

NFSv4
Internet-Draft
Intended status: Standards Track
Expires: July 25, 2016

B. Halevy

T. Haynes
Primary Data
January 22, 2016

**Parallel NFS (pNFS) Flexible File Layout
draft-ietf-nfsv4-flex-files-07.txt**

Abstract

The Parallel Network File System (pNFS) allows a separation between the metadata (onto a metadata server) and data (onto a storage device) for a file. The Flexible File Layout Type is defined in this document as an extension to pNFS to allow the use of storage devices in a fashion such that they require only a quite limited degree of interaction with the metadata server, using already existing protocols. Client side mirroring is also added to provide replication of files.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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

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

This Internet-Draft will expire on July 25, 2016.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
1.1.	Definitions	3
1.2.	Difference Between a Data Server and a Storage Device	5
1.3.	Requirements Language	6
2.	Coupling of Storage Devices	6
2.1.	LAYOUTCOMMIT	6
2.2.	Fencing Clients from the Data Server	6
2.2.1.	Implementation Notes for Synthetic uids/gids	7
2.2.2.	Example of using Synthetic uids/gids	7
2.3.	State and Locking Models	8
3.	XDR Description of the Flexible File Layout Type	9
3.1.	Code Components Licensing Notice	10
4.	Device Addressing and Discovery	11
4.1.	ff_device_addr4	11
4.2.	Storage Device Multipathing	13
5.	Flexible File Layout Type	14
5.1.	ff_layout4	14
5.1.1.	Error codes from LAYOUTGET	17
5.1.2.	Client Interactions with FF_FLAGS_NO_IO_THRU_MDS	18
5.2.	Interactions Between Devices and Layouts	18
5.3.	Handling Version Errors	18
6.	Striping via Sparse Mapping	19
7.	Recovering from Client I/O Errors	19
8.	Mirroring	20
8.1.	Selecting a Mirror	21
8.2.	Writing to Mirrors	21
8.3.	Metadata Server Resilvering of the File	22
9.	Flexible Files Layout Type Return	22
9.1.	I/O Error Reporting	23
9.1.1.	ff_ioerr4	23
9.2.	Layout Usage Statistics	24
9.2.1.	ff_io_latency4	24
9.2.2.	ff_layoutupdate4	25
9.2.3.	ff_iostats4	25
9.3.	ff_layoutreturn4	26
10.	Flexible Files Layout Type LAYOUTERROR	27
11.	Flexible Files Layout Type LAYOUTSTATS	27
12.	Flexible File Layout Type Creation Hint	27
12.1.	ff_layouthint4	28
13.	Recalling a Layout	28
13.1.	CB_RECALL_ANY	28

14.	Client Fencing	29
15.	Security Considerations	30
15.1.	Kerberized File Access	30
15.1.1.	Loosely Coupled	31
15.1.2.	Tightly Coupled	31
16.	IANA Considerations	31
17.	References	31
17.1.	Normative References	31
17.2.	Informative References	32
Appendix A.	Acknowledgments	32
Appendix B.	RFC Editor Notes	33
	Authors' Addresses	33

1. Introduction

In the parallel Network File System (pNFS), the metadata server returns Layout Type structures that describe where file data is located. There are different Layout Types for different storage systems and methods of arranging data on storage devices. This document defines the Flexible File Layout Type used with file-based data servers that are accessed using the Network File System (NFS) protocols: NFSv3 [[RFC1813](#)], NFSv4.0 [[RFCNFSv4](#)], NFSv4.1 [[RFC5661](#)], and NFSv4.2 [[NFSv42](#)].

To provide a global state model equivalent to that of the Files Layout Type, a back-end control protocol MAY be implemented between the metadata server and NFSv4.1+ storage devices. It is out of scope for this document to specify the wire protocol of such a protocol, yet the requirements for the protocol are specified in [[RFC5661](#)] and clarified in [[pNFSLayouts](#)].

1.1. Definitions

control protocol: is a set of requirements for the communication of information on layouts, stateids, file metadata, and file data between the metadata server and the storage devices (see [[pNFSLayouts](#)]).

client-side mirroring: is when the client and not the server is responsible for updating all of the mirrored copies of a layout segment.

data file: is that part of the file system object which describes the payload and not the object. E.g., it is the file contents.

data server (DS): is one of the pNFS servers which provides the contents of a file system object which is a regular file. Depending on the layout, there might be one or more data servers

over which the data is striped. Note that while the metadata server is strictly accessed over the NFSv4.1+ protocol, depending on the Layout Type, the data server could be accessed via any protocol that meets the pNFS requirements.

fencing: is when the metadata server prevents the storage devices from processing I/O from a specific client to a specific file.

File Layout Type: is a Layout Type in which the storage devices are accessed via the NFS protocol.

layout: informs a client of which storage devices it needs to communicate with (and over which protocol) to perform I/O on a file. The layout might also provide some hints about how the storage is physically organized.

layout iomode: describes whether the layout granted to the client is for read or read/write I/O.

layout segment: describes a sub-division of a layout. That sub-division might be by the iomode (see Sections [3.3.20](#) and [12.2.9](#) of [\[RFC5661\]](#)), a striping pattern (see [Section 13.3 of \[RFC5661\]](#)), or requested byte range.

layout stateid: is a 128-bit quantity returned by a server that uniquely defines the layout state provided by the server for a specific layout that describes a Layout Type and file (see [Section 12.5.2 of \[RFC5661\]](#)). Further, [Section 12.5.3](#) describes the difference between a layout stateid and a normal stateid.

layout type: describes both the storage protocol used to access the data and the aggregation scheme used to lay out the file data on the underlying storage devices.

loose coupling: is when the metadata server and the storage devices do not have a control protocol present.

metadata file: is that part of the file system object which describes the object and not the payload. E.g., it could be the time since last modification, access, etc.

metadata server (MDS): is the pNFS server which provides metadata information for a file system object. It also is responsible for generating layouts for file system objects. Note that the MDS is responsible for directory-based operations.

mirror: is a copy of a layout segment. While mirroring can be used for backing up a layout segment, the copies can be distributed

such that each remote site has a locally available copy. Note that if one copy of the mirror is updated, then all copies must be updated.

recalling a layout: is when the metadata server uses a back channel to inform the client that the layout is to be returned in a graceful manner. Note that the client could be able to flush any writes, etc., before replying to the metadata server.

revoking a layout: is when the metadata server invalidates the layout such that neither the metadata server nor any storage device will accept any access from the client with that layout.

resilvering: is the act of rebuilding a mirrored copy of a layout segment from a known good copy of the layout segment. Note that this can also be done to create a new mirrored copy of the layout segment.

rsize: is the data transfer buffer size used for reads.

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

storage device: is another term used almost interchangeably with data server. See [Section 1.2](#) for the nuances between the two.

tight coupling: is when the metadata server and the storage devices do have a control protocol present.

wsiz: is the data transfer buffer size used for writes.

