NFSv4
Internet-Draft
Updates: 3530 (if approved)
Intended status: Standards Track
Expires: April 15, 2014

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

October 12, 2013

NFSv4.0 migration: Specification Update
draft-ietf-nfsv4-rfc3530-migration-update-03

Abstract

The migration feature of NFSv4 allows for responsibility for a single filesystem to move 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 clarifies and corrects the NFSv4.0 specification (RFC3530 and possible successors) to address these problems.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 15, 2014.

Table of Contents

1. Introduction

This document is a standards track document which corrects the existing definitive specification of the NFSv4.0 protocol, in [RFC3530] and the one expected to become definitive (now in [cur-rfc3530-bis]). 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:

- Filesystem migration, particularly when it involves transparent state migration.

- The construction and interpretation of the nfs_clientid4 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. Background

Implementation experience with transparent state migration has exposed a number of problems with the existing specification of this feature, in [RFC3530] and in RFC3530bis (see the draft at [cur-rfc3530-bis]). The symptoms were:

- After migration of a filesystem, 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 (id and verifier), 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 clientid4's, 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.

Specification of requirements 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, although not "RECOMMENDED", by existing NFSv4.0 specifications ([RFC3530] and RFC3530bis, whose current draft is at [cur-rfc3530-bis]).

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

This document updates the existing NFSv4.0 specifications ([RFC3530] and RFC3530bis, whose current draft is at [cur-rfc3530-bis]) as follows:

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

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

- It gives some guidance regarding the factors that might govern clients' choice of a client id construction approach and RECOMMENDS that clients construct client id strings in 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.

For a more complete explanation of the choices made in addressing these issues, see [info-migr]).

4. Client Identity Definition

This chapter is a replacement for sections 8.1.1 and 8.1.2 in [RFC3530] and for sections 9.1.1 and 9.1.2 in RFC3530bis (see the draft at [cur-rfc3530-bis]). The replaced sections are named "client ID" and "Server Release of Clientid."

It supersedes the replaced sections.

4.1. Differences from Replaced Sections

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

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

- It clearly addresses the need to save 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 server address trunking patterns.

4.2. 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 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).

The basis of the client identification infrastructure is encapsulated in the following data structure:

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

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

- The id field is a variable-length string which uniquely identifies a specific client. Although, we describe it as a string and it 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 string MAY be different for each server network address that the client accesses, rather than common to all server network addresses.

- The verifier is a client incarnation identifier that is used by the server to detect client reboots. Only if the verifier 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, once it receives confirmation of the new nfs_clientd4 via SETCLIENTID_CONFIRM.

  As a security measure, 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 4.4.

- The algorithm for generating the string should not assume that the client's network address is forever fixed. Changes might occur between client incarnations and even while the client is still running in its current incarnation.

  Having the client id string change simply because of a network address change would mean that successive SETCLIENTID operations for the same client would appear as from different clients, interfering with the use of the nfs_client_id4 verifier 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 a similar algorithm for generating the id string, will generate a conflicting id string.

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 same server. This applies across server restarts or reboots.

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 clientid4's independently, and 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 4.4.

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, the set of distinct owner values used with that client constitutes the set of 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 the 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 which has a matching id string and a non-matching boot 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 clientid 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 the section entitled "Server Failure and Recovery"). In cases of server or client error resulting in this error, use of SETCLIENTID to establish a new lease is desirable as well.

In the last two cases, different recovery procedures are required. See Section 5.3 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 a NFS4ERR_STALE_STATEID error using a stateid derived from its current clientid4, since this indicates a situation, such as server reboot which has invalidated the existing clientid4 and associated stateids (see the section entitled "lock-owner" for details).

See the detailed descriptions of SETCLIENTID and SETCLIENTID_CONFIRM for a complete specification of these operations.

4.3. 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 that event, 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.

4.4. 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 4.6.

- 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 4.5.

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 4.7.

Construction of the client id string has arisen as a difficult issue because of the way in which the NFS protocols have evolved.

- 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 occasion 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 filesystem in the client's file name space, 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 (they return the same server name in responding to an EXCHANGE_ID).

NFSv4.0 is unfortunately halfway between these two. 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 [RFC3530], became more substantial as implementation proceeded.

- In the absence of any implementation of the fs_locations-related features (replication, referral, and migration), the situation is very similar to that of NFSv3, with the addition of state but with no concern to provide 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 on the same IP address, 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 are clients. 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.

4.5. 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 clientid/lease per server network addresses.

