

NFSv4 Working Group
Internet-Draft
Expires: December 21, 2006

D. Noveck
Network Appliance
June 19, 2006

Chapters for Migration, Replication, and Referrals for v4.1 Draft
draft-dnoveck-location-chapters-00

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with [Section 6 of BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on December 21, 2006.

Copyright Notice

Copyright (C) The Internet Society (2006).

Abstract

This includes material proposed for inclusion into the next v4.1 draft. It includes one revised chapter, one rewritten chapter, and a set of suggestions for changes in other part of the v4.1 document.

Internet-Draft

Location Chapters for 4.1

June 2006

Table of Contents

1.	Introduction	3
2.	Single-server Name Space	5
2.1.	Server Exports	5
2.2.	Browsing Exports	5
2.3.	Server Pseudo Filesystem	6
2.4.	Multiple Roots	6
2.5.	Filehandle Volatility	6
2.6.	Exported Root	7
2.7.	Mount Point Crossing	7
2.8.	Security Policy and Name Space Presentation	7
3.	Multi-server Name Space	9
3.1.	Location attributes	9
3.2.	File System Presence or Absence	9
3.3.	Getting Attributes for an Absent File System	10
3.3.1.	GETATTR Within an Absent File System	11
3.3.2.	REaddir and Absent File Systems	12
3.4.	Uses of Location Information	12
3.4.1.	File System Replication	13
3.4.2.	File System Migration	14
3.4.3.	Referrals	14
3.5.	Additional Client-side Considerations	15
3.6.	Effecting File System Transitions	16
3.6.1.	Transparent File System Transitions	17
3.6.2.	Filehandles and File System Transitions	18
3.6.3.	Fileid's and File System Transitions	19
3.6.4.	Fsid's and File System Transitions	20
3.6.5.	The Change Attribute and File System Transitions	20
3.6.6.	Lock State and File System Transitions	20
3.6.7.	Write Verifiers and File System Transitions	24
3.7.	Effecting File System Referrals	24
3.7.1.	Referral Example (LOOKUP)	25
3.7.2.	Referral Example (REaddir)	28
3.8.	The Attribute fs_absent	31
3.9.	The Attribute fs_locations	31
3.10.	The Attribute fs_locations_info	33
3.11.	The Attribute fs_status	42
4.	Other Changes	46
5.	References	47
	Author's Address	48
	Intellectual Property and Copyright Statements	49

1. Introduction

This draft consists of mainly of proposed material for the v4.1 draft, to replace the handling of server name space, migration, and replication in the current v4.1 draft, which has been inherited pretty much unchanged from [RFC3530](#) [1].

It is based on the current working group document on ideas published in some previously published but now expired individual and working group drafts (Guide for Referrals in NFSv4, and Next Steps for NFSv4 Migration/Replication).

Other than stylistic changes that reflect the difference in purpose between those earlier drafts and the v4.1 spec-to-be, the following changes have been made:

- o Deleted type as an attribute to be returned in an absent fs (sometimes). Too much additional complexity for no real value.
- o Added RMDA-capability flag to fs_locations_info as requested by Trond.
- o Added field telling how-current fs is to fs_status as requested by Craig E.
- o In light of "we don't know if this is the right set of stuff" comments I'd heard, restructured the fs_locations_info stuff for greater changeability/expandability. Also hope to make this easier to understand without compromising other goals.
- o Deleted fh-replacement stuff due to lack of interest. Could add back if people really want it.
- o Deleted VLCACHE bit. Lack of interest. Could come back.
- o Deleted option to transparently split an fs. Could come back if

there is a groundswell of support.

- o Re-organized some of the continuity information in a way that I thinks adds clarity to this. Got rid of same-fs and now have explicitly server, endpoint, and sharing classes, which is a lot clearer. In doing this, I adopted a straight session orientation, although I have not had time to update this document to fully reflect the working group's decisions to make sessions mandatory.

This document proposes the following changes relative to the current draft of the spec for NFSv4 Minor Version 1 ([draft-ietf-nfsv4-minorversion1-02.txt](#) [2]):

Noveck

Expires December 21, 2006

[Page 3]

Internet-Draft

Location Chapters for 4.1

June 2006

- o Deletion of the current chapter 4 (Filesystem Migration and Replication) from that draft.
- o Replacement of the current chapter 5 (NFS Server Name Space) by the second chapter of this document (Single-server Name Space).
- o Deletion of [section 6.14](#) (Migration, Replication and State) from the current v4.1 draft.
- o Addition to the draft of the third chapter of this document, (Multi-server Name Space) following the current chapter 10 of the v4.1 drafts (NFSv4.1 Sessions). This provides the replacement (rewritten to reflect referrals and other changes of the current draft's) chapter 4 and [section 6.14](#).
- o Minor changes to other areas of the spec as set fourth in the fourth chapter of this document (Other Spec Changes Needed).

[2.](#) Single-server Name Space

This chapter describes the NFSv4 single-server name space. Single-server namespaces may be presented directly to clients, or they may be used as a basis to form larger multi-server namespaces (e.g. site-wide or organization-wide) to be presented to clients, as described in [Section 3](#).

[2.1.](#) Server Exports

On a UNIX server, the name space describes all the files reachable by pathnames under the root directory or "/". On a Windows NT server the name space constitutes all the files on disks named by mapped disk letters. NFS server administrators rarely make the entire server's filesystem name space available to NFS clients. More often portions of the name space are made available via an "export" feature. In previous versions of the NFS protocol, the root filehandle for each export is obtained through the MOUNT protocol; the client sends a string that identifies the export of name space and the server returns the root filehandle for it. The MOUNT protocol supports an EXPORTS procedure that will enumerate the server's exports.

[2.2.](#) Browsing Exports

The NFS version 4 protocol provides a root filehandle that clients can use to obtain filehandles for the exports of a particular server, via a series of LOOKUP operations within a COMPOUND, to traverse a path. A common user experience is to use a graphical user interface (perhaps a file "Open" dialog window) to find a file via progressive browsing through a directory tree. The client must be able to move from one export to another export via single-component, progressive LOOKUP operations.

This style of browsing is not well supported by the NFS version 2 and 3 protocols. The client expects all LOOKUP operations to remain within a single server filesystem. For example, the device attribute will not change. This prevents a client from taking name space paths that span exports.

An automounter on the client can obtain a snapshot of the server's name space using the EXPORTS procedure of the MOUNT protocol. If it understands the server's pathname syntax, it can create an image of the server's name space on the client. The parts of the name space that are not exported by the server are filled in with a "pseudo filesystem" that allows the user to browse from one mounted filesystem to another. There is a drawback to this representation of the server's name space on the client: it is static. If the server

administrator adds a new export the client will be unaware of it.

[2.3.](#) Server Pseudo Filesystem

NFS version 4 servers avoid this name space inconsistency by presenting all the exports for a given server within the framework of a single namespace, for that server. An NFS version 4 client uses LOOKUP and READDIR operations to browse seamlessly from one export to another. Portions of the server name space that are not exported are bridged via a "pseudo filesystem" that provides a view of exported directories only. A pseudo filesystem has a unique fsid and behaves like a normal, read only filesystem.

Based on the construction of the server's name space, it is possible that multiple pseudo filesystems may exist. For example,

/a	pseudo filesystem
/a/b	real filesystem
/a/b/c	pseudo filesystem
/a/b/c/d	real filesystem

Each of the pseudo filesystems are considered separate entities and therefore will have its own unique fsid.

[2.4.](#) Multiple Roots

The DOS and Windows operating environments are sometimes described as having "multiple roots". Filesystems are commonly represented as disk letters. MacOS represents filesystems as top level names. NFS version 4 servers for these platforms can construct a pseudo file system above these root names so that disk letters or volume names are simply directory names in the pseudo root.

[2.5.](#) Filehandle Volatility

The nature of the server's pseudo filesystem is that it is a logical representation of filesystem(s) available from the server. Therefore, the pseudo filesystem is most likely constructed dynamically when the server is first instantiated. It is expected that the pseudo filesystem may not have an on disk counterpart from which persistent filehandles could be constructed. Even though it is preferable that the server provide persistent filehandles for the pseudo filesystem, the NFS client should expect that pseudo file system filehandles are volatile. This can be confirmed by checking the associated "fh_expire_type" attribute for those filehandles in question. If the filehandles are volatile, the NFS client must be prepared to recover a filehandle value (e.g. with a series of LOOKUP operations) when receiving an error of NFS4ERR_FHEXPIRED.

[2.6.](#) Exported Root

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

/	disk1	(exported)
/a	disk2	(not exported)

/a/b disk3 (exported)

Because disk2 is not exported, disk3 cannot be reached with simple LOOKUPS. The server must bridge the gap with a pseudo-filesystem.

[2.7.](#) Mount Point Crossing

The server filesystem environment may be constructed in such a way that one filesystem contains a directory which is 'covered' or mounted upon by a second filesystem. For example:

/a/b (filesystem 1)
/a/b/c/d (filesystem 2)

The pseudo filesystem for this server may be constructed to look like:

/ (place holder/not exported)
/a/b (filesystem 1)
/a/b/c/d (filesystem 2)

It is the server's responsibility to present the pseudo filesystem that is complete to the client. If the client sends a lookup request for the path "/a/b/c/d", the server's response is the filehandle of the filesystem "/a/b/c/d". In previous versions of the NFS protocol, the server would respond with the filehandle of directory "/a/b/c/d" within the filesystem "/a/b".