[1.2.](#) Difference Between a Data Server and a Storage Device

We defined a data server as a pNFS server, which implies that it can utilize the NFSv4.1+ protocol to communicate with the client. As such, only the File Layout Type would currently meet this requirement. The more generic concept is a storage device, which can use any protocol to communicate with the client. The requirements for a storage device to act together with the metadata server to provide data to a client are that there is a Layout Type specification for the given protocol and that the metadata server has granted a layout to the client. Note that nothing precludes there being multiple supported Layout Types (i.e., protocols) between a metadata server, storage devices, and client.

As storage device is the more encompassing terminology, this document utilizes it over data server.

1.3. Requirements Language

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

2. Coupling of Storage Devices

The coupling of the metadata server with the storage devices can be either tight or loose. In a tight coupling, there is a control protocol present to manage security, LAYOUTCOMMITs, etc. With a loose coupling, the only control protocol might be a version of NFS. As such, semantics for managing security, state, and locking models MUST be defined.

2.1. LAYOUTCOMMIT

With a tightly coupled system, when the metadata server receives a LAYOUTCOMMIT (see [Section 18.42 of \[RFC5661\]](#)), the semantics of the File Layout Type MUST be met (see [Section 12.5.4 of \[RFC5661\]](#)). It is the responsibility of the client to make sure the data file is stable before the metadata server begins to query the storage devices about the changes to the file. With a loosely coupled system, if any WRITE to a storage device did not result with stable_how equal to FILE_SYNC, a LAYOUTCOMMIT to the metadata server MUST be preceded with a COMMIT to the storage device. Note that if the client has not done a COMMIT to the storage device, then the LAYOUTCOMMIT might not be synchronized to the last WRITE operation to the storage device.

2.2. Fencing Clients from the Data Server

With loosely coupled storage devices, the metadata server uses synthetic uids and gids for the data file, where the uid owner of the data file is allowed read/write access and the gid owner is allowed read only access. As part of the layout (see `ffds_user` and `ffds_group` in [Section 5.1](#)), the client is provided with the user and group to be used in the Remote Procedure Call (RPC) [\[RFC5531\]](#) credentials needed to access the data file. Fencing off of clients is achieved by the metadata server changing the synthetic uid and/or gid owners of the data file on the storage device to implicitly revoke the outstanding RPC credentials.

With this loosely coupled model, the metadata server is not able to fence off a single client, it is forced to fence off all clients. However, as the other clients react to the fencing, returning their

layouts and trying to get new ones, the metadata server can hand out a new uid and gid to allow access.

Note: it is recommended to implement common access control methods at the storage device filesystem to allow only the metadata server root (super user) access to the storage device, and to set the owner of all directories holding data files to the root user. This approach provides a practical model to enforce access control and fence off cooperative clients, but it can not protect against malicious clients; hence it provides a level of security equivalent to AUTH_SYS.

With tightly coupled storage devices, the metadata server sets the user and group owners, mode bits, and ACL of the data file to be the same as the metadata file. And the client must authenticate with the storage device and go through the same authorization process it would go through via the metadata server.

2.2.1. Implementation Notes for Synthetic uids/gids

The selection method for the synthetic uids and gids to be used for fencing in loosely coupled storage devices is strictly an implementation issue. I.e., an administrator might restrict a range of such ids available to the Lightweight Directory Access Protocol (LDAP) 'uid' field [[RFC4519](#)]. She might also be able to choose an id that would never be used to grant access. Then when the metadata server had a request to access a file, a SETATTR would be sent to the storage device to set the owner and group of the data file. The user and group might be selected in a round robin fashion from the range of available ids.

Those ids would be sent back as `ffds_user` and `ffds_group` to the client. And it would present them as the RPC credentials to the storage device. When the client was done accessing the file and the metadata server knew that no other client was accessing the file, it could reset the owner and group to restrict access to the data file.

When the metadata server wanted to fence off a client, it would change the synthetic uid and/or gid to the restricted ids. Note that using a restricted id ensures that there is a change of owner and at least one id available that never gets allowed access.

2.2.2. Example of using Synthetic uids/gids

The user `loghyr` creates a file `"ompha.c"` on the metadata server and it creates a corresponding data file on the storage device.

The metadata server entry may look like:


```
-rw-r--r--    1 loghyr  staff    1697 Dec  4 11:31 ompha.c
```

On the storage device, it may be assigned some random synthetic uid/gid to deny access:

```
-rw-r-----    1 19452   28418    1697 Dec  4 11:31 data_ompha.c
```

When the file is opened on a client, since the layout knows nothing about the user (and does not care), whether loghyr or garbo opens the file does not matter. The owner and group are modified and those values are returned.

```
-rw-r-----    1 1066     1067     1697 Dec  4 11:31 data_ompha.c
```

The set of synthetic gids on the storage device should be selected such that there is no mapping in any of the name services used by the storage device. I.e., each group should have no members.

If the layout segment has an iomode of LAYOUTIOMODE4_READ, then the metadata server should return a synthetic uid that is not set on the storage device. Only the synthetic gid would be valid.

The client is thus solely responsible for enforcing file permissions in a loosely coupled model. To allow loghyr write access, it will send an RPC to the storage device with a credential of 1066:1067. To allow garbo read access, it will send an RPC to the storage device with a credential of 1067:1067. The value of the uid does not matter as long as it is not the synthetic uid granted it when getting the layout.

While pushing the enforcement of permission checking onto the client may seem to weaken security, the client may already be responsible for enforcing permissions before modifications are sent to a server. With cached writes, the client is always responsible for tracking who is modifying a file and making sure to not coalesce requests from multiple users into one request.

2.3. State and Locking Models

Metadata file OPEN, LOCK, and DELEGATION operations are always executed only against the metadata server.

The metadata server responds to state changing operations by executing them against the respective data files on the storage devices. It then sends the storage device open stateid as part of the layout (see the `ffm_stateid` in [Section 5.1](#)) and it is then used by the client for executing READ/WRITE operations against the storage device.

NFSv4.1+ storage devices that do not return the EXCHGID4_FLAG_USE_PNFS_DS flag set to EXCHANGE_ID are indicating that they are loosely coupled. As such, they are treated the same way as NFSv4 storage devices.

NFSv4.1+ storage devices that do identify themselves with the EXCHGID4_FLAG_USE_PNFS_DS flag set to EXCHANGE_ID are strongly coupled. They will be using a back-end control protocol as described in [RFC5661] to implement a global stateid model as defined there.

3. XDR Description of the Flexible File Layout Type

This document contains the external data representation (XDR) [RFC4506] description of the Flexible File Layout Type. The XDR description is embedded in this document in a way that makes it simple for the reader to extract into a ready-to-compile form. The reader can feed this document into the following shell script to produce the machine readable XDR description of the Flexible File Layout Type:

<CODE BEGINS>

```
#!/bin/sh
grep '^ *///' $* | sed 's?^ */// ??' | sed 's?^ *///$??'
```

<CODE ENDS>

That is, if the above script is stored in a file called "extract.sh", and this document is in a file called "spec.txt", then the reader can do:

```
sh extract.sh < spec.txt > flex_files_prot.x
```

The effect of the script is to remove leading white space from each line, plus a sentinel sequence of "///".