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 SHOULD NOT

use the non-uniform client id string approach, except where it is necessary for compatibility with existing server implementations (For details of arranging use of multiple client id string approaches, see Section 4.7).

4.6. 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 information as to client identity, 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 nfs_client_id4 boot verifier:

- In [RFC3530], the client is told to change the boot verifier when reboot occurs, but there is no explicit statement as to the converse, so that any requirement to keep the verifier constant unless rebooting is only present by implication.

- Many existing clients change the boot verifier 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 verifier 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 boot verifier should change if and only if a new client instance is created, typically as a result of a reboot.

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, we never have a delegation under one clientid revoked because of a reference to the same file from the same client under a different clientid.

- 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.

How to balance these considerations depends on implementation goals.

4.7. Mixing client id string Approaches

As noted above, a client which 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 [RFC3530], to keep these sets of state separate, and will have problems in 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 clientid for the same client, established using a different IP address. This causes difficulty for a multi-homed client using the uniform client id string approach.

  Although this behavior is not correct, such servers still exist and the spec 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, as long as:

- The uniform client id string approach is used when accessing servers that may return NFS4ERR_MOVED and 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

One effective way for clients to handle this is to support the uniform client id string approach as the default, but allow a mount option to specify use of the non-uniform client id string approach for particular mount points, as long as such mount points are not used when migration is to be supported.

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 clientid's corresponding to the same server, one for each approach and would then have to keep these separate.

4.8. 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.

We need to clarify 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 blind-sided and encounter disconcerting server behavior, as mentioned in Section 4.6. 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 clientid's 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 clientid's 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 do that.

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 accessible, for each nfs_client_id4 used by the uniform approach (only one in general) 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 a 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. We'll refer 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 it will use the common nfs_client_id4 and specify the current target IP address as part of the callback parameters. We call the clientid4 and SETCLIENTID verifier returned by this operation XC and XV.

Note that when the client has done previous SETCLIENTID's, to any IP addresses, with more than one principal or authentication flavor, we have the possibility of receiving NFS4ERR_CLID_INUSE, since we do not yet know which of our connections with existing IP addresses might be trunked with our 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 our SETCLIENTID 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 that we don't want to confirm something that we are not sure we want to happen, what is to be done next depends on information about existing clientid4's.

- 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 which 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.

- o If one or more matching clientid4's is found, none of which is marked unresolved, the new IP address in entered and marked unresolved. After applying the steps below to each of the lead IP addresses with a matching clientid4, the address will have been resolved: It may 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

So for each lead IP address IPn with a clientid4 matching XC, the following steps are done.

- o 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 are correctly implemented.

- o 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 clientid, in the event that X and IPn are trunked together. So assume that we do that SETCLIENTID on IP address IPn and get back a setclientid_confirm value (in the form of a verifier4) 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 clientid XC is duplicated by distinct servers, the probability that Sc 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 spurious matches further.

- Note that we don't want this to happen if address X is not associated with this server. So we do a SETCLIENTID_CONFIRM on address X using the setclientid_confirm value SCn.

- If the setclientid_confirm value generated on X is accepted on IPn, then X and IPn are recognized as connected to the same server and the entry for X is marked as associated with IPn. The entry is now resolved and processing can be restarted for IP addresses whose clientid4 matched XC but whose resolution had been deferred.

- If the confirm value generated on IPn is not accepted on X, then X and IPn are distinct and the callback update will not be confirmed. So we go on to the next IPn, until we run out of them. If it happens that we run out of potential matches, then we can treat X as connected to a distinct server and then update and confirm its callback parameters on that basis.

Note here that we may set a number of possible values for the callback parameters to be used for XC, one for the possibility that X is untrunked, and others for each potential match with an existing IPn. Although there are multiple such updates at most one will be

confirmed and, if X is untrunked, its original callback parameters will be put in effect by its SETCLIENTID_CONFIRM.

The procedure described above must be performed so as to exclude the possibility that multiple SETCLIENTID's, 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 periodically use the clientid4 XC in RENEW operations, directed to both the IP address X and the current lead IP address that is currently being tested for identity.

- When XC becomes invalid on X, the resolution process should be terminated, subject to being redone later. Before redoing the resolution, XC should be checked on all the lead IP addresses on which it was valid. Once a new clientid4 is established on any servers on which XC became invalid, a new clientid4 can be established on X and the resolution process for X can be restarted.