The NFS client will be able to determine if it crosses a server mount point by a change in the value of the "fsid" attribute.

[2.8.](#) Security Policy and Name Space Presentation

The application of the server's security policy needs to be carefully considered by the implementor. One may choose to limit the viewability of portions of the pseudo filesystem based on the server's perception of the client's ability to authenticate itself properly. However, with the support of multiple security mechanisms and the ability to negotiate the appropriate use of these mechanisms, the server is unable to properly determine if a client will be able

to authenticate itself. If, based on its policies, the server

chooses to limit the contents of the pseudo filesystem, the server may effectively hide filesystems from a client that may otherwise have legitimate access.

As suggested practice, the server should apply the security policy of a shared resource in the server's namespace to the components of the resource's ancestors. For example:

```
/
/a/b
/a/b/c
```

The /a/b/c directory is a real filesystem and is the shared resource. The security policy for /a/b/c is Kerberos with integrity. The server should apply the same security policy to /, /a, and /a/b. This allows for the extension of the protection of the server's namespace to the ancestors of the real shared resource.

For the case of the use of multiple, disjoint security mechanisms in the server's resources, the security for a particular object in the server's namespace should be the union of all security mechanisms of all direct descendants.

[3.](#) Multi-server Name Space

NFSv4.1 supports attributes that allow a namespace to extend beyond the boundaries of a single server. Use of such multi-server namespaces is optional, and for many purposes, single-server namespaces are perfectly acceptable. Use of multi-server namespaces can provide many advantages, however, by separating a file system's logical position in a name space from the (possibly changing) logistical and administrative considerations that result in particular file systems being located on particular servers.

[3.1.](#) Location attributes

NFSv4 contains recommended attributes that allow file systems on one server to be associated with one or more instances of that file system on other servers. These attributes specify such file systems by specifying a server name (either a DNS name or an IP address) together with the path of that filesystem within that server's single-server name space.

The `fs_locations_info` recommended attribute allows specification of one more file systems locations where the data corresponding to a given file system may be found. This attributes provides to the client, in addition to information about file system locations, extensive information about the various file system choices (e.g. priority for use, writability, currency, etc.) as well as information to help the client efficiently effect as seamless a transition as possible among multiple file system instances, when and if that should be necessary.

The `fs_locations` recommended attribute is inherited from NFSv4.0 and only allows specification of the file system locations where the data corresponding to a given file system may be found. Servers should make this attribute available whenever `fs_locations_info` is supported, but client use of `fs_locations_info` is to be preferred.

[3.2.](#) File System Presence or Absence

A given location in an NFSv4 namespace (typically but not necessarily a multi-server namespace) can have a number of file system locations associated with it (via the `fs_locations` or `fs_locations_info` attribute). There may also be an actual current file system at that location, accessible via normal namespace operations (e.g. LOOKUP). In this case there, the file system is said to be "present" at that position in the namespace and clients will typically use it, reserving use of additional locations specified via the location-

related attributes to situations in which the principal location is no longer available.

When there is no actual filesystem at the namespace location in question, the file system is said to be "absent". An absent file system contains no files or directories other than the root and any reference to it, except to access a small set of attributes useful in determining alternate locations, will result in an error, NFS4ERR_MOVED. Note that if the server ever returns NFS4ERR_MOVED (i.e. file systems may be absent), it **MUST** support the fs_locations attribute and **SHOULD** support the fs_locations_info and fs_absent attributes.

While the error name suggests that we have a case of a file system which once was present, and has only become absent later, this is only one possibility. A position in the namespace may be permanently absent with the file system(s) designated by the location attributes the only realization. The name NFS4ERR_MOVED reflects an earlier, more limited conception of its function, but this error will be returned whenever the referenced file system is absent, whether it has moved or not.

Except in the case of GETATTR-type operations (to be discussed later), when the current filehandle at the start of an operation is within an absent file system, that operation is not performed and the error NFS4ERR_MOVED returned, to indicate that the filesystem is absent on the current server.

Because a GETFH cannot succeed, if the current filehandle is within an absent file system, filehandles within an absent filesystem cannot be transferred to the client. When a client does have filehandles within an absent file system, it is the result of obtaining them when the file system was present, and having the file system become absent subsequently.

It should be noted that because the check for the current filehandle being within an absent filesystem happens at the start of every operation, operations which change the current filehandle so that it is within an absent filesystem will not result in an error. This allows such combinations as PUTFH-GETATTR and LOOKUP-GETATTR to be used to get attribute information, particularly location attribute information, as discussed below.

The recommended file system attribute `fs_absent` can be used to interrogate the present/absent status of a given file system.

[3.3.](#) Getting Attributes for an Absent File System

When a file system is absent, most attributes are not available, but it is necessary to allow the client access to the small set of attributes that are available, and most particularly those that give

Noveck

Expires December 21, 2006

[Page 10]

Internet-Draft

Location Chapters for 4.1

June 2006

information about the correct current locations for this file system, `fs_locations` and `fs_locations_info`.

[3.3.1.](#) GETATTR Within an Absent File System

As mentioned above, an exception is made for GETATTR in that attributes may be obtained for a filehandle within an absent file system. This exception only applies if the attribute mask contains at least one attribute bit that indicates the client is interested in a result regarding an absent file system: `fs_locations`, `fs_locations_info`, or `fs_absent`. If none of these attributes is requested, GETATTR will result in an NFS4ERR_MOVED error.

When a GETATTR is done on an absent file system, the set of supported attributes is very limited. Many attributes, including those that are normally mandatory will not be available on an absent file system. In addition to the attributes mentioned above (`fs_locations`, `fs_locations_info`, `fs_absent`), the following attributes SHOULD be available on absent file systems, in the case of recommended attributes at least to the same degree that they are available on present file systems.

change: This attribute is useful for absent file systems and can be helpful in summarizing to the client when any of the location-related attributes changes.

fsid: This attribute should be provided so that the client can determine file system boundaries, including, in particular, the boundary between present and absent file systems.

mounted_on_fileid: For objects at the top of an absent file system this attribute needs to be available. Since the fileid is one

which is within the present parent file system, there should be no need to reference the absent file system to provide this information.

Other attributes SHOULD NOT be made available for absent file systems, even when it is possible to provide them. The server should not assume that more information is always better and should avoid gratuitously providing additional information.

When a GETATTR operation includes a bit mask for one of the attributes `fs_locations`, `fs_locations_info`, or `absent`, but where the bit mask includes attributes which are not supported, GETATTR will not return an error, but will return the mask of the actual attributes supported with the results.

Handling of VERIFY/NVERIFY is similar to GETATTR in that if the

attribute mask does not include `fs_locations`, `fs_locations_info`, or `absent`, the error `NFS4ERR_MOVED` will result. It differs in that any appearance in the attribute mask of an attribute not supported for an absent file system (and note that this will include some normally mandatory attributes), will also cause an `NFS4ERR_MOVED` result.

[3.3.2.](#) READDIR and Absent File Systems

A READDIR performed when the current filehandle is within an absent file system will result in an `NFS4ERR_MOVED` error, since, unlike the case of GETATTR, no such exception is made for READDIR.

Attributes for an absent file system may be fetched via a READDIR for a directory in a present file system, when that directory contains the root directories of one or more absent filesystems. In this case, the handling is as follows:

- o If the attribute set requested includes one of the attributes `fs_locations`, `fs_locations_info`, or `absent`, then fetching of attributes proceeds normally and no `NFS4ERR_MOVED` indication is returned, even when the `rdattr_error` attribute is requested.
- o If the attribute set requested does not include one of the attributes `fs_locations`, `fs_locations_info`, or `fs_absent`, then if the `rdattr_error` attribute is requested, each directory entry for

the root of an absent file system, will report NFS4ERR_MOVED as the value of the rdattn_error attribute.

- o If the attribute set requested does not include any of the attributes fs_locations, fs_locations_info, fs_absent, or rdattn_error then the occurrence of the root of an absent file system within the directory will result in the REaddir failing with an NFSER_MOVED error.
- o The unavailability of an attribute because of a file system's absence, even one that is ordinarily mandatory, does not result in any error indication. The set of attributes returned for the root directory of the absent filesystem in that case is simply restricted to those actually available.

[3.4.](#) Uses of Location Information

The location-bearing attributes (fs_locations and fs_locations_info), provide, together with the possibility of absent filesystems, a number of important facilities in providing reliable, manageable, and scalable data access.

When a file system is present, these attribute can provide

alternative locations, to be used to access the same data, in the event that server failures, communications problems, or other difficulties, make continued access to the current file system impossible or otherwise impractical. Provision of such alternate locations is referred to as "replication" although there are cases in which replicated sets of data are not in fact present, and the replicas are instead different paths to the same data.