The embedded XDR file header follows. Subsequent XDR descriptions, with the sentinel sequence are embedded throughout the document.

Note that the XDR code contained in this document depends on types from the NFSv4.1 nfs4_prot.x file [RFC5662]. This includes both nfs types that end with a 4, such as offset4, length4, etc., as well as more generic types such as uint32_t and uint64_t.

3.1. Code Components Licensing Notice

Both the XDR description and the scripts used for extracting the XDR description are Code Components as described in [Section 4](#) of "Legal Provisions Relating to IETF Documents" [[LEGAL](#)]. These Code Components are licensed according to the terms of that document.

<CODE BEGINS>

```
/// /*
///  * Copyright (c) 2012 IETF Trust and the persons identified
///  * as authors of the code. All rights reserved.
///  *
///  * Redistribution and use in source and binary forms, with
///  * or without modification, are permitted provided that the
///  * following conditions are met:
///  *
///  * o Redistributions of source code must retain the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer.
///  *
///  * o Redistributions in binary form must reproduce the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer in the documentation and/or other
///  *   materials provided with the distribution.
///  *
///  * o Neither the name of Internet Society, IETF or IETF
///  *   Trust, nor the names of specific contributors, may be
///  *   used to endorse or promote products derived from this
///  *   software without specific prior written permission.
///  *
///  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
///  * AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
///  * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
///  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
///  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
///  * EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
///  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
///  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
///  * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
///  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
///  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
///  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
///  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
///  * IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
///  * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
///  *
///  * This code was derived from RFCTBD10.
```



```
/// * Please reproduce this note if possible.
/// */
///
/// /*
/// * flex_files_prot.x
/// */
///
/// /*
/// * The following include statements are for example only.
/// * The actual XDR definition files are generated separately
/// * and independently and are likely to have a different name.
/// * %#include <nfsv42.x>
/// * %#include <rpc_prot.x>
/// */
///
```

<CODE ENDS>

4. Device Addressing and Discovery

Data operations to a storage device require the client to know the network address of the storage device. The NFSv4.1+ GETDEVICEINFO operation ([Section 18.40 of \[RFC5661\]](#)) is used by the client to retrieve that information.

4.1. ff_device_addr4

The ff_device_addr4 data structure is returned by the server as the storage protocol specific opaque field da_addr_body in the device_addr4 structure by a successful GETDEVICEINFO operation.

<CODE BEGINS>

```
/// struct ff_device_versions4 {
///     uint32_t      ffdv_version;
///     uint32_t      ffdv_minorversion;
///     uint32_t      ffdv_rsize;
///     uint32_t      ffdv_wsize;
///     bool          ffdv_tightly_coupled;
/// };
///
/// struct ff_device_addr4 {
///     multipath_list4  ffda_netaddrs;
///     ff_device_versions4 ffdv_versions<>;
/// };
///
```


<CODE ENDS>

The `ffda_netaddrs` field is used to locate the storage device. It MUST be set by the server to a list holding one or more of the device network addresses.

The `ffda_versions` array allows the metadata server to present choices as to NFS version, minor version, and coupling strength to the client. The `ffdv_version` and `ffdv_minorversion` represent the NFS protocol to be used to access the storage device. This layout specification defines the semantics for `ffdv_versions` 3 and 4. If `ffdv_version` equals 3 then the server MUST set `ffdv_minorversion` to 0 and `ffdv_tightly_coupled` to false. The client MUST then access the storage device using the NFSv3 protocol [[RFC1813](#)]. If `ffdv_version` equals 4 then the server MUST set `ffdv_minorversion` to one of the NFSv4 minor version numbers and the client MUST access the storage device using NFSv4.

Note that while the client might determine that it cannot use any of the configured combinations of `ffdv_version`, `ffdv_minorversion`, and `ffdv_tightly_coupled`, when it gets the device list from the metadata server, there is no way to indicate to the metadata server as to which device it is version incompatible. If however, the client waits until it retrieves the layout from the metadata server, it can at that time clearly identify the storage device in question (see [Section 5.3](#)).

The `ffdv_rsize` and `ffdv_wsize` are used to communicate the maximum `rsize` and `wsize` supported by the storage device. As the storage device can have a different `rsize` or `wsize` than the metadata server, the `ffdv_rsize` and `ffdv_wsize` allow the metadata server to communicate that information on behalf of the storage device.

`ffdv_tightly_coupled` informs the client as to whether the metadata server is tightly coupled with the storage devices or not. Note that even if the data protocol is at least NFSv4.1, it may still be the case that there is loose coupling in effect. If `ffdv_tightly_coupled` is not set, then the client MUST commit writes to the storage devices for the file before sending a `LAYOUTCOMMIT` to the metadata server. I.e., the writes MUST be committed by the client to stable storage via issuing `WRITES` with `stable_how == FILE_SYNC` or by issuing a `COMMIT` after `WRITES` with `stable_how != FILE_SYNC` (see [Section 3.3.7 of \[RFC1813\]](#)).

4.2. Storage Device Multipathing

The Flexible File Layout Type supports multipathing to multiple storage device addresses. Storage device level multipathing is used for bandwidth scaling via trunking and for higher availability of use in the event of a storage device failure. Multipathing allows the client to switch to another storage device address which may be that of another storage device that is exporting the same data stripe unit, without having to contact the metadata server for a new layout.

To support storage device multipathing, `ffda_netaddrs` contains an array of one or more storage device network addresses. This array (data type `multipath_list4`) represents a list of storage devices (each identified by a network address), with the possibility that some storage device will appear in the list multiple times.

The client is free to use any of the network addresses as a destination to send storage device requests. If some network addresses are less desirable paths to the data than others, then the MDS SHOULD NOT include those network addresses in `ffda_netaddrs`. If less desirable network addresses exist to provide failover, the RECOMMENDED method to offer the addresses is to provide them in a replacement device-ID-to-device-address mapping, or a replacement device ID. When a client finds no response from the storage device using all addresses available in `ffda_netaddrs`, it SHOULD send a `GETDEVICEINFO` to attempt to replace the existing device-ID-to-device-address mappings. If the MDS detects that all network paths represented by `ffda_netaddrs` are unavailable, the MDS SHOULD send a `CB_NOTIFY_DEVICEID` (if the client has indicated it wants device ID notifications for changed device IDs) to change the device-ID-to-device-address mappings to the available addresses. If the device ID itself will be replaced, the MDS SHOULD recall all layouts with the device ID, and thus force the client to get new layouts and device ID mappings via `LAYOUTGET` and `GETDEVICEINFO`.

