NFSv4 B. Halevy
Internet-Draft T. Haynes
Intended status: Informational Primary Data
Expires: April 10, 2015 October 07, 2014

Parallel NFS (pNFS) Flexible File Layout
draft-ietf-nfsv4-flex-files-02.txt

Abstract

The Parallel Network File System (pNFS) allows a separation between the metadata and data for a file. The metadata file access is handled via Network File System version 4 (NFSv4) minor version 1 (NFSv4.1) and the data file access is specific to the protocol being used between the client and storage device. The client is informed by the metadata server as to which protocol to use via a Layout Type. The Flexible File Layout Type is defined in this document as an extension to NFSv4.1 to allow the use of storage devices which need not be tightly coupled to the metadata server.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 10, 2015.

Table of Contents

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 [RFC3530], 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]).

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 provide 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 NFSv4.1 protocol. It is defined in Section 13 of [RFC5661].

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 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 lays 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 file. While mirroring can be used for backing up a file, the copies can be distrbuted such that each remote site has a locally cached copy. Note that if one copy of the mirror is updated, then all copies must be updated.

Object Layout Type: is a Layout Type in which the storage devices are accessed via the OSD protocol [ANSI400-2004]. It is defined in [RFC5664].

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 file from a known good copy of the file. Note that this can also be done to create a new mirrored copy of the file.

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.

wsize: 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.

A file is split into metadata and data. The "metadata file" is that part of the file stored on the metadata server. The "data file" is

that part of the file stored on the storage device. And the "file" is the combination of the two.

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]). With a loosely coupled system, a LAYOUTCOMMIT to the metadata server MUST be proceeded with a COMMIT to the storage device. I.e., 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. 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. Security Models

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, the client is provided with the rpc credentials to be used (see ffm_auth in Section 5.1) to access the data file. Fencing off clients is achieved by using SETATTR by the server to change the uid and/or gid owners of the data file to implicitly revoke the outstanding rpc credentials. Note: it is recommended to implement common access control methods at the storage device filesystem exports level 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 security method, when using weak auth flavors such as AUTH_SYS, 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 NFSv3.

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.

2.3. State and Locking Models

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

With NFSv4 storage devices, the metadata server, in response to the state changing operation, executes 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.

Standalone NFSv4.1 storage devices that do not return the EXCHGID4_FLAG_USE_PNFS_DS flag to EXCHANGE_ID are used the same way as NFSv4 storage devices.

NFSv4.1 clustered storage devices that do identify themselves with the EXCHGID4_FLAG_USE_PNFS_DS flag to EXCHANGE_ID use 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:

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

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.

```
/// /*
///  * 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>
///  */
///
```

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_addr

The ff_device_addr 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.

```
/// struct ff_device_addr {
///     multipath_list4 ffda_netaddrs;
///     uint32_t        ffda_version;
///     uint32_t        ffda_minorversion;
///     uint32_t        ffda_rsize;
///     uint32_t        ffda_wsize;
///     bool            ffda_tightly_coupled;
/// };
///
```

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_version and ffda_minorversion represent the NFS protocol to be used to access the storage device. This layout specification defines the semantics for ffda_versions 3 and 4. If ffda_version equals 3 then server MUST set ffda_minorversion to 0 and the client MUST access the storage device using the NFSv3 protocol [RFC1813]. If ffda_version equals 4 then the server MUST set ffda_minorversion

to one of the NFSv4 minor version numbers and the client MUST access the storage device using NFSv4.

The ffda_rsize and ffda_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 ffda_rsize and ffda_wsize allow the metadata server to communicate that information on behalf of the storage device.

ffda_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 no control protocol present. If ffda_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 case 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 device (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 optimal paths to the data than others, then the MDS SHOULD NOT include those network addresses in ffda_netaddrs. If less optimal 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 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:

```
    enum layouttype4 {
        LAYOUT4_NFSV4_1_FILES   = 1,
        LAYOUT4_OSD2_OBJECTS    = 2,
        LAYOUT4_BLOCK_VOLUME    = 3,
        LAYOUT4_FLEX_FILES      = 0x80000004
[[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;
    };
```

[[AI10: Remember, using experimental version number to track changes to the XDR via LAYOUT4_FLEX_FILES! --TH]]

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 obviously must be interpreted by the Flexible File Layout Type implementation. This section defines the structure of this opaque value, ff_layout4.

5.1. ff_layout4

```
/// struct ff_data_server4 {
///     deviceid4               ffds_deviceid;
///     uint32_t                ffds_efficiency;
///     stateid4                ffds_stateid;
///     nfs_fh4                 ffds_fhandle;
///     opaque_auth             ffds_auth;
/// };
///

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

/// struct ff_layout4 {
///     length4                 ffl_stripe_unit;
///     ff_mirror4              ffl_mirrors<>;
/// };
///
```

The ff_layout4 structure specifies a layout over a set of mirrored copies of the data file. This mirroring protects against loss of data files.

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 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.