When a file system is present and becomes absent, clients can be given the opportunity to have continued access to their data, at an alternate location. In this case, a continued attempt to use the data in the now-absent file system will result in an NFSERR_MOVED error and at that point the successor locations (typically only one but multiple choices are possible) can be fetched and used to continue access. Transfer of the file system contents to the new location is referred to as "migration", but it should be kept in mind that there are cases in which this term can be used, like "replication" when there is no actual data migration per se.

Where a file system was not previously present, specification of file system location provides a means by which file systems located on one server can be associated with a name space defined by another server, thus allowing a general multi-server namespace facility. Designation of such a location, in place of an absent filesystem, is called "referral".

[3.4.1.](#) File System Replication

The `fs_locations` and `fs_locations_info` attributes provide alternative locations, to be used to access data in place of the current file system. On first access to a filesystem, the client should obtain the value of the set alternate locations by interrogating the `fs_locations` or `fs_locations_info` attribute, with the latter being preferred.

In the event that server failures, communications problems, or other difficulties, make continued access to the current file system impossible or otherwise impractical, the client can use the alternate locations as a way to get continued access to his data.

The alternate locations may be physical replicas of the (typically read-only) file system data, or they may reflect alternate paths to the same server or provide for the use of various form of server clustering in which multiple servers provide alternate ways of accessing the same physical file system. How these different modes of file system transition are represented within the `fs_locations` and `fs_locations_info` attributes and how the client deals with file system transition issues will be discussed in detail below.

[3.4.2.](#) File System Migration

When a file system is present and becomes absent, clients can be given the opportunity to have continued access to their data, at an alternate location, as specified by the `fs_locations` or `fs_locations_info` attribute. Typically, a client will be accessing the file system in question, get a an `NFS4ERR_MOVED` error, and then use the `fs_locations` or `fs_locations_info` attribute to determine the new location of the data. When `fs_locations_info` is used, additional information will be available which will define the nature of the client's handling of the transition to a new server.

Such migration can be helpful in providing load balancing or general resource reallocation. The protocol does not specify how the filesystem will be moved between servers. It is anticipated that a number of different server-to-server transfer mechanisms might be used with the choice left to the server implementor. The NFSv4.1 protocol specifies the method used to communicate the migration event between client and server.

The new location may be an alternate communication path to the same server, or, in the case of various forms of server clustering, another server providing access to the same physical file system. The client's responsibilities in dealing with this transition depend on the specific nature of the new access path and how and whether data was in fact migrated. These issues will be discussed in detail below.

Although a single successor location is typical, multiple locations may be provided, together with information that allows priority among the choices to be indicated, via information in the `fs_locations_info` attribute. Where suitable clustering mechanisms make it possible to provide multiple identical file systems or paths to them, this allows the client the opportunity to deal with any resource or communications issues that might limit data availability.

[3.4.3.](#) Referrals

Referrals provide a way of placing a file system in a location essentially without respect to its physical location on a given server. This allows a single server or a set of servers to present a multi-server namespace that encompasses filesystems located on multiple servers. Some likely uses of this include establishment of site-wide or organization-wide namespaces, or even knitting such together into a truly global namespace.

Referrals occur when a client determines, upon first referencing a position in the current namespace, that it is part of a new file

system and that that file system is absent. When this occurs, typically by receiving the error `NFS4ERR_MOVED`, the actual location or locations of the file system can be determined by fetching the `fs_locations` or `fs_locations_info` attribute.

Use of multi-server namespaces is enabled by NFSv4 but is not required. The use of multi-server namespaces and their scope will depend on the application used, and system administration preferences.

Multi-server namespaces can be established by a single server providing a large set of referrals to all of the included filesystems. Alternatively, a single multi-server namespace may be administratively segmented with separate referral file systems (on separate servers) for each separately-administered section of the name space. Any segment or the top-level referral file system may use replicated referral file systems for higher availability.

[3.5.](#) Additional Client-side Considerations

When clients make use of servers that implement referrals and migration, care should be taken so that a user who mounts a given filesystem that includes a referral or a relocated filesystem continue to see a coherent picture of that user-side filesystem despite the fact that it contains a number of server-side filesystems which may be on different servers.

One important issue is upward navigation from the root of a server-side filesystem to its parent (specified as ".." in UNIX). The client needs to determine when it hits an fsid root going up the filetree. When at such a point, and needs to ascend to the parent, it must do so locally instead of sending a LOOKUPP call to the server. The LOOKUPP would normally return the ancestor of the target filesystem on the target server, which may not be part of the space that the client mounted.

Another issue concerns refresh of referral locations. When referrals are used extensively, they may change as server configurations change. It is expected that clients will cache information related to traversing referrals so that future client side requests are resolved locally without server communication. This is usually rooted in client-side name lookup caching. Clients should periodically purge this data for referral points in order to detect changes in location information. When the change attribute changes for directories that hold referral entries or for the referral entries themselves, clients should consider any associated cached referral information to be out of date.

3.6. Effecting File System Transitions

Transitions between file system instances, whether due to switching between replicas upon server unavailability, or in response to a server-initiated migration event are best dealt with together. Even though the prototypical use cases of replication and migration contain distinctive sets of features, when all possibilities for these operations are considered, the underlying unity of these operations, from the client's point of view is clear, even though for the server pragmatic considerations will normally force different implementation strategies for planned and unplanned transitions.

A number of methods are possible for servers to replicate data and to track client state in order to allow clients to transition between file system instances with a minimum of disruption. Such methods vary between those that use inter-server clustering techniques to limit the changes seen by the client, to those that are less aggressive, use more standard methods of replicating data, and impose a greater burden on the client to adapt to the transition.

The NFSv4.1 protocol does not impose choices on clients and servers with regard to that spectrum of transition methods. In fact, there are many valid choices, depending on client and application requirements and their interaction with server implementation choices. The NFSv4.1 protocol does define the specific choices that can be made, how these choices are communicated to the client and how the client is to deal with any discontinuities.

In the sections below references will be made to various possible server implementation choices as a way of illustrating the transition scenarios that clients may deal with. The intent here is not to define or limit server implementations but rather to illustrate the range of issues that clients may face.

In the discussion below, references will be made to a file system having a particular property or of two file systems (typically the source and destination) belonging to a common class of any of several types. Two file systems that belong to such a class share some important aspect of file system behavior that clients may depend upon when present, to easily effect a seamless transition between file system instances. Conversely, where the file systems do not belong to such a common class, the client has to deal with various sorts of implementation discontinuities which may cause performance or other issues in effecting a transition.

Where the `fs_locations_info` attribute is available, such file system classification data will be made directly available to the client.

See [Section 3.10](#) for details. When only `fs_locations` is available,

default assumptions with regard to such classifications have to be inferred. See [Section 3.9](#) for details.

In cases in which one server is expected to accept opaque values from the client that originated from another server, it is a wise implementation practice for the servers to encode the "opaque" values in network byte order. If this is done, 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.

[3.6.1](#). Transparent File System Transitions

Discussion of transition possibilities will start at the most transparent end of the spectrum of possibilities. When there are multiple paths to a single server, and there are network problems that force another path to be used, or when a path is to be put out of service, a replication or migration event may occur without any real replication or migration. Nevertheless, such events fit within the same general framework in that there is a transition between file system locations, communicated just as other, less transparent transitions are communicated.

There are cases of transparent transitions that may happen independent of location information, in that a specific host name, may map to several IP addresses, allowing session trunking to provide alternate paths. In other cases, however multiple addresses may have separate location entries for specific file systems to preferentially direct traffic for those specific file systems to certain server addresses, subject to planned or unplanned, corresponding to a nominal replication or migrations event.

The specific details of the transition depend on file system equivalence class information (as provided by the `fs_locations_info` and `fs_locations` attributes).

- o Where the old and new filesystems belong to the same `_endpoint_` class, the transition consists of creating a new connection which is associated with the existing session to the old server

endpoint. Where a connection cannot be associated with the existing session, the target server must be able to recognize the sessionid as invalid and force creation on a new session or a new client id.

- o Where the old and new filesystems do not belong to the same `_endpoint_` classes, but to the same `_server_` class, the transition consists of creating a new session, associated with the existing

clientid. Where the clientid is stale, the stale, the target server must be able to recognize the clientid as no longer valid and force creation of a new clientid.

In either of the above cases, the file system may be shown as belonging to the same `_sharing_` class, class allowing the alternate session or connection to be established in advance and used either to accelerate the file system transition when necessary (avoiding connection latency), or to provide higher performance by actively using multiple paths simultaneously.

When two file systems belong to the same `_endpoint_` class, or `_sharing_` class, many transition issues are eliminated, and any information indicating otherwise is ignored as erroneous.

In all such transparent transition cases, the following apply:

- o File handles stay the same if persistent and if volatile are only subject to expiration, if they would be in the absence of file system transition.
- o Fileid values do not change across the transition.
- o The file system will have the same fsid in both the old and new the old and new locations.
- o Change attribute values are consistent across the transition and do not have to be refetched. When change attributes indicate that a cached object is still valid, it can remain cached.
- o Session, client, and state identifier retain their validity across the transition, except where their staleness is recognized and reported by the new server. Except where such staleness requires

it, no lock reclamation is needed.

- o Write verifiers are presumed to retain their validity and can be presented to COMMIT, with the expectation that if COMMIT on the new server accept them as valid, then that server has all of the data unstably written to the original server and has committed it to stable storage as requested.