Generally, if two network addresses appear in `ffda_netaddrs`, they will designate the same storage device. When the storage device is accessed over NFSv4.1 or a higher minor version, the two storage device addresses will support the implementation of client ID or session trunking (the latter is RECOMMENDED) as defined in [\[RFC5661\]](#). The two storage device addresses will share the same server owner or major ID of the server owner. It is not always necessary for the two storage device addresses to designate the same storage device with trunking being used. For example, the data could be read-only, and the data consist of exact replicas.

5. Flexible File Layout Type

The layout4 type is defined in [\[RFC5662\]](#) as follows:

<CODE BEGINS>

```
enum layouttype4 {
    LAYOUT4_NFSV4_1_FILES    = 1,
    LAYOUT4_OSD2_OBJECTS     = 2,
    LAYOUT4_BLOCK_VOLUME     = 3,
    LAYOUT4_FLEX_FILES       = 4
    [[RFC Editor: please modify the LAYOUT4_FLEX_FILES
    to be the layouttype assigned by IANA]]
};

struct layout_content4 {
    layouttype4      loc_type;
    opaque           loc_body<>;
};

struct layout4 {
    offset4          lo_offset;
    length4          lo_length;
    layoutiomode4     lo_iomode;
    layout_content4  lo_content;
};
```

<CODE ENDS>

This document defines structure associated with the layouttype4 value LAYOUT4_FLEX_FILES. [\[RFC5661\]](#) specifies the loc_body structure as an XDR type "opaque". The opaque layout is uninterpreted by the generic pNFS client layers, but is interpreted by the Flexible File Layout Type implementation. This section defines the structure of this otherwise opaque value, ff_layout4.

5.1. ff_layout4

<CODE BEGINS>

```
/// const FF_FLAGS_NO_LAYOUTCOMMIT    = 0x00000001;
/// const FF_FLAGS_NO_IO_THRU_MDS     = 0x00000002;

/// typedef uint32_t                  ff_flags4;
///
```



```
/// struct ff_data_server4 {
///     deviceid4          ffds_deviceid;
///     uint32_t           ffds_efficiency;
///     stateid4           ffds_stateid;
///     nfs_fh4            ffds_fh_vers<>;
///     fattr4_owner       ffds_user;
///     fattr4_owner_group ffds_group;
/// };
///

/// struct ff_mirror4 {
///     ff_data_server4    ffm_data_servers<>;
/// };
///

/// struct ff_layout4 {
///     length4            ffl_stripe_unit;
///     ff_mirror4         ffl_mirrors<>;
///     ff_flags4          ffl_flags;
///     uint32_t           ffl_stats_collect_hint;
/// };
///
```

<CODE ENDS>

The `ff_layout4` structure specifies a layout over a set of mirrored copies of that portion of the data file described in the current layout segment. This mirroring protects against loss of data in layout segments. Note that while not explicitly shown in the above XDR, each `layout4` element returned in the `logr_layout` array of `LAYOUTGET4res` (see [Section 18.43.1 of \[RFC5661\]](#)) describes a layout segment. Hence each `ff_layout4` also describes a layout segment.

It is possible that the file is concatenated from more than one layout segment. Each layout segment MAY represent different striping parameters, applying respectively only to the layout segment byte range.

The `ffl_stripe_unit` field is the stripe unit size in use for the current layout segment. The number of stripes is given inside each mirror by the number of elements in `ffm_data_servers`. If the number of stripes is one, then the value for `ffl_stripe_unit` MUST default to zero. The only supported mapping scheme is sparse and is detailed in [Section 6](#). Note that there is an assumption here that both the stripe unit size and the number of stripes is the same across all mirrors.

The `ffl_mirrors` field is the array of mirrored storage devices which provide the storage for the current stripe, see Figure 1.

The `ffl_stats_collect_hint` field provides a hint to the client on how often the server wants it to report LAYOUTSTATS for a file. The time is in seconds.

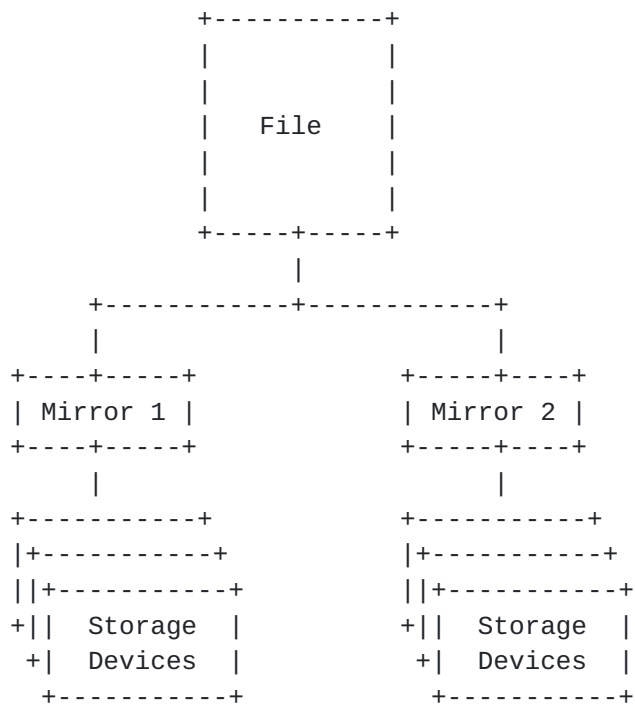


Figure 1

The `ffs_mirrors` field represents an array of state information for each mirrored copy of the current layout segment. Each element is described by a `ff_mirror4` type.

`ffds_deviceid` provides the deviceid of the storage device holding the data file.

`ffds_fh_vers` is an array of filehandles of the data file matching to the available NFS versions on the given storage device. There MUST be exactly as many elements in `ffds_fh_vers` as there are in `ffda_versions`. Each element of the array corresponds to a particular combination of `ffdv_version`, `ffdv_minorversion`, and `ffdv_tightly_coupled` provided for the device. The array allows for server implementations which have different filehandles for different combinations of version, minor version, and coupling strength. See [Section 5.3](#) for how to handle versioning issues between the client and storage devices.

For tight coupling, `ffds_stateid` provides the `stateid` to be used by the client to access the file. For loose coupling and a NFSv4 storage device, the client may use an anonymous `stateid` to perform I/O on the storage device as there is no use for the metadata server `stateid` (no control protocol). In such a scenario, the server MUST set the `ffds_stateid` to be the anonymous `stateid`.

For loosely coupled storage devices, `ffds_user` and `ffds_group` provide the synthetic user and group to be used in the RPC credentials that the client presents to the storage device to access the data files. For tightly coupled storage devices, the user and group on the storage device will be the same as on the metadata server. I.e., if `ffdv_tightly_coupled` (see [Section 4.1](#)) is set, then the client MUST ignore both `ffds_user` and `ffds_group`.