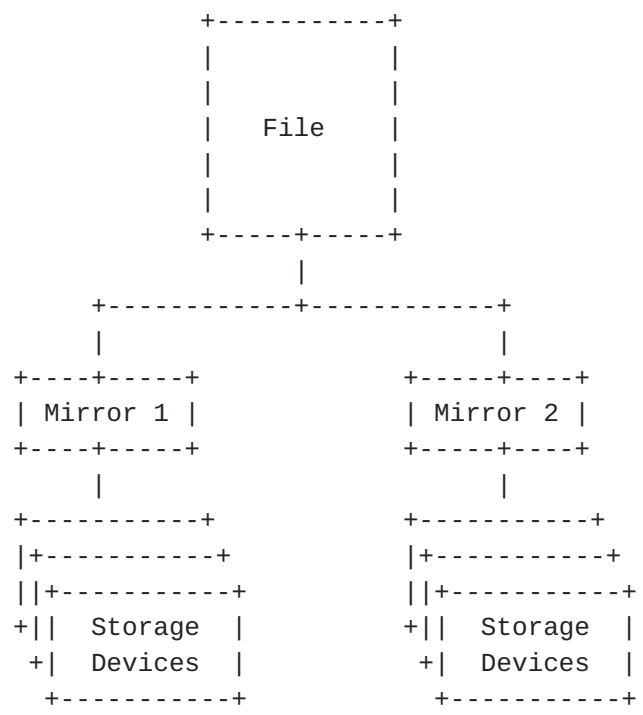

Figure 1

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

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

ffds_fhandle provides the filehandle of the data file on the given storage device. 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 zero.

For NFSv3 storage devices, ffds_auth provides the RPC credentials to be used by the client to access the data files. For NFSv4.x storage devices, the server SHOULD use the AUTH_NONE flavor and a zero length opaque body to minimize the returned structure length. The client MUST ignore ffds_auth in this case. [[AI6: Even for tightly coupled systems, that cannot be correct! --TH]]

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.

6. Striping via Sparse Mapping

While other Layout Types support both dense and sparse mapping of logical offsets to phyisical 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 * 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_ioerr structure (see Section 9.1).

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 files. 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 file. 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 file 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 file) nor the location of said storage device.

8.1. Selecting a Mirror

When the metadata server grants a layout to a client, it can 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 intefaces bewteen 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 file are presented below.

8.2. Writing to Mirrors

The client is responsible for updating all mirrored copies of the file that it is given in the layout. 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.

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 file 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 file 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 file, 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 IO 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:

```
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;
};
```

If the lora_layout_type layout type is LAYOUT4_FLEX_FILES, then the lrf_body opaque value is defined by the ff_layoutreturn4 type. The new type allows the client to report I/O error information or layout usage statistics back to the metadata server as defined below.

9.1. ff_ioerr

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

Recall that [NFSv42] defines device_error4 as:

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

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 16.6 of [NFSv42].

9.2. ff_iostats

```
/// struct ff_iostats4 {
///     offset4         ffis_offset;
///     length4         ffis_length;
///     stateid4        ffis_stateid;
///     uint32_t        ffis_duration;
///     io_info4        ffis_read;
///     io_info4        ffis_write;
///     layoutupdate4   ffis_layoutupdate;
/// };
///
```

Recall that [NFSv42] defines io_info4 as:

```
   struct io_info4 {
           uint32_t        ii_count;
           uint64_t        ii_bytes;
   };
```

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_duration represents the number of seconds the reported burst of I/O lasted. 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. [[AI7: Need to define whether we are using ffis_layoutupdate or not. --TH]] [[AI8: Actually, ffis_duration might be what we plop down in there. In any event, ffis_duration needs some work. --TH]]

9.3. ff_layoutreturn

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

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 can use LAYOUTERROR (see Section 16.6 of [NFSv42]) to communicate that information.

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 can use LAYOUTSTATS (see Section 16.7 of [NFSv42]) to communicate that information.

12. Flexible File Layout Type Creation Hint

The layouthint4 type is defined in the [RFC5661] as follows:

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

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

```
/// 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;
/// };
///
```

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 Layouts

The Flexible File Layout Type metadata server should recall outstanding layouts in the following cases:

- When the file's security policy changes, i.e., Access Control Lists (ACLs) or permission mode bits are set.

- When the file's layout changes, rendering outstanding layouts invalid.

- When there are sharing conflicts.

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:

```
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;
};
```

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:

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

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/or device address mappings and reassign these resources to other clients (see "Recalling a Layout" 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 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 with respect to an entity accessing data via a client, including security countermeasures to defend against threats that NFSv4 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 files 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.

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-minorversion2-22 (Work In Progress), April 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.

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

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

[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.

[RFC5664] Halevy, B., Ed., Welch, B., Ed., and J. Zelenka, Ed., "Object-Based Parallel NFS (pNFS) Operations", RFC 5664, January 2010.

[pNFSLayouts] Haynes, T., "Considerations for a New pNFS Layout Type", draft-haynes-nfsv4-layout-types-02 (Work In Progress), April 2014.

17.2. Informative References

[ANSI400-2004] Weber, R., Ed., "ANSI INCITS 400-2004, Information Technology - SCSI Object-Based Storage Device Commands (OSD)", December 2004.

Appendix A. Acknowledgments

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

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
Primary Data, Inc.

Email: bhalevy@primarydata.com
URI: http://www.primarydata.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