[3.6.2.](#) Filehandles and File System Transitions

There are a number of ways in which filehandles can be handled across a file system transition. These can be divided into two broad classes depending upon whether the two file systems across which the transition happens share sufficient state to effect some sort of continuity of filesystem handling.

Noveck

Expires December 21, 2006

[Page 18]

Internet-Draft

Location Chapters for 4.1

June 2006

When there is no such co-operation in filehandle assignment, the two file systems are reported as being in different `_handle_` classes. In this case, all filehandles are assumed to expire as part of the file system transition. Note that this behavior does not depend on `fh_expire_type` attribute and supersedes the specification of `FH4_VOL_MIGRATION` bit, which only affects behavior when `fs_locations_info` is not available.

When there is co-operation in filehandle assignment, the two file systems are reported as being in the same `_handle_` classes. In this case, persistent filehandle remain valid after the file system transition, while volatile filehandles (excluding those while are only volatile due to the `FH4_VOL_MIGRATION` bit) are subject to expiration on the target server.

[3.6.3.](#) Fileid's and File System Transitions

In NFSv4.0, the issue of continuity of fileid's in the event of a file system transition was not addressed. The general expectation had been that in situations in which the two filesystem instances are created by a single vendor using some sort of filesystem image copy, fileid's will be consistent across the transition while in the analogous multi-vendor transitions they will not. This poses difficulties, especially for the client without special knowledge of the of the transition mechanisms adopted by the server.

It is important to note that while clients themselves may have no trouble with a fileid changing as a result of a file system transition event, applications do typically have access to the fileid (e.g. via stat), and the result of this is that an application may work perfectly well if there is no filesystem instance transition or if any such transition is among instances created by a single vendor, yet be unable to deal with the situation in which a multi-vendor transition occurs, at the wrong time.

Providing the same fileid's in a multi-vendor (multiple server vendors) environment has generally been held to be quite difficult. While there is work to be done, it needs to be pointed out that this difficulty is partly self-imposed. Servers have typically identified fileid with inode number, i.e. with a quantity used to find the file in question. This identification poses special difficulties for migration of an fs between vendors where assigning the same index to a given file may not be possible. Note here that a fileid does not require that it be useful to find the file in question, only that it is unique within the given fs. Servers prepared to accept a fileid as a single piece of metadata and store it apart from the value used to index the file information can relatively easily maintain a fileid value across a migration event, allowing a truly transparent

migration event.

In any case, where servers can provide continuity of fileids, they should and the client should be able to find out that such continuity is available, and take appropriate action. Information about the continuity (or lack thereof) of fileid's across a file system is represented by specifying whether the file systems in question are of the same `_fileid_` class.

[3.6.4.](#) Fsid's and File System Transitions

Since fsid's are only unique within a per-server basis, it is to be expected that they will change during a file system transition. Clients should not make the fsid's received from the server visible to application since they may not be globally unique, and because they may change during a file system transition event. Applications are best served if they are isolated from such transitions to the extent possible.

[3.6.5.](#) The Change Attribute and File System Transitions

Since the change attribute is defined as a server-specific one, change attributes fetched from one server are normally presumed to be invalid on another server. Such a presumption is troublesome since it would invalidate all cached change attributes, requiring refetching. Even more disruptive, the absence of any assured continuity for the change attribute means that even if the same value is gotten on refetch no conclusions can be drawn as to whether the object in question has changed. The identical change attribute could be merely an artifact, of a modified file with a different change attribute construction algorithm, with that new algorithm just happening to result in an identical change value.

When the two file systems have consistent change attribute formats, and this fact is communicated to the client by reporting as in the same `_change_` class, the client may assume a continuity of change attribute construction and handle this situation just as it would be handled without any filesystem transition.

[3.6.6.](#) Lock State and File System Transitions

In a file system transition, the two file systems may have co-operated in state management. When this is the case, and the two file systems belong to the same `_state_` class, the two file systems will have compatible state environments. In the case of migration, the servers involved in the migration of a filesystem SHOULD transfer all server state from the original to the new server. When this is done, it must be done in a way that is transparent to the client.

With replication, such a degree of common state is typically not the case. Clients, however should use the information provided by the `fs_locations_info` attribute to determine whether such sharing is in effect when this is available, and only if that attribute is not available depend on these defaults.

This state transfer will reduce disruption to the client when a file system transition. 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. This holds true for the clientid as well. Since responsibility for an entire filesystem is transferred with such

an event, there is no possibility that conflicts will arise on the new server as a result of the transfer of locks.

As part of the transfer of information between servers, leases would be transferred as well. The leases being transferred to the new server will typically have a different expiration time from those for the same client, previously on the old server. To maintain the property that all leases on a given server for a given client expire at the same time, the server should advance the expiration time to the later of the leases being transferred or the leases already present. This allows the client to maintain lease renewal of both classes without special effort.

When the two servers belong to the same `_state_` class, it does not necessarily mean that when dealing with the transition, the client will not have to reclaim state. However it does mean that the client may proceed using his current `clientid` and `stateid`'s just as if there had been no file system transition event and only reclaim state when an `NFS4ERR_STALE_CLIENTID` or `NFS4ERR_STALE_STATEID` error is received.

File systems co-operating in state management may actually share state or simply divide the id space so as to recognize (and reject as stale) each others state and clients id's. Servers which do share state may not do under all conditions or all times. The requirement for the server is that if it cannot be sure in accepting an id that it reflects the locks the client was given, it must treat all associated state as stale and report it as such to the client.

When two file systems belong to different `_state_` classes, the client must establish a new state on the destination, and reclaim if possible. In this case, old `stateids` and `clientid`'s should not be presented to the new server since there is no assurance that they will not conflict with id's valid on that server.

In either case, when actual locks are not known to be maintained, the destination server may establish a grace period specific to the given

file system, with non-reclaim locks being rejected for that file system, even though normal locks are being granted for other file systems. Clients should not infer the absence of a grace period for file systems being transitioned to a server from responses to requests for other file systems.

In the case of lock reclamation for a given file system after a file system transition, edge conditions can arise similar to those for reclaim after server reboot (although in the case of the planned state transfer associated with migration, these can be avoided by securely recording lock state as part of state migration. Where the destination server cannot guarantee that locks will not be incorrectly granted, the destination server should not establish a file-system-specific grace period.

In place of a file-system-specific version of RECLAIM_COMPLETE, servers may assume that an attempt to obtain a new lock, other than be reclaim, indicate the end of the client's attempt to reclaim locks for that file system. [NOTE: The alternative would be to adapt RECLAIM_COMPLETE to this task].

Information about client identity that may be propagated between servers in the form of nfs_client_id4 and associated verifiers, under the assumption that the client presents the same values to all the servers with which it deals. [NOTE: This contradicts what is currently said about SETCLIENTID, and interacts with the issue of what sessions should do about this.]

Servers are encouraged to provide facilities to allow locks to be reclaimed on the new server after a file system transition. Often, however, in cases in which the two file systems are not of the same _state _ class, such facilities may not be available and client should be prepared to re-obtain locks, even though it is possible that the client may have his LOCK or OPEN request denied due to a conflicting lock. In some environments, such as the transition between read-only file systems, such denial of locks should not pose large difficulties in practice. When an attempt to re-establish a lock on a new server is denied, the client should treat the situation as if his original lock had been revoked. In all cases in which the lock is granted, the client cannot assume that no conflicting could have been granted in the interim. Where change attribute continuity is present, the client may check the change attribute to check for unwanted file modifications. Where even this is not available, and the file system is not read-only a client may reasonably treat all pending locks as having been revoked.

[3.6.6.1.](#) Leases and File System Transitions

In the case of lease renewal, the client may not be submitting requests for a filesystem that has been transferred to another server. This can occur because of the lease renewal mechanism. The client renews leases for all filesystems when submitting a request to any one filesystem at the 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. To accomplish this, all operations which renew leases for a client (i.e. OPEN, CLOSE, READ, WRITE, RENEW, LOCK, LOCKT, LOCKU), will return the error NFS4ERR_LEASE_MOVED if responsibility for any of the leases to be renewed has been transferred to a new server. This condition will continue until the client receives an NFS4ERR_MOVED error and the server receives the subsequent GETATTR for the fs_locations or fs_locations_info attribute for an access to each filesystem for which a lease has been moved to a new server.

[ISSUE: There is a conflict between this and the idea in the sessions text that we can have every op in the session implicitly renew the lease. This needs to be dealt with. D. Noveck will create an issue in the issue tracker.]

When a client receives an NFS4ERR_LEASE_MOVED error, it should perform an operation on each filesystem associated with the server in question. When the client receives an NFS4ERR_MOVED error, the client can follow the normal process to obtain the new server information (through the fs_locations and fs_locations_info attributes) and perform renewal of those leases on the new server, unless information in fs_locations_info attribute shows that no state could have been transferred. If the server has not had state transferred to it transparently, the client will receive either NFS4ERR_STALE_CLIENTID or NFS4ERR_STALE_STATEID from the new server, as described above, and the client can then recover state information as it does in the event of server failure.