The allowed values for both `ffds_user` and `ffds_group` are specified in [Section 5.9 of \[RFC5661\]](#). For NFSv3 compatibility, user and group strings that consist of decimal numeric values with no leading zeros can be given a special interpretation by clients and servers that choose to provide such support. The receiver may treat such a user or group string as representing the same user as would be represented by an NFSv3 uid or gid having the corresponding numeric value. Note that if using Kerberos for security, the expectation is that these values will be a `name@domain` string.

`ffds_efficiency` describes the metadata server's evaluation as to the effectiveness of each mirror. Note that this is per layout and not per device as the metric may change due to perceived load, availability to the metadata server, etc. Higher values denote higher perceived utility. The way the client can select the best mirror to access is discussed in [Section 8.1](#).

`ffl_flags` is a bitmap that allows the metadata server to inform the client of particular conditions that may result from the more or less tight coupling of the storage devices. `FF_FLAGS_NO_LAYOUTCOMMIT` can be set to indicate that the client is not required to send `LAYOUTCOMMIT` to the metadata server. `FF_FLAGS_NO_IO_THRU_MDS` can be set to indicate that the client SHOULD not send IO operations to the metadata server. I.e., even if a storage device is partitioned from the client, the client SHOULD not try to proxy the IO through the metadata server.

[5.1.1](#). Error codes from LAYOUTGET

[RFC5661] provides little guidance as to how the client is to proceed with a `LAYOUTGET` which returns an error of either `NFS4ERR_LAYOUTTRYLATER`, `NFS4ERR_LAYOUTUNAVAILABLE`, and `NFS4ERR_DELAY`.

NFS4ERR_LAYOUTUNAVAILABLE: there is no layout available and the IO is to go to the metadata server. Note that it is possible to have had a layout before a recall and not after.

NFS4ERR_LAYOUTTRYLATER: there is some issue preventing the layout from being granted. If the client already has an appropriate layout, it SHOULD continue with IO to the storage devices.

NFS4ERR_DELAY: there is some issue preventing the layout from being granted. If the client already has an appropriate layout, it SHOULD not continue with IO to the storage devices.

5.1.2. Client Interactions with FF_FLAGS_NO_IO_THRU_MDS

If the client does not ask for a layout for a file, then the IO will go through the metadata server. Thus, even if the metadata server sets the FF_FLAGS_NO_IO_THRU_MDS flag, it can recall the layout and either not set the flag on the new layout or not provide a layout. When a client encounters an error with a storage device, it typically returns the layout to the metadata server and requests a new layout. The client's IO would then proceed according to the status codes as outlined in [Section 5.1.1](#).

5.2. Interactions Between Devices and Layouts

In [\[RFC5661\]](#), the File Layout Type is defined such that the relationship between multipathing and filehandles can result in either 0, 1, or N filehandles (see [Section 13.3](#)). Some rationals for this are clustered servers which share the same filehandle or allowing for multiple read-only copies of the file on the same storage device. In the Flexible File Layout Type, while there is an array of filehandles, they are independent of the multipathing being used. If the metadata server wants to provide multiple read-only copies of the same file on the same storage device, then it should provide multiple `ff_device_addr4`, each as a mirror. The client can then determine that since the `ffds_fh_vers` are different, then there are multiple copies of the file for the current layout segment available.

5.3. Handling Version Errors

When the metadata server provides the `ffda_versions` array in the `ff_device_addr4` (see [Section 4.1](#)), the client is able to determine if it can not access a storage device with any of the supplied combinations of `ffdv_version`, `ffdv_minorversion`, and `ffdv_tightly_coupled`. However, due to the limitations of reporting errors in `GETDEVICEINFO` (see [Section 18.40 in \[RFC5661\]](#), the client is not able to specify which specific device it can not communicate

with over one of the provided `ffdv_version` and `ffdv_minorversion` combinations. Using `ff_ioerr4` (see [Section 9.1.1](#) inside either the LAYOUTRETURN (see [Section 18.44 of \[RFC5661\]](#)) or the LAYOUTERROR (see [Section 15.6 of \[NFSv42\]](#) and [Section 10](#) of this document), the client can isolate the problematic storage device.

The error code to return for LAYOUTRETURN and/or LAYOUTERROR is `NFS4ERR_MINOR_VERS_MISMATCH`. It does not matter whether the mismatch is a major version (e.g., client can use NFSv3 but not NFSv4) or minor version (e.g., client can use NFSv4.1 but not NFSv4.2), the error indicates that for all the supplied combinations for `ffdv_version` and `ffdv_minorversion`, the client can not communicate with the storage device. The client can retry the GETDEVICEINFO to see if the metadata server can provide a different combination or it can fall back to doing the I/O through the metadata server.

6. Striping via Sparse Mapping

While other Layout Types support both dense and sparse mapping of logical offsets to physical offsets within a file (see for example [Section 13.4 of \[RFC5661\]](#)), the Flexible File Layout Type only supports a sparse mapping.

With sparse mappings, the logical offset within a file (L) is also the physical offset on the storage device. As detailed in [Section 13.4.4 of \[RFC5661\]](#), this results in holes across each storage device which does not contain the current stripe index.

L : logical offset into the file

W : stripe width

W = number of elements in `ffm_data_servers`

S : number of bytes in a stripe

$S = W * \text{ffl_stripe_unit}$

N : stripe number

$N = L / S$

7. Recovering from Client I/O Errors

The pNFS client may encounter errors when directly accessing the storage devices. However, it is the responsibility of the metadata server to recover from the I/O errors. When the LAYOUT4_FLEX_FILES layout type is used, the client MUST report the I/O errors to the server at LAYOUTRETURN time using the `ff_ioerr4` structure (see [Section 9.1.1](#)).

The metadata server analyzes the error and determines the required recovery operations such as recovering media failures or reconstructing missing data files.

The metadata server SHOULD recall any outstanding layouts to allow it exclusive write access to the stripes being recovered and to prevent other clients from hitting the same error condition. In these cases, the server MUST complete recovery before handing out any new layouts to the affected byte ranges.

Although it MAY be acceptable for the client to propagate a corresponding error to the application that initiated the I/O operation and drop any unwritten data, the client SHOULD attempt to retry the original I/O operation by requesting a new layout using LAYOUTGET and retry the I/O operation(s) using the new layout, or the client MAY just retry the I/O operation(s) using regular NFS READ or WRITE operations via the metadata server. The client SHOULD attempt to retrieve a new layout and retry the I/O operation using the storage device first and only if the error persists, retry the I/O operation via the metadata server.

8. Mirroring

The Flexible File Layout Type has a simple model in place for the mirroring of the file data constrained by a layout segment. There is no assumption that each copy of the mirror is stored identically on the storage devices, i.e., one device might employ compression or deduplication on the data. However, the over the wire transfer of the file contents MUST appear identical. Note, this is a construct of the selected XDR representation that each mirrored copy of the layout segment has the same striping pattern (see Figure 1).

The metadata server is responsible for determining the number of mirrored copies and the location of each mirror. While the client may provide a hint to how many copies it wants (see [Section 12](#)), the metadata server can ignore that hint and in any event, the client has no means to dictate neither the storage device (which also means the coupling and/or protocol levels to access the layout segments) nor the location of said storage device.