- When XC does not becomes invalid on X, but becomes invalid on the current IPn being tested, it should be concluded that X and IPn do not match and that it is time to advance to the next IPn, if any.

- 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.

If we have run out of IPn's without finding a matching server, X is 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.

4.9. Client id string construction details

This section gives more detailed guidance on client id construction. 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.
- 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 4.2, an example of a well generated id string is one that includes:

- The client's network address, or more safely, an address that has previously been used in that capacity.
- For a user level NFSv4.0 client, it should contain additional information to distinguish the client from other user level clients running on the same host, such as a universally unique identifier (UUID).
- Additional information that tends to be unique, such as one or more of:
  - The client machine's serial number (for privacy reasons, it is best to perform some one way function on the serial number).

  - A MAC address. Note that this can cause difficulties when there are configuration changes or when a client has multiple network adapters.
  - 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.

5. Locking and Multi-Server Namespace

This chapter is a replacement for section 7.7.6, "Lock State and File System transitions", in RFC3530bis (see the draft at [cur-rfc3530-bis]).

With respect to [RFC3530], it serves as a replacement for section 8.14, "Migration, Replication, and State".

It supersedes the replaced sections.

5.1. Changes from Replaced Sections

These changes 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 filesystem 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 openowners and lockowners is to be managed with regard to migration.
- The description of handling of the NFS4ERR_LEASE_MOVED has been rewritten for greater clarity.

5.2. Lock State and Filesystem Transitions

When responsibility for handling a given filesystem is transferred to a new server (migration) or the client chooses to use an alternate server (e.g., in response to server unresponsiveness) in the context of filesystem replication, 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 filesystem 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. When doing so, servers acting as replicas or immigrating filesystems 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 filesystem.

5.3. Migration and State

In the case of migration, the servers involved in the migration of a filesystem SHOULD transfer all server state associated with the migrating filesystem from source to the destination server. This must be done in a way that is transparent to the client. This state transfer will ease the client's transition when a filesystem 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
- That they represent the same types of locks, whether opens, delegations, advisory byte-range locks, or mandatory byte-range locks.
- That they have the same lock particulars, including such things as access modes, deny modes, and byte ranges.
- That 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 co-ordinate 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.

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 boot verifier mismatch.

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

Since responsibility for an entire filesystem 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 SHOULD provide a filesystem-specific grace period, to allow clients to reclaim locks associated with files in the migrated filesystem. 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 filesystem 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 op or a READ op 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.

5.3.1. Migration and clientid's

Handling of clientid values is similar to that for stateids. However, there are some differences that derive from the fact that a clientid is an object which spans multiple filesystems while a stateid is inherently limited to a single filesystem.

The clientid4 and nfs_client_id4 information (id string and boot verifier) 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 boot 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 clientid4's. Since boot verifiers 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.

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. The client can determine the new clientid (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 clientid with respect to the server in question.

5.3.2. Migration and state owner information

In addition to stateids, the locks they represent, and clientid information, servers also need to transfer information related to the current status of openowners and lockowners.

This information includes:

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

When clients are implemented to isolate each openowner and lockowner to a particular filesystem, 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.

Note that when servers take this approach for all owners whose state is limited to the particular filesystem being migrated, doing so will not cause difficulties for clients not adhering to an approach in which owners are isolated to particular filesystems. 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 filesystem but other filesystems 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 filesystem being migrated, owner status SHOULD be propagated to the destination filesystem.

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

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

Note that a server may obey all of the conditions above without the overhead of keeping track of set of filesystems 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 filesystem fixed, preparatory to propagating it to the destination filesystem. If a client is free to create new locks associated with existing owners on other filesystems, the owner information may be propagated to the destination filesystem, even though, at the time the filesystem migration is recognized by the client to have occurred, the last operation associated with the owner may not be associated with the migrating filesystem.

When source server propagates owner-related state associated with owners that span multiple filesystems, 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 which 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 filesystem is not, when the reissued request is received, part of the set of filesystems handled by that server.

One sort of case may require more complex handling. When multiple filesystem 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 filesystem 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 filesystems.

As a way of understanding the situations which 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 clientid's designating C, which we refer to as CX and CY.

- Initially server X supports filesystems 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 filesystem F1 through F4 on behalf of owner O (either a lockowner or an openowner), with each request going to X or Y depending on where the relevant filesystem 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, find the

target filesystem has moved to Y, and need to re-issue 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 situations, the server needs to provide support for dealing with retransmission of owner-sequenced requests that diverges 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. Such support only needs to be provided for requests issued before the migration event whose status as the last by sequence is invalidated by the migration event.