[3.6.6.2.](#) Transitions and the Lease_time Attribute

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

When state is transferred transparently, that state should include the correct value of the lease_time attribute. The lease_time attribute on the destination server must never be less than that on the source since this would result in premature expiration of leases

granted by the source server. Upon transitions 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).

If state has not been transferred transparently, either because the file systems are shown as being in different state classes or because the client sees a real or simulated server reboot), 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 lock before potentially conflicting non-reclaimed locks are granted.

[3.6.7.](#) Write Verifiers and File System Transitions

In a file system transition, the two file systems may be clustered in the handling of unstably written data. When this is the case, and the two file systems belong to the same `_verifier_` class, valid verifiers from one system may be recognized by the other and superfluous writes avoided. There is no requirement that all valid verifiers be recognized, but it cannot be the case that a verifier is recognized as valid when it is not. [NOTE: We need to resolve the issue of proper verifier scope].

When two file systems belong to different `_verifier_` classes, the client must assume that all unstable writes in existence at the time file system transition, have been lost since there is no way the old verifier can be recognized as valid (or not) on the target server.

[3.7.](#) Effecting File System Referrals

Referrals are effected when an absent file system is encountered, and one or more alternate locations are made available by the `fs_locations` or `fs_locations_info` attributes. The client will typically get an `NFS4ERR_MOVED` error, fetch the appropriate location information and proceed to access the file system on different server, even though it retains its logical position within the original namespace.

The examples given in the sections below are somewhat artificial in

that an actual client will not typically do a multi-component lookup, but will have cached information regarding the upper levels of the name hierarchy. However, these example are chosen to make the required behavior clear and easy to put within the scope of a small number of requests, without getting unduly into details of how specific clients might choose to cache things.

[3.7.1.](#) Referral Example (LOOKUP)

Let us suppose that the following COMPOUND is issued in an environment in which `/src/linux/2.7/latest` is absent from the target server. This may be for a number of reasons. It may be the case that the file system has moved, or, it may be the case that the target server is functioning mainly, or solely, to refer clients to the servers on which various file systems are located.

- o PUTROOTFH
- o LOOKUP "src"
- o LOOKUP "linux"
- o LOOKUP "2.7"
- o LOOKUP "latest"
- o GETFH
- o GETATTR fsid,fileid,size,ctime

Under the given circumstances, the following will be the result.

- o PUTROOTFH --> NFS_OK. The current fh is now the root of the pseudo-fs.
- o LOOKUP "src" --> NFS_OK. The current fh is for `/src` and is within the pseudo-fs.
- o LOOKUP "linux" --> NFS_OK. The current fh is for `/src/linux` and is within the pseudo-fs.
- o LOOKUP "2.7" --> NFS_OK. The current fh is for `/src/linux/2.7` and

is within the pseudo-fs.

- o LOOKUP "latest" --> NFS_OK. The current fh is for /src/linux/2.7/latest and is within a new, absent fs, but ... the client will never see the value of that fh.
- o GETFH --> NFS4ERR_MOVED. Fails because current fh is in an absent fs at the start of the operation and the spec makes no exception for GETFH.
- o GETATTR fsid,fileid,size,ctime. Not executed because the failure of the GETFH stops processing of the COMPOUND.

Given the failure of the GETFH, the client has the job of determining the root of the absent file system and where to find that file system, i.e. the server and path relative to that server's root fh. Note here that in this example, the client did not obtain filehandles and attribute information (e.g. fsid) for the intermediate directories, so that he would not be sure where the absent file system starts. It could be the case, for example, that /src/linux/2.7 is the root of the moved filesystem and that the reason that the lookup of "latest" succeeded is that the filesystem was not absent on that op but was moved between the last LOOKUP and the GETFH (since COMPOUND is not atomic). Even if we had the fsid's for all of the intermediate directories, we could have no way of knowing that /src/linux/2.7/latest was the root of a new fs, since we don't yet have its fsid.

In order to get the necessary information, let us re-issue the chain of lookup's with GETFH's and GETATTR's to at least get the fsid's so we can be sure where the appropriate fs boundaries are. The client could choose to get fs_locations_info at the same time but in most cases the client will have a good guess as to where fs boundaries are (because of where NFS4ERR_MOVED was gotten and where not) making fetching of fs_locations_info unnecessary.

OP01: PUTROOTFH --> NFS_OK

- Current fh is root of pseudo-fs.

OP02: GETATTR(fsid) --> NFS_OK

- Just for completeness. Normally, clients will know the fsid of the pseudo-fs as soon as they establish communication with a server.

OP03: LOOKUP "src" --> NFS_OK

OP04: GETATTR(fsid) --> NFS_OK

- Get current fsid to see where fs boundaries are. The fsid will be that for the pseudo-fs in this example, so no boundary.

OP05: GETFH --> NFS_OK

- Current fh is for /src and is within pseudo-fs.

OP06: LOOKUP "linux" --> NFS_OK

- Current fh is for /src/linux and is within pseudo-fs.

OP07: GETATTR(fsid) --> NFS_OK

- Get current fsid to see where fs boundaries are. The fsid will be that for the pseudo-fs in this example, so no boundary.

OP08: GETFH --> NFS_OK

- Current fh is for /src/linux and is within pseudo-fs.

OP09: LOOKUP "2.7" --> NFS_OK

- Current fh is for /src/linux/2.7 and is within pseudo-fs.

OP10: GETATTR(fsid) --> NFS_OK

- Get current fsid to see where fs boundaries are. The fsid will be that for the pseudo-fs in this example, so no boundary.

OP11: GETFH --> NFS_OK

- Current fh is for /src/linux/2.7 and is within pseudo-fs.

OP12: LOOKUP "latest" --> NFS_OK

- Current fh is for /src/linux/2.7/latest and is within a new, absent fs, but ...
- The client will never see the value of that fh

OP13: GETATTR(fsid, fs_locations_info) --> NFS_OK

- We are getting the fsid to know where the fs boundaries are. Note that the fsid we are given will not necessarily be preserved at the new location. That fsid might be different and in fact the fsid we have for this fs might be a valid fsid of a different fs on that new server.
- In this particular case, we are pretty sure anyway that what has moved is /src/linux/2.7/latest rather than /src/linux/2.7 since we have the fsid of the latter and it is that of the pseudo-fs, which presumably cannot move. However, in other examples, we might not have this kind of information to rely on (e.g. /src/linux/2.7 might be a non-pseudo filesystem separate from /src/linux/2.7/latest), so we need to have another reliable source of information on the boundary of the fs which is moved. If, for example, the filesystem "/src/linux" had moved we would have a case of

Noveck

Expires December 21, 2006

[Page 27]

Internet-Draft

Location Chapters for 4.1

June 2006

migration rather than referral and once the boundaries of the migrated filesystem were clear we could fetch fs_locations_info.

- We are fetching fs_locations_info because the fact that we got an NFS4ERR_MOVED at this point means that it is most likely that this is a referral and we need the destination. Even if it is the case that "/src/linux/2.7" is a filesystem which has migrated, we will still need the location information for that file system.

OP14: GETFH --> NFS4ERR_MOVED

- Fails because current fh is in an absent fs at the start of the operation and the spec makes no exception for GETFH. Note that this has the happy consequence that we don't have to worry about

the volatility or lack thereof of the fh. If the root of the fs on the new location is a persistent fh, then we can assume that this fh, which we never saw is a persistent fh, which, if we could see it, would exactly match the new fh. At least, there is no evidence to disprove that. On the other hand, if we find a volatile root at the new location, then the filehandle which we never saw must have been volatile or at least nobody can prove otherwise.

Given the above, the client knows where the root of the absent file system is, by noting where the change of fsid occurred. The fs_locations_info attribute also gives the client the actual location of the absent file system, so that the referral can proceed. The server gives the client the bare minimum of information about the absent file system so that there will be very little scope for problems of conflict between information sent by the referring server and information of the file system's home. No filehandles and very few attributes are present on the referring server and the client can treat those it receives as basically transient information with the function of enabling the referral.

[3.7.2.](#) Referral Example (READDIR)

Another context in which a client may encounter referrals is when it does a READDIR on directory in which some of the sub-directories are the roots of absent file systems.

Suppose such a directory is read as follows:

- o PUTROOTFH
- o LOOKUP "src"

- o LOOKUP "linux"
- o LOOKUP "2.7"
- o READDIR (fsid, size, ctime, mounted_on_fileid)

In this case, because rgetattr_error is not requested,

fs_locations_info is not requested, and some of attributes cannot be provided the result will be an NFS4ERR_MOVED error on the READDIR, with the detailed results as follows:

- o PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.
- o LOOKUP "src" --> NFS_OK. The current fh is for /src and is within the pseudo-fs.
- o LOOKUP "linux" --> NFS_OK. The current fh is for /src/linux and is within the pseudo-fs.
- o LOOKUP "2.7" --> NFS_OK. The current fh is for /src/linux/2.7 and is within the pseudo-fs.
- o READDIR (fsid, size, ctime, mounted_on_fileid) --> NFS4ERR_MOVED. Note that the same error would have been returned if /src/linux/2.7 had migrated, when in fact it is because the directory contains the root of an absent fs.