The updating of mirrored layout segments is done via client-side mirroring. With this approach, the client is responsible for making sure modifications get to all copies of the layout segments it is informed of via the layout. If a layout segment is being resilvered to a storage device, that mirrored copy will not be in the layout. Thus the metadata server MUST update that copy until the client is presented it in a layout. Also, if the client is writing to the layout segments via the metadata server, e.g., using an earlier

version of the protocol, then the metadata server MUST update all copies of the mirror. As seen in [Section 8.3](#), during the resilvering, the layout is recalled, and the client has to make modifications via the metadata server.

[8.1.](#) Selecting a Mirror

When the metadata server grants a layout to a client, it MAY let the client know how fast it expects each mirror to be once the request arrives at the storage devices via the `ffds_efficiency` member. While the algorithms to calculate that value are left to the metadata server implementations, factors that could contribute to that calculation include speed of the storage device, physical memory available to the device, operating system version, current load, etc.

However, what should not be involved in that calculation is a perceived network distance between the client and the storage device. The client is better situated for making that determination based on past interaction with the storage device over the different available network interfaces between the two. I.e., the metadata server might not know about a transient outage between the client and storage device because it has no presence on the given subnet.

As such, it is the client which decides which mirror to access for reading the file. The requirements for writing to a mirrored layout segments are presented below.

[8.2.](#) Writing to Mirrors

The client is responsible for updating all mirrored copies of the layout segments that it is given in the layout. A single failed update is sufficient to fail the entire operation. I.e., if all but one copy is updated successfully and the last one provides an error, then the client needs to return the layout to the metadata server with an error indicating that the update failed to that storage device. If the client is updating the mirrors serially, then it SHOULD stop at the first error encountered and report that to the metadata server. If the client is updating the mirrors in parallel, then it SHOULD wait until all storage devices respond such that it can report all errors encountered during the update.

The metadata server is then responsible for determining if it wants to remove the errant mirror from the layout, if the mirror has recovered from some transient error, etc. When the client tries to get a new layout, the metadata server informs it of the decision by the contents of the layout. The client MUST NOT make any assumptions that the contents of the previous layout will match those of the new

one. If it has updates that were not committed, it MUST resend those updates to all mirrors.

8.3. Metadata Server Resilvering of the File

The metadata server may elect to create a new mirror of the layout segments at any time. This might be to resilver a copy on a storage device which was down for servicing, to provide a copy of the layout segments on storage with different storage performance characteristics, etc. As the client will not be aware of the new mirror and the metadata server will not be aware of updates that the client is making to the layout segments, the metadata server MUST recall the writable layout segment(s) that it is resilvering. If the client issues a LAYOUTGET for a writable layout segment which is in the process of being resilvered, then the metadata server MUST deny that request with a NFS4ERR_LAYOUTTRYLATER. The client can then perform the I/O through the metadata server.

9. Flexible Files Layout Type Return

layoutreturn_file4 is used in the LAYOUTRETURN operation to convey layout-type specific information to the server. It is defined in [\[RFC5661\]](#) as follows:

<CODE BEGINS>

```
struct layoutreturn_file4 {
    offset4      lrf_offset;
    length4      lrf_length;
    stateid4     lrf_stateid;
    /* layouttype4 specific data */
    opaque       lrf_body<>;
};

union layoutreturn4 switch(layoutreturn_type4 lr_returntype) {
    case LAYOUTRETURN4_FILE:
        layoutreturn_file4    lr_layout;
    default:
        void;
};
```



```

struct LAYOUTRETURN4args {
    /* CURRENT_FH: file */
    bool          lora_reclaim;
    layoutreturn_stateid  lora_recallstateid;
    layouttype4    lora_layout_type;
    layoutiomode4   lora_iomode;
    layoutreturn4   lora_layoutreturn;
};

```

<CODE ENDS>

If the `lora_layout_type` layout type is `LAYOUT4_FLEX_FILES`, then the `lrf_body` opaque value is defined by `ff_layoutreturn4` (See [Section 9.3](#)). It allows the client to report I/O error information or layout usage statistics back to the metadata server as defined below.

9.1. I/O Error Reporting

9.1.1. `ff_ioerr4`

<CODE BEGINS>

```

/// struct ff_ioerr4 {
///     offset4      ffie_offset;
///     length4      ffie_length;
///     stateid4     ffie_stateid;
///     device_error4  ffie_errors<>;
/// };
///

```

<CODE ENDS>

Recall that [[NFSv42](#)] defines `device_error4` as:

<CODE BEGINS>

```

struct device_error4 {
    deviceid4      de_deviceid;
    nfsstat4       de_status;
    nfs_opnum4     de_opnum;
};

```

<CODE ENDS>

The `ff_ioerr4` structure is used to return error indications for data files that generated errors during data transfers. These are hints to the metadata server that there are problems with that file. For

each error, `ffie_errors.de_deviceid`, `ffie_offset`, and `ffie_length` represent the storage device and byte range within the file in which the error occurred; `ffie_errors` represents the operation and type of error. The use of `device_error4` is described in Section 15.6 of [\[NFSv42\]](#).

Even though the storage device might be accessed via NFSv3 and reports back NFSv3 errors to the client, the client is responsible for mapping these to appropriate NFSv4 status codes as `de_status`. Likewise, the NFSv3 operations need to be mapped to equivalent NFSv4 operations.

9.2. Layout Usage Statistics

9.2.1. `ff_io_latency4`

<CODE BEGINS>

```
/// struct ff_io_latency4 {
///     uint64_t      ffil_ops_requested;
///     uint64_t      ffil_bytes_requested;
///     uint64_t      ffil_ops_completed;
///     uint64_t      ffil_bytes_completed;
///     uint64_t      ffil_bytes_not_delivered;
///     nfstime4      ffil_total_busy_time;
///     nfstime4      ffil_aggregate_completion_time;
/// };
///
```

<CODE ENDS>

Both operation counts and bytes transferred are kept in the `ff_io_latency4`. READ operations are used for read latencies. Both WRITE and COMMIT operations are used for write latencies.

"Requested" counters track what the client is attempting to do and "completed" counters track what was done. Note that there is no requirement that the client only report completed results that have matching requested results from the reported period.

`ffil_bytes_not_delivered` is used to track the aggregate number of bytes requested by not fulfilled due to error conditions. `ffil_total_busy_time` is the aggregate time spent with outstanding RPC calls, `ffil_aggregate_completion_time` is the sum of all latencies for completed RPC calls.

Note that LAYOUTSTATS are cumulative, i.e., not reset each time the operation is sent. If two LAYOUTSTATS ops for the same file, layout stateid, and originating from the same NFS client are processed at

the same time by the metadata server, then the one containing the larger values contains the most recent time series data.

