: +1 512 401 1019
Email: piyush.shivam@oracle.com

Charles Lever
Oracle Corporation
1015 Granger Avenue
Ann Arbor, MI 48104
United States of America

Phone: +1 734 274 2396
Email: chuck.lever@oracle.com

Bill Baker
Oracle Corporation
5300 Riata Park Ct.
Austin, TX 78727
United States of America

Phone: +1 512 401 1081
Email: bill.baker@oracle.com