So now suppose that we reissue with rdattrib_error:

- o PUTROOTFH
- o LOOKUP "src"
- o LOOKUP "linux"
- o LOOKUP "2.7"
- o READDIR (rdattrib_error, fsid, size, ctime, mounted_on_fileid)

The results will be:

- o PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.
- o LOOKUP "src" --> NFS_OK. The current fh is for /src and is within the pseudo-fs.

- o LOOKUP "linux" --> NFS_OK. The current fh is for /src/linux and is within the pseudo-fs.
- o LOOKUP "2.7" --> NFS_OK. The current fh is for /src/linux/2.7 and is within the pseudo-fs.
- o REaddir (rdattr_error, fsid, size, ctime, mounted_on_fileid) --> NFS_OK. The attributes for "latest" will only contain rdattr_error with the value will be NFS4ERR_MOVED, together with an fsid value and an a value for mounted_on_fileid.

So suppose we do another REaddir to get fs_locations_info, although we could have used a GETATTR directly, as in the previous section.

- o PUTROOTFH
- o LOOKUP "src"
- o LOOKUP "linux"
- o LOOKUP "2.7"
- o REaddir (rdattr_error, fs_locations_info, mounted_on_fileid, fsid, size, ctime)

The results would be:

- o PUTROOTFH --> NFS_OK. The current fh is at the root of the pseudo-fs.
- o LOOKUP "src" --> NFS_OK. The current fh is for /src and is within the pseudo-fs.
- o LOOKUP "linux" --> NFS_OK. The current fh is for /src/linux and is within the pseudo-fs.
- o LOOKUP "2.7" --> NFS_OK. The current fh is for /src/linux/2.7 and is within the pseudo-fs.
- o REaddir (rdattr_error, fs_locations_info, mounted_on_fileid, fsid, size, ctime) --> NFS_OK. The attributes will be as shown below.

The attributes for "latest" will only contain

- o rdattr_error (value: NFS4ERR_MOVED)
- o fs_locations_info)

- o `mounted_on_fileid` (value: unique fileid within referring fs)
- o `fsid` (value: unique value within referring server)

The attribute entry for "latest" will not contain size or ctime.

[3.8.](#) The Attribute `fs_absent`

In order to provide the client information about whether the current file system is present or absent, the `fs_absent` attribute may be interrogated.

As noted above, this attribute, when supported, may be requested of absent filesystems without causing `NFS4ERR_MOVED` to be returned and it should always be available. Servers are strongly urged to support this attribute on all filesystems if they support it on any filesystem.

[3.9.](#) The Attribute `fs_locations`

The `fs_locations` attribute is structured in the following way:

```
struct fs_location {
    utf8str_cis    server<>;
    pathname4      rootpath;
};

struct fs_locations {
    pathname4      fs_root;
    fs_location    locations<>;
};
```

The `fs_location` struct is used to represent the location of a filesystem by providing a server name and the path to the root of the file system within that server's namespace. When a set of servers have corresponding file systems at the same path within their namespaces, an array of server names may be provided. An entry in the server array is an UTF8 string and represents one of a traditional DNS host name, IPv4 address, or IPv6 address. It is not a requirement that all servers that share the same rootpath be listed in one `fs_location` struct. The array of server names is provided for convenience. Servers that share the same rootpath may also be listed in separate `fs_location` entries in the `fs_locations` attribute.

The `fs_locations` struct and attribute contains an array of such locations. Since the name space of each server may be constructed differently, the "`fs_root`" field is provided. The path represented by `fs_root` represents the location of the filesystem in the current

server's name space, i.e. that of the server from which the `fs_locations` attribute was obtained. The `fs_root` path is meant to aid the client by clearly referencing the root of the file system whose locations are being reported, no matter what object within the current file system, the current filehandle designates.

As an example, suppose there is a replicated filesystem located at two servers (`servA` and `servB`). At `servA`, the filesystem is located at path `"/a/b/c"`. At, `servB` the filesystem is located at path `"/x/y/z"`. If the client were to obtain the `fs_locations` value for the directory at `"/a/b/c/d"`, it might not necessarily know that the filesystem's root is located in `servA`'s name space at `"/a/b/c"`. When the client switches to `servB`, it will need to determine that the directory it first referenced at `servA` is now represented by the path `"/x/y/z/d"` on `servB`. To facilitate this, the `fs_locations` attribute provided by `servA` would have a `fs_root` value of `"/a/b/c"` and two entries in `fs_locations`. One entry in `fs_locations` will be for itself (`servA`) and the other will be for `servB` with a path of `"/x/y/z"`. With this information, the client is able to substitute `"/x/y/z"` for the `"/a/b/c"` at the beginning of its access path and construct `"/x/y/z/d"` to use for the new server.

Since `fs_locations` attribute lacks information defining various attributes of the various file system choices presented, it should only be interrogated and used when `fs_locations_info` is not available. When `fs_locations` is used, information about the specific locations should be assumed based on the following rules.

The following rules are general and apply irrespective of the context.

- o When a DNS server name maps to multiple IP addresses, they should be considered identical, i.e. of the same `_endpoint_` class.
- o Except in the case of servers sharing an `_endpoint_` class, all listed servers should be considered as of the same `_handle_` class,

if and only if, the current `fh_expire_type` attribute does not include the `FH4_VOL_MIGRATION` bit. Note that in the case of referral, filehandle issues do not apply since there can be no filehandles known within the current file system nor is there any access to the `fh_expire_type` attribute on the referring (absent) file system.

- o Except in the case of servers sharing an `_endpoint_` class, all listed servers should be considered as of the same `_fileid_` class, if and only if, the `fh_expire_type` attribute indicates persistent filehandles and does not include the `FH4_VOL_MIGRATION` bit. Note that in the case of referral, fileid issues do not apply since

there can be no fileids known within the referring (absent) file system nor is there any access to the `fh_expire_type` attribute.

- o Except in the case of servers sharing an `_endpoint_` class, all listed servers should be considered as of different `_change_` classes.

For other class assignments, handling depends of file system transitions depends on the reasons for the transition:

- o When the transition is due to migration, the target should be treated as being of the same `_state_` and `_verifier_` class as the source.
- o When the transition is due to failover to another replica, the target should be treated as being of a different `_state_` and `_verifier_` class from the source.

The specific choices reflect typical implementation patterns for failover and controlled migration respectively. Since other choices are possible and useful, this information is better obtained by using `fs_locations_info`.

See the section "Security Considerations" for a discussion on the recommendations for the security flavor to be used by any `GETATTR` operation that requests the `"fs_locations"` attribute.

[3.10](#). The Attribute `fs_locations_info`

The `fs_locations_info` attribute is intended as a more functional replacement for `fs_locations` which will continue to exist and be supported. Clients can use it get a more complete set of information about alternative file system locations. When the server does not support `fs_locations_info`, `fs_locations` can be used to get a subset of the information. A server which supports `fs_locations_info` MUST support `fs_locations` as well.

There are several sorts of additional information present in `fs_locations_info`, that aren't available in `fs_locations`:

- o Attribute continuity information to allow a client to select a location which meets the transparency requirements of the applications accessing the data and to take advantage of optimizations that server guarantees as to attribute continuity may provide (e.g. change attribute).
- o Filesystem identity information which indicates when multiple replicas, from the clients point of view, correspond to the same

target filesystem, allowing them to be used interchangeably, without disruption, as multiple paths to the same thing.

- o Information which will bear on the suitability of various replicas, depending on the use that the client intends. For example, many applications need an absolutely up-to-date copy (e.g. those that write), while others may only need access to the most up-to-date copy reasonably available.
- o Server-derived preference information for replicas, which can be used to implement load-balancing while giving the client the entire `fs` list to be used in case the primary fails.

The `fs_locations_info` attribute consists of a root pathname (just like `fs_locations`), together with an array of `location4_item` structures.

```
struct locations4_server {
    int32_t      currency;
    uint32_t     info<>;
    utf8str_cis  server;
};

const LIBX_GFLAGS      = 0;
const LIBX_TFLAGS      = 1;

const LIBX_CLSHARE     = 2;
const LIBX_CLSERVER    = 3;
const LIBX_CLENDPOINT = 4;
const LIBX_CLHANDLE    = 5;
const LIBX_CLFILEID    = 6;
const LIBX_CLVERIFIER  = 7;
```

```

const LIBX_CLSTATE      = 8;

const LIBX_READRANK     = 9;
const LIBX_WRITERANK    = 10;
const LIBX_READORDER    = 11;
const LIBX_WRITEORDER   = 12;

const LIGF_WRITABLE     = 0x01;
const LIGF_CUR_REQ      = 0x02;
const LIGF_ABSENT       = 0x04;
const LIGF_GOING        = 0x08;

const LITF_RDMA         = 0x01;

struct locations4_item {
    locations4_server entries<>;
    pathname4          rootpath;
};

struct locations4_info {
    pathname4          fs_root;
    locations4_item items<>;
};

```

The `fs_locations_info` attribute is structured similarly to the `fs_locations` attribute. A top-level structure (`fs_locations4` or `locations4_info`) contains the entire attribute including the root pathname of the fs and an array of lower-level structures that define replicas that share a common root path on their respective servers. Those lower-level structures in turn (`fs_locations4` or `location4_item`) contain a specific pathname and information on one or more individual server replicas. For that last lowest-level

information, `fs_locations` has a server name in the form of `utf8str_cis`, while `fs_locations_info` has a `location4_server` structure that contains per-server-replica information in addition to the server name.