9.2.2. ff_layoutupdate4

<CODE BEGINS>

```
/// struct ff_layoutupdate4 {
///     netaddr4      ffl_addr;
///     nfs_fh4       ffl_fhandle;
///     ff_io_latency4 ffl_read;
///     ff_io_latency4 ffl_write;
///     nfstime4      ffl_duration;
///     bool          ffl_local;
/// };
///
```

<CODE ENDS>

ffl_addr differentiates which network address the client connected to on the storage device. In the case of multipathing, ffl_fhandle indicates which read-only copy was selected. ffl_read and ffl_write convey the latencies respectively for both read and write operations. ffl_duration is used to indicate the time period over which the statistics were collected. ffl_local if true indicates that the I/O was serviced by the client's cache. This flag allows the client to inform the metadata server about "hot" access to a file it would not normally be allowed to report on.

9.2.3. ff_iostats4

<CODE BEGINS>

```
/// struct ff_iostats4 {
///     offset4      ffis_offset;
///     length4      ffis_length;
///     stateid4     ffis_stateid;
///     io_info4     ffis_read;
///     io_info4     ffis_write;
///     deviceid4    ffis_deviceid;
///     ff_layoutupdate4 ffis_layoutupdate;
/// };
///
```

<CODE ENDS>

Recall that [[NFSv42](#)] defines io_info4 as:

<CODE BEGINS>

```
struct io_info4 {
    uint64_t      ii_count;
    uint64_t      ii_bytes;
};
```

<CODE ENDS>

With pNFS, the data transfers are performed directly between the pNFS client and the storage devices. Therefore, the metadata server has no visibility to the I/O stream and cannot use any statistical information about client I/O to optimize data storage location. `ff_iostats4` MAY be used by the client to report I/O statistics back to the metadata server upon returning the layout. Since it is infeasible for the client to report every I/O that used the layout, the client MAY identify "hot" byte ranges for which to report I/O statistics. The definition and/or configuration mechanism of what is considered "hot" and the size of the reported byte range is out of the scope of this document. It is suggested for client implementation to provide reasonable default values and an optional run-time management interface to control these parameters. For example, a client can define the default byte range resolution to be 1 MB in size and the thresholds for reporting to be 1 MB/second or 10 I/O operations per second. For each byte range, `ffis_offset` and `ffis_length` represent the starting offset of the range and the range length in bytes. `ffis_read.ii_count`, `ffis_read.ii_bytes`, `ffis_write.ii_count`, and `ffis_write.ii_bytes` represent, respectively, the number of contiguous read and write I/Os and the respective aggregate number of bytes transferred within the reported byte range.

The combination of `ffis_deviceid` and `ffl_addr` uniquely identify both the storage path and the network route to it. Finally, the `ffl_fhandle` allows the metadata server to differentiate between multiple read-only copies of the file on the same storage device.

9.3. `ff_layoutreturn4`

<CODE BEGINS>

```
/// struct ff_layoutreturn4 {
///     ff_ioerr4      fflr_ioerr_report<>;
///     ff_iostats4    fflr_iostats_report<>;
/// };
///
```

<CODE ENDS>

When data file I/O operations fail, `fflr_ioerr_report<>` is used to report these errors to the metadata server as an array of elements of type `ff_ioerr4`. Each element in the array represents an error that occurred on the data file identified by `ffie_errors.de_deviceid`. If no errors are to be reported, the size of the `fflr_ioerr_report<>` array is set to zero. The client MAY also use `fflr_iostats_report<>` to report a list of I/O statistics as an array of elements of type `ff_iostats4`. Each element in the array represents statistics for a particular byte range. Byte ranges are not guaranteed to be disjoint and MAY repeat or intersect.

10. Flexible Files Layout Type LAYOUTERROR

If the client is using NFSv4.2 to communicate with the metadata server, then instead of waiting for a `LAYOUTRETURN` to send error information to the metadata server (see [Section 9.1](#)), it MAY use `LAYOUTERROR` (see Section 15.6 of [\[NFSv42\]](#)) to communicate that information. For the Flexible Files Layout Type, this means that `LAYOUTERROR4args` is treated the same as `ff_ioerr4`.

11. Flexible Files Layout Type LAYOUTSTATS

If the client is using NFSv4.2 to communicate with the metadata server, then instead of waiting for a `LAYOUTRETURN` to send I/O statistics to the metadata server (see [Section 9.2](#)), it MAY use `LAYOUTSTATS` (see Section 15.7 of [\[NFSv42\]](#)) to communicate that information. For the Flexible Files Layout Type, this means that `LAYOUTSTATS4args.lsa_layoutupdate` is overloaded with the same contents as in `ffis_layoutupdate`.

12. Flexible File Layout Type Creation Hint

The `layouthint4` type is defined in the [\[RFC5661\]](#) as follows:

<CODE BEGINS>

```
struct layouthint4 {
    layouttype4      loh_type;
    opaque           loh_body<>;
};
```

<CODE ENDS>

The `layouthint4` structure is used by the client to pass a hint about the type of layout it would like created for a particular file. If the `loh_type` layout type is `LAYOUT4_FLEX_FILES`, then the `loh_body` opaque value is defined by the `ff_layouthint4` type.

12.1. ff_layouthint4

<CODE BEGINS>

```
/// union ff_mirrors_hint switch (bool ffmc_valid) {  
///     case TRUE:  
///         uint32_t    ffmc_mirrors;  
///     case FALSE:  
///         void;  
/// };  
///  
  
/// struct ff_layouthint4 {  
///     ff_mirrors_hint    fflh_mirrors_hint;  
/// };  
///
```

<CODE ENDS>

This type conveys hints for the desired data map. All parameters are optional so the client can give values for only the parameter it cares about.

13. Recalling a Layout

While [Section 12.5.5 of \[RFC5661\]](#) discusses layout type independent reasons for recalling a layout, the Flexible File Layout Type metadata server should recall outstanding layouts in the following cases:

- o When the file's security policy changes, i.e., Access Control Lists (ACLs) or permission mode bits are set.
- o When the file's layout changes, rendering outstanding layouts invalid.
- o When there are sharing conflicts.
- o When a file is being resilvered, either due to being repaired after a write error or to load balance.

13.1. CB_RECALL_ANY

The metadata server can use the CB_RECALL_ANY callback operation to notify the client to return some or all of its layouts. The [\[RFC5661\]](#) defines the following types:

<CODE BEGINS>


```
const RCA4_TYPE_MASK_FF_LAYOUT_MIN    = -2;
const RCA4_TYPE_MASK_FF_LAYOUT_MAX    = -1;
[[RFC Editor: please insert assigned constants]]
```

```
struct CB_RECALL_ANY4args {
    uint32_t      craa_layouts_to_keep;
    bitmap4       craa_type_mask;
};
```

<CODE ENDS>

[[AI13: No, 5661 does not define these above values. The ask here is to create these and _add_ them to 5661. --TH]]

Typically, CB_RECALL_ANY will be used to recall client state when the server needs to reclaim resources. The `craa_type_mask` bitmap specifies the type of resources that are recalled and the `craa_layouts_to_keep` value specifies how many of the recalled Flexible File Layouts the client is allowed to keep. The Flexible File Layout Type mask flags are defined as follows:

<CODE BEGINS>

```
/// enum ff_cb_recall_any_mask {
///     FF_RCA4_TYPE_MASK_READ = -2,
///     FF_RCA4_TYPE_MASK_RW   = -1
/// [[RFC Editor: please insert assigned constants]]
/// };
///
```

<CODE ENDS>

They represent the iomode of the recalled layouts. In response, the client SHOULD return layouts of the recalled iomode that it needs the least, keeping at most `craa_layouts_to_keep` Flexible File Layouts.