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 filesystem 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 filesystem 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 we can have 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 filesystem 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 the next sequence to be received.

- When the owner sequence value for a migrating filesystem is less

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

- When the owner sequence value for a migrating filesystem is equal to the corresponding value on the destination server, one has two different "last" requests which 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 filesystem.

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

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

5.4. 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 filesystems, 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.

5.5. Notification of Migrated Lease

A filesystem can be migrated to another server while a client that has state related to that filesystem 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 filesystem 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 which 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 filesystem 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 filesystem 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 only required when responsibility for at least one stateid has been affected. In the case of a null lease, where the only associated state is a clientid, an NFS4ERR_LEASE_MOVED error SHOULD NOT be generated.

Upon receiving the NFS4ERR_LEASE_MOVED error, a client that supports filesystem migration MUST perform the necessary GETATTR operation for each of the filesystems containing state that have been migrated and so give the server evidence that it is aware of the migration of the filesystem. Once the client has done this for all migrated filesystems 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 filesystems is described below. This approach divides the process into two phases, one devoted to finding the migrated filesystems and the second devoted to doing the necessary GETATTRs.

The client can find the migrated filesystems by building and issuing one or more COMPOUND requests, each consisting of a set of PUTFH/ GETFH pairs, each pair using an fh in one of the filesystems in question. All such COMPOUND requests can be done in parallel. The successful completion of such a request indicates that none of the filesystems interrogated have been migrated while termination with NFS4ERR_MOVED indicates that the filesystem 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 filesystems which remain in an uncertain state.

Once the migrated filesystems 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 filesystems found by this process, by interrogating the fs_locations attribute for an fh within each of the migrated filesystems. 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 filesystems 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 NFS4ERR_MOVED. In this situation, it is likely that the client would find its lease expired although a server may use "courtesy" locks 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 filesystem 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 which is not intended to signal that a client has recognized a migrated lease SHOULD be prefixed with a guard operation which fails with NFS4ERR_MOVED if the file handle 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 filesystem is gone.

5.6. 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 re-fetch 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.

6. Server Implementation Considerations

This chapter provides suggestions to help server implementers deal with issues involved in the transparent transfer of filesystem-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.

6.1. Relation of Locking State Transfer to Other Aspects of Filesystem Motion

In many cases, state transfer will be part of a larger function wherein the contents of a filesystem 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 filesystem can be transferred from server to server by making the storage devices on which the filesystem resides physically accessible from multiple servers, and transferring the right and responsibility for handling that filesystem from server to server.

  In such implementations, the transfer of locking state happens on its own, as described in Section 6.2. The transfer of physical access to the filesystem happens after the locking state is transferred and before any subsequent access to the filesystem. In cases where such transfer is not instantaneous, there will be a period in which all operations on the filesystem 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 6.2.

- In other implementations, filesystem data and metadata must be copied from the server where it has existed to the destination server. Because of the typical amounts of data involved, it is

generally not practical to hold off access to the filesystem while this transfer is going on. Normal access to the filesystem, including modifying operations, will generally happen while the transfer is going on.

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

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

filesystem was residing elsewhere, rather than copying the filesystem data anew.

In such cases, the copying of filesystem data and metadata is replaced by a process which validates each visible filesystem object, copying new objects and updating those that have changed since the filesystem 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 filesystem while this update is going on.

Eventually the filesystem updating process will complete. At this point, there will be two valid copies of the filesystem, 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 filesystem on the destination server.

6.2. 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 filesystem 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 which 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 which modify locking state are not allowed to return NFS4ERR_DELAY.

Note that the first problem and many 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 filesystem.

- On a LOOKUP or LOOKUPP that crosses into the target filesystem.

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 filesystem, 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 filesystem 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 which 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 filesystem 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 filesystem 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 filesystem locking state, given that all those started after T are dealt with by returning NFS4ERR_DELAY upon setting filehandles within the target filesystem. Among the ways of doing this are:

- Keeping track of the earliest request started which is still in execution (for example, by keeping a list of active requests ordered by request start time). The server can then define T' to be the first time at which the earliest-started active request started after time T.

- Keeping track of the count of requests, started before time T which have a filehandle within the target filesystem 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

filesystem-specific 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 filesystems. The effect of renewal for the transitioning filesystem 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 the state transfer was completed plus the lease period.

- RELEASE_LOCKOWNER can be handled by propagating the fact of the lockowner 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 7.3 for details.)