The `location4_server` structure consists of the following items:

- o An indication of file system up-to-date-ness (currency) in terms

of approximate seconds before the present. A negative value indicates that the server is unable to give any reasonably useful value here. A zero indicates that filesystem is the actual writable data or a reliably coherent and fully up-to-date copy. Positive values indicate how out- of-date this copy can normally be before it is considered for update. Such a value is not a guarantee that such updates will always be performed on the required schedule but instead serve as a hint about how far behind the most up-to-date copy of the data, this copy would normally be expected to be.

- o A counted array of 32-bit words containing various sorts of data, about the particular file system instance. This data includes general flags, transport capability flags, file system equivalence class information, and selection priority information. The encoding will be discussed below.
- o The server string. For the case of the replica currently being accessed (via GETATTR), a null string may be used to indicate the current address being used for the RPC call.

Data within the info array, is in the form of 8-bit data items even though that array is, from XDR's point of view an array of 32-bit integers. This definition was chosen because:

- o The kinds of data in the info array, representing, flags, file system classes and priorities among set of file systems representing the same data are such that eight bits provides a quite acceptable range of values. Even where there might be more than 256 such file system instances, having more than 256 distinct classes or priorities is unlikely.
- o XDR does not have any means to declare an 8-bit data type, other than an ASCII string, and using 32-bit data types would lead to significant space inefficiency.
- o Explicit definition of the various specific data items within XDR would limit expandability in that any extension within a subsequent minor version would require yet another attribute, leading to specification and implementation clumsiness.

- o Such explicit definitions would also make it impossible to propose

standards-track extensions apart from a full minor version.

Each 8-bit successive field within this array is designated by a constant byte-index as defined above. More significant bit fields within a single word have successive indices with a transition to the next word following the most significant 8-bit field in each word.

The set of info data is subject to expansion in a future minor version, or in a standard-track RFC, within the context of a single minor version. The server **SHOULD NOT** send and the client **MUST** not use indices within the info array that are not defined in standards-track RFC's.

The following fragment of c++ code (with Doxygen-style comments) illustrates how data items within the info array can be found using a byte-index such as specified by the constants beginning with "LIBX_". The associated InfoArray object is assume to be initialized with "Length" containing the XDR-specified length in terms of 32-bit words and "Data" containing the array of words encoded by the "info<>" specification.

```
class InfoArray {
private:
    uint32_t    Length;
    uint32_t    Data[];

public:
    uint8_t     GetValue(int byteIndex);
};

/// @brief Get the value of a locations4_server info value
///
/// This method obtains the specific info value given a
/// byte index defined in the NFSv4.1 spec or another
/// later standards-track document.
///
/// @param[in] byteIndex The byte index identifying the
///                    item requested.
/// @returns The value of the requested item.

uint8_t InfoArray::GetItem(int byteIndex) {

    int         wordIndex = byteIndex/4;
    int         byteWithinWord = byteIndex % 4;

    if (wordIndex >= Length) {
        return (0);
    }

    uint32_t    ourWord = Data[wordIndex];
    return ((ourWord >> (byteWithinWord*8)) & 0xff);
}
```

The info array contains within it:

- o Two 8-bit flag fields, one devoted to general file-system characteristics and a second reserved for transport-related capabilities.
- o Seven 8-bit class values which define various file system equivalence classes as explained below.
- o Four 8-bit priority values which govern file system selection as explained below.

The general file system characteristics flag (at byte index LIBX_GFLAGS) has the following bits defined within it:

- o LIGF_WRITABLE indicates that this fs target is writable, allowing it to be selected by clients which may need to write on this filesystem. When the current filesystem instance is writable, then any other filesystem to which the client might switch must incorporate within its data any committed write made on the current filesystem instance. See the section on verifier class, for issues related to uncommitted writes. While there is no harm in not setting this flag for a filesystem that turns out to be writable, turning the flag on for read-only filesystem can cause problems for clients who select a migration or replication target based on it and then find themselves unable to write.
- o LIGF_CUR_REQ indicates that this replica is the one on which the request is being made. Only a single server entry may have this flag set and in the case of a referral, no entry will have it.
- o LIGF_ABSENT indicates that this entry corresponds an absent filesystem replica. It can only be set if LIGF_CUR_REQ is set. When both such bits are set it indicates that a filesystem instance is not usable but that the information in the entry can be used to determine the sorts of continuity available when switching from this replica to other possible replicas. Since this bit can only be true if LIGF_CUR_REQ is true, the value could be determined using the fs_absent attribute but the information is also made available here for the convenience of the client. An entry with this bit, since it represents a true filesystem (albeit absent) does not appear in the event of a referral, but only where a filesystem has been accessed at this location and subsequently been migrated.
- o LIGF_GOING indicates that a replica, while still available, should not be used further. The client, if using it, should make an orderly transfer to another filesystem instance as expeditiously as possible. It is expected that file systems going out of service will be announced as LIGF_GOING some time before the actual loss of service and that the valid_for value will be sufficiently small to allow servers to detect and act on scheduled events while large enough that the cost of the requests to fetch the fs_locations_info values will not be excessive. Values on the

order of ten minutes seem reasonable.

The transport-flag field (at byte index LIBX_TFLAGS) contains the following bits related to the transport capabilities of the specific file system.

- o LITF_RDMA indicates that this file system provides NFSv4.1 file system access using an RDMA-capable transport.

Noveck

Expires December 21, 2006

[Page 39]

Internet-Draft

Location Chapters for 4.1

June 2006

Attribute continuity and filesystem identity information are expressed by defining equivalence relations on the sets of file systems presented to the client. Each such relation is expressed as a set of file system equivalence classes. For each relation, a file system has an 8-bit class number. Two file systems belong to the same class if both have identical non-zero class numbers. Zero is treated as non-matching. Most often, the relevant question for the client will be whether a given replica is identical-with/continuous-to the current one in a given respect but the information should be available also as to whether two other replicas match in that respect as well.

The following fields specify the file system's class numbers for the equivalence relations used in determining the nature of file system transitions. See [Section 3.6](#) for details about how this information is to be used.

- o The field with byte-index LIBX_CLSHARE defines the sharing class for the file system.
- o The field with byte-index LIBX_CLSERVER defines the server class for the file system.
- o The field with byte-index LIBX_CLENDPOINT defines the endpoint class for the file system.
- o The field with byte-index LIBX_CLHANDLE defines the handle class for the file system.
- o The field with byte-index LIBX_CLFILEID defines the fileid class for the file system.

- o The field with byte-index LIBX_CLVERIFIER defines the verifier class for the file system.
- o The field with byte-index LIBX_CLSTATE defines the state class for the file system.

Server-specified preference information is also provided via 8-bit values within the info array. The values provide a rank and an order (see below) to be used with separate values specifiable for the cases of read-only and writable file systems. These values are compared for different file systems to establish the server-specified preference, with lower values indicating "more preferred".

Rank is used to express a strict server-imposed ordering on clients, with lower values indicating "more preferred." Clients should attempt to use all replicas with a given rank before they use one

with a higher rank. Only if all of those file systems are unavailable should the client proceed to those of a higher rank.

Within a rank, the order value is used to specify the server's preference to guide the client's selection when the client's own preferences are not controlling, with lower values of order indicating "more preferred." If replicas are approximately equal in all respects, clients should defer to the order specified by the server. When clients look at server latency as part of their selection, they are free to use this criterion but it is suggested that when latency differences are not significant, the server-specified order should guide selection.

- o The field at byte index LIBX_READRANK gives the rank value to be used for read-only access.
- o The field at byte index LIBX_READORDER gives the order value to be used for read-only access.
- o The field at byte index LIBX_WRITERANK gives the rank value to be used for writable access.
- o The field at byte index LIBX_WRITEORDER gives the order value to be used for writable access.

Depending on the potential need for write access by a given client, one of the pairs of rank and order values is used. The read rank and order should only be used if the client knows that only reading will ever be done or if it is prepared to switch to a different replica in the event that any write access capability is required in the future.

The `locations4_info` structure, encoding the `fs_locations_info` attribute contains the following:

- o The `fs_root` field which contains the pathname of the root of the current filesystem on the current server, just as it does the `fs_locations4` structure.
- o An array of `locations4_item` structures, which contain information about replicas of the current filesystem. Where the current filesystem is actually present, or has been present, i.e. this is not a referral situation, one of the `locations4_item` structure will contain a `locations4_server` for the current server. This structure will have `LIGF_ABSENT` set if the current filesystem is absent, i.e. normal access to it will return `NFS4ERR_MOVED`.
- o The `valid_for` field specifies a time for which it is reasonable for a client to use the `fs_locations_info` attribute without

refetch. The `valid_for` value does not provide a guarantee of validity since servers can unexpectedly go out of service or become inaccessible for any number of reasons. Clients are well-advised to refetch this information for actively accessed filesystem at every `valid_for` seconds. This is particularly important when filesystem replicas may go out of service in a controlled way using the `LIGF_GOING` flag to communicate an ongoing change. The server should set `valid_for` to a value which allows well-behaved clients to notice the `LIF_GOING` flag and make an orderly switch before the loss of service becomes effective. If this value is zero, then no refetch interval is appropriate and the client need not refetch this data on any particular schedule. In the event of a transition to a new filesystem instance, a new value of the `fs_locations_info` attribute will be fetched at the destination and it is to be expected that this may have a different `valid_for` value, which the client should then use, in the same fashion as the previous value.