The `PNFS_FF_RCA4_TYPE_MASK_READ` flag notifies the client to return layouts of iomode `LAYOUTIOMODE4_READ`. Similarly, the `PNFS_FF_RCA4_TYPE_MASK_RW` flag notifies the client to return layouts of iomode `LAYOUTIOMODE4_RW`. When both mask flags are set, the client is notified to return layouts of either iomode.

14. Client Fencing

In cases where clients are uncommunicative and their lease has expired or when clients fail to return recalled layouts within a lease period, at the least the server MAY revoke client layouts and reassign these resources to other clients (see [Section 12.5.5](#) in

[RFC5661])). To avoid data corruption, the metadata server MUST fence off the revoked clients from the respective data files as described in [Section 2.2](#).

15. Security Considerations

The pNFS extension partitions the NFSv4.1+ file system protocol into two parts, the control path and the data path (storage protocol). The control path contains all the new operations described by this extension; all existing NFSv4 security mechanisms and features apply to the control path. The combination of components in a pNFS system is required to preserve the security properties of NFSv4.1+ with respect to an entity accessing data via a client, including security countermeasures to defend against threats that NFSv4.1+ provides defenses for in environments where these threats are considered significant.

The metadata server enforces the file access-control policy at LAYOUTGET time. The client should use suitable authorization credentials for getting the layout for the requested iomode (READ or RW) and the server verifies the permissions and ACL for these credentials, possibly returning NFS4ERR_ACCESS if the client is not allowed the requested iomode. If the LAYOUTGET operation succeeds the client receives, as part of the layout, a set of credentials allowing it I/O access to the specified data files corresponding to the requested iomode. When the client acts on I/O operations on behalf of its local users, it MUST authenticate and authorize the user by issuing respective OPEN and ACCESS calls to the metadata server, similar to having NFSv4 data delegations. If access is allowed, the client uses the corresponding (READ or RW) credentials to perform the I/O operations at the data file's storage devices. When the metadata server receives a request to change a file's permissions or ACL, it SHOULD recall all layouts for that file and it MUST fence off the clients holding outstanding layouts for the respective file by implicitly invalidating the outstanding credentials on all data files comprising before committing to the new permissions and ACL. Doing this will ensure that clients re-authorize their layouts according to the modified permissions and ACL by requesting new layouts. Recalling the layouts in this case is courtesy of the server intended to prevent clients from getting an error on I/Os done after the client was fenced off.

15.1. Kerberized File Access

15.1.1. Loosely Coupled

Under this coupling model, the principal used to authenticate the metadata file is different than that used to authenticate the data file. I.e., the synthetic principals generated to control access to the data file could prove to be difficult to manage.

While RPCSEC_GSS version 3 (RPCSEC_GSSv3) [[rpcsec_gssv3](#)] could be used to authorize the client to the storage device on behalf of the metadata server, such a requirement exceeds the loose coupling model. I.e., each of the metadata server, storage device, and client would have to implement RPCSEC_GSSv3.

In all, while either an elaborate schema could be used to automatically authenticate principals or RPCSEC_GSSv3 aware clients, metadata server, and storage devices could be deployed, if more secure authentication is desired, tight coupling should be considered as described in the next section.

15.1.2. Tightly Coupled

With tight coupling, the principal used to access the metadata file is exactly the same as used to access the data file. Thus there are no security issues related to using Kerberos with a tightly coupled system.

16. IANA Considerations

As described in [[RFC5661](#)], new layout type numbers have been assigned by IANA. This document defines the protocol associated with the existing layout type number, LAYOUT4_FLEX_FILES.

17. References

17.1. Normative References

- [LEGAL] IETF Trust, "Legal Provisions Relating to IETF Documents", November 2008, <<http://trustee.ietf.org/docs/IETF-Trust-License-Policy.pdf>>.
- [NFSv42] Haynes, T., "NFS Version 4 Minor Version 2", [draft-ietf-nfsv4-minorversion2-28](#) (Work In Progress), November 2014.
- [RFC1813] IETF, "NFS Version 3 Protocol Specification", [RFC 1813](#), June 1995.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.

- [RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, [RFC 4506](#), May 2006.
- [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", [RFC 5531](#), May 2009.
- [RFC5661] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 Protocol", [RFC 5661](#), January 2010.
- [RFC5662] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", [RFC 5662](#), January 2010.
- [RFCNFSv4] Haynes, T. and D. Noveck, "NFS Version 4 Protocol", [draft-ietf-nfsv4-rfc3530bis-35](#) (work in progress), Dec 2014.
- [pNFSLayouts] Haynes, T., "Considerations for a New pNFS Layout Type", [draft-ietf-nfsv4-layout-types-02](#) (Work In Progress), October 2014.

[17.2. Informative References](#)

- [RFC4519] Sciberras, A., Ed., "Lightweight Directory Access Protocol (LDAP): Schema for User Applications", [RFC 4519](#), DOI 10.17487/RFC4519, June 2006, <<http://www.rfc-editor.org/info/rfc4519>>.
- [rpcsec_gssv3] Adamson, W. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", November 2014.

[Appendix A. Acknowledgments](#)

Those who provided miscellaneous comments to early drafts of this document include: Matt W. Benjamin, Adam Emerson, J. Bruce Fields, and Lev Solomonov.

Those who provided miscellaneous comments to the final drafts of this document include: Anand Ganesh, Robert Wipfel, Gobikrishnan Sundharraj, and Trond Myklebust.

Idan Kedar caught a nasty bug in the interaction of client side mirroring and the minor versioning of devices.

Dave Noveck provided a comprehensive review of the document during the working group last call.

Olga Kornievskaia lead the charge against the use of a credential versus a principal in the fencing approach. Andy Adamson and Benjamin Kaduk helped to sharpen the focus.

Tigran Mkrtchyan provided the use case for not allowing the client to proxy the IO through the data server.

[Appendix B](#). RFC Editor Notes

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

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

Authors' Addresses

Benny Halevy

Email: bhalevy@gmail.com

Thomas Haynes
Primary Data, Inc.
4300 El Camino Real Ste 100
Los Altos, CA 94022
USA

Phone: +1 408 215 1519

Email: thomas.haynes@primarydata.com