The approach outlined above, wherein NFS$ERR_DELAY is returned based primarily on the use of current and saved filehandles in the filesystem, prevents all reference to the transitioning filesystem, rather than limiting the delayed operations to those that change locking state on the transitioning filesystem. 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 filesystem 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 lockowner deletion is recorded locally for later transmission.

- During the restrictive phase, operations that change locking state for the filesystem in transition are prevented by returning NFS4ERR_DELAY on any attempt to make a filehandle within that filesystem either the current or saved filehandle for a request.

RELEASE_LOCKOWNER operations may return NFS4ERR_DELAY, but if they are processed, the lockowner 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 filesystem.
- At this point locking state, including stateids, locks, owner strings are 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 filesystem.
- 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 filesystem transition as having occurred and return NFS4ERR_MOVED when an attempt is made to access it.

7. Additional Changes

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

7.1. Summary of Additional Changes from Previous Documents

We summarize here all the remaining changes, not included in the two main chapters.

- New definition of the CLID_INUSE error.
- A revised description of SETCLIENTID, which brings the description into sync with the rest of the spec regarding CLID_INUSE.

- A revision to the Security Considerations section, indicating why integrity protection is needed for the SETCLIENTID operation.

- A revision of the error definitions chapter to allow RELEASE_LOCKOWNER to return NFS4ERR_DELAY, with appropriate constraints to assure interoperability with clients not expecting this error to be returned.

7.2. 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.

7.3. 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 filesystem migration. Such clients may be following the error specifications in [RFC3530] and [cur-rfc3530-bis] 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 filesystem 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 filesystem in question.

In the context of lockowner release, the set of filesystems such that server state being made fixed can result in NFS4ERR_DELAY must include the filesystem on which the operation associated with the current lockowner seqid was performed.

In addition, this set may include other filesystems on which an operation associated with an earlier seqid for the current lockowner seqid was performed, since servers will have to deal

with the issue of an owner being used in succession for multiple filesystems.

Thus, a client that is prepared to receive NFS4ERR_MOVED after creating state associated with a given filesystem, 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 filesystem.

7.4. Operation 35: SETCLIENTID - Negotiate Client ID

7.4.1. SYNOPSIS

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

7.4.2. ARGUMENT

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

7.4.3. 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;
};
```

7.4.4. 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 which, 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.

7.4.5. IMPLEMENTATION

To understand how to implement SETCLIENTID, make the following notations. 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 fields 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.

7.4.5.1. IMPLEMENTATION (preparatory phase)

Since SETCLIENTID is a non-idempotent operation, let us assume that the server is implementing the duplicate request cache (DRC).

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 }.

- 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, or if all such records have a recorded principal which matches that of the current request's principal, then

- 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 and 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 clientaddr is returned with this error, the use of the same id string with multiple clientaddr's is not prohibited, while its use with multiple principals is prohibited.

7.4.5.2. 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.

- 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.

7.5. Security Considerations revision

The last paragraph of the "Security Considerations" section 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 is 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 verifier forces an undesired loss of client state. See the section "Client Identity Definition" for further discussion.

8. Security Considerations

Is modified as specified in Section 7.5.

9. IANA Considerations

This document does not require actions by IANA.

10. Acknowledgements

The editor and authors of this document gratefully acknowledge the contributions of Trond Myklebust of NetApp and Robert Thurlow of Oracle. We also thank Tom Haynes of NetApp 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.

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, April 2003.

11.2. Informative References

[RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, June 1995.

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

[cur-rfc3530-bis] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", 2013, <http://www.ietf.org/id/draft-ietf-nfsv4-rfc3530bis-26.txt>.

Work in progress.

[info-migr] Noveck, D., Ed., Shivam, P., Lever, C., and B. Baker, "NFSv4 migration: Implementation experience and spec issues to resolve ", 2012, <http://www.ietf.org/id/draft-ietf-nfsv4-migration-issues-03.txt>.

Work in progress.

Authors' Addresses

David Noveck (editor)
EMC Corporation
228 South Street
Hopkinton, MA 01748
US

Phone: +1 508 249 5748

Email: david.noveck@emc.com

Piyush Shivam
Oracle Corporation
5300 Riata Park Ct.
Austin, TX 78727
US

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

Charles Lever
Oracle Corporation
1015 Granger Avenue
Ann Arbor, MI 48104
US

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

Bill Baker
Oracle Corporation
5300 Riata Park Ct.
Austin, TX 78727
US

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