As noted above, the `fs_locations_info` attribute, when supported, may be requested of absent filesystems without causing `NFS4ERR_MOVED` to be returned and it is generally expected that will be available for both present and absent filesystems even if only a single `location_server` entry is present, designating the current (present) filesystem, or two `location_server` entries designating the current (and now previous) location of an absent filesystem and its successor location. Servers are strongly urged to support this attribute on all filesystems if they support it on any filesystem.

[3.11](#). The Attribute `fs_status`

In an environment in which multiple copies of the same basic set of data are available, information regarding the particular source of such data and the relationships among different copies, can be very helpful in providing consistent data to applications.

```
enum status4_type {
    STATUS4_FIXED = 1,
    STATUS4_UPDATED = 2,
    STATUS4_INTERLOCKED = 3,
    STATUS4_WRITABLE = 4,
    STATUS4_ABSENT = 5
};

struct fs4_status {
    status4_type    type;
    utf8str_cs     source;
```



```

        utf8str_cs    current;
        int32_t       age;
        nfstime4      version;
};

```

The type value indicates the kind of filesystem image represented. This is of particular importance when using the version values to determine appropriate succession of filesystem images. Five types are distinguished:

- o STATUS4_FIXED which indicates a read-only image in the sense that it will never change. The possibility is allowed that as a result of migration or switch to a different image, changed data can be accessed but within the confines of this instance, no change is allowed. The client can use this fact to aggressively cache.
- o STATUS4_UPDATED which indicates an image that cannot be updated by the user writing to it but may be changed exogenously, typically because it is a periodically updated copy of another writable filesystem somewhere else.
- o STATUS4_VERSIONED which indicates that the image, like the STATUS4_UPDATED case, is updated exogenously, but it provides a guarantee that the server will carefully update the associated version value so that the client, may if it chooses, protect itself from a situation in which it reads data from one version of the filesystem, and then later reads data from an earlier version of the same filesystem. See below for a discussion of how this can be done.
- o STATUS4_WRITABLE which indicates that the filesystem is an actual writable one. The client need not of course actually write to the filesystem, but once it does, it should not accept a transition to anything other than a writable instance of that same filesystem.

- o STATUS4_ABSENT which indicates that the information is the last valid for a filesystem which is no longer present.

The opaque strings source and current provide a way of presenting

information about the source of the filesystem image being present. It is not intended that client do anything with this information other than make it available to administrative tools. It is intended that this information be helpful when researching possible problems with a filesystem image that might arise when it is unclear if the correct image is being accessed and if not, how that image came to be made. This kind of debugging information will be helpful, if, as seems likely, copies of filesystems are made in many different ways (e.g. simple user-level copies, filesystem-level point-in-time copies, cloning of the underlying storage), under a variety of administrative arrangements. In such environments, determining how a given set of data was constructed can be very helpful in resolving problems.

The opaque string 'source' is used to indicate the source of a given filesystem with the expectation that tools capable of creating a filesystem image propagate this information, when that is possible. It is understood that this may not always be possible since a user-level copy may be thought of as creating a new data set and the tools used may have no mechanism to propagate this data. When a filesystem is initially created associating with it data regarding how the filesystem was created, where it was created, by whom, etc. can be put in this attribute in a human-readable string form so that it will be available when propagated to subsequent copies of this data.

The opaque string 'current' should provide whatever information is available about the source of the current copy. Such information as the tool creating it, any relevant parameters to that tool, the time at which the copy was done, the user making the change, the server on which the change was made etc. All information should be in a human-readable string form.

The age provides an indication of how out-of-date the file system currently is with respect to its ultimate data source (in case of cascading data updates). This complements the currency field of `locations4_server` (See [Section 3.10](#)) in the following way: the information in `locations4_server.currency` gives a bound for how out of date the data in a file system might typically get, while the age gives a bound on how out of date that data actually is. Negative values imply no information is available. A zero means that this data is known to be current. A positive value means that this data is known to be no older than that number of seconds with respect to the ultimate data source.

The version field provides a version identification, in the form of a time value, such that successive versions always have later time values. When the filesystem type is anything other than STATUS4_VERSIONED, the server may provide such a value but there is no guarantee as to its validity and clients will not use it except to provide additional information to add to 'source' and 'current'.

When the type is STATUS4_VERSIONED, servers should provide a value of version which progresses monotonically whenever any new version of the data is established. This allows the client, if reliable image progression is important to it, to fetch this attribute as part of each COMPOUND where data or metadata from the filesystem is used.

When it is important to the client to make sure that only valid successor images are accepted, it must make sure that it does not read data or metadata from the filesystem without updating its sense of the current state of the image, to avoid the possibility that the fs_status which the client holds will be one for an earlier image, and so accept a new filesystem instance which is later than that but still earlier than updated data read by the client.

In order to do this reliably, it must do a GETATTR of fs_status that follows any interrogation of data or metadata within the filesystem in question. Often this is most conveniently done by appending such a GETATTR after all other operations that reference a given filesystem. When errors occur between reading filesystem data and performing such a GETATTR, care must be exercised to make sure that the data in question is not used before obtaining the proper fs_status value. In this connection, when an OPEN is done within such a versioned filesystem and the associated GETATTR of fs_status is not successfully completed, the open file in question must not be accessed until that fs_status is fetched.

The procedure above will ensure that before using any data from the filesystem the client has in hand a newly-fetched current version of the filesystem image. Multiple values for multiple requests in flight can be resolved by assembling them into the required partial order (and the elements should form a total order within it) and using the last. The client may then, when switching among filesystem instances, decline to use an instance which is not of type STATUS4_VERSIONED or whose version field is earlier than the last one obtained from the predecessor filesystem instance.

[4.](#) Other Changes

This is a list of changes in other areas of the spec that need to be made to conform with what is written here.

- o Need to add `fs_absent`, `fs_locations_info`, and `fs_status` to the list of recommended attributes.
- o Need to add `NFS4ERR_MOVED` to all the ops that don't currently include it to match what the spec says. Alternatively, we may want to factor this out and create a list of errors that any op can receive.
- o Change the definition of `NFS4ERR_MOVED` in [section 20](#) to indicate that it just means that the fs is not there and may never have really "moved".
- o Delete sections [6.14](#) and [6.14.*](#) which have been incorporated in the new chapter.
- o In the spirit of the "minior" issue, fix instances of ampersand-lt which need to come out as less-than. Also in that spirit, fix errors marked "TDB" in the error list.
- o Add `locations4_server`, `locations4_item`, and `locations4_info` as the appropriate sections [1.2.*](#).
- o Add `status4_type` and `fs4_status` as the appropriate sections [1.2.*](#).
- o Delete the errors `NFS4ERR_MOVED_DATA_AND_STATE` and `NFS4ERR_MOVED_DATA` from [section 20](#).
- o Replace the sixth paragraph of [section 2.2.3](#) with the following text: `FH4_VOL_MIGRATION` The filehandle will expire as a result of a file system transition (migration or replication), in those case in which the continuity of filehandle use is not specified by `_handle_ class` information within the `fs_locations_info` attribute. When this bit is set, clients without access to `fs_locations_info` information should assume file handles will expire on file system transitions.

- o Note that the last sentence of the paragraph referred to above has been removed and was never true. It is one thing to say that a file handle may expire (i.e. that you have to be prepared for the server to tell you it is expired) and another to say that you must decide it is expired even if the server may not necessarily recognize as expired (because he has no idea what your handles look like).

- o Replace the tenth paragraph of [section 2.2.3](#) with the following text: Servers which provide volatile filehandles that may expire while open require special care as regards handling of RENAMEs and REMOVEs. This situation can arise if FH4_VOL_MIGRATION or FH4_VOL_RENAME is set, if FH4_VOLATILE_ANY is set and FH4_NOEXPIRE_WITH_OPEN not set, or if a non-readonly file system has a transition target in a different `_handle _` class. In these cases, the server should deny a RENAME or REMOVE that would affect an OPEN file of any of the components leading to the OPEN file. In addition, the server should deny all RENAME or REMOVE requests during the grace period, in order to make sure that reclaims of files where filehandles may have expired do not do a reclaim for the wrong file.

[5.](#) References

- [1] 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.
- [2] Shepler, S., "NFSv4 Minor Version 1", [draft-ietf-nfsv4-minorversion1-02](#) (work in progress), March 2006.

Noveck

Expires December 21, 2006

[Page 47]

Internet-Draft

Location Chapters for 4.1

June 2006

Author's Address

David Noveck
Network Appliance
1601 Trapelo Road, Suite 16
Waltham, MA 02454
US

Phone: +1 781 961 9291
Email: dnoveck@netapp.com

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any

copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright Statement

Copyright (C) The Internet Society (2006). This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.