NFSv4 B. Halevy
Internet-Draft T. Haynes
Intended status: Informational Primary Data
Expires: October 19, 2014 April 17, 2014

Parallel NFS (pNFS) Flexible Files Layout
draft-bhalevy-nfsv4-flex-files-02.txt

Abstract

Parallel NFS (pNFS) extends Network File System version 4 (NFSv4) to allow clients to directly access file data on the storage used by the NFSv4 server. This ability to bypass the server for data access can increase both performance and parallelism, but requires additional client functionality for data access, some of which is dependent on the class of storage used, i.e., the Layout Type. The main pNFS operations and data types in NFSv4 Minor version 1 specify a layout-type-independent layer; layout-type-specific information is conveyed using opaque data structures whose internal structure is further defined by the particular layout type specification. This document specifies the NFSv4.1 Flexible Files pNFS Layout as a companion to the main NFSv4 Minor version 1 specification for use of pNFS with Data Servers over NFSv4 or higher minor versions using flexible, per-file striping topology.

Table of Contents

1. Introduction

In pNFS, the file server returns typed layout structures that describe where file data is located. There are different layouts for different storage systems and methods of arranging data on storage devices. This document defines the layout used with file-based data servers that are accessed using the Network File System (NFS) Protocol: NFSv3 [RFC1813], NFSv4 [RFC3530], and NFSv4.1 [RFC5661].

In contrast to the LAYOUT4_NFSV4_1_FILES layout type [RFC5661] that also uses NFSv4.1 to access the data server, the Flexible Files layout defines a model of device metadata and striping patterns that is inspired by the object layout [RFC5664] that provide flexible, per-file striping patterns and simple device information suitable aggregating standalone NFS servers into a centrally managed pNFS cluster.

To provide a global state model equivalent to that of the files layout a back-end control protocol may be implemented between the metadata server (MDS) and NFSv4.1 data servers (DSs). 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. 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. Method of Operation

This section describes the semantics and format of flexible file-based layouts for pNFS. Flexible file-based layouts use the LAYOUT4_FLEX_FILES layout type. The LAYOUT4_FLEX_FILES type defines striping data across multiple NFS Data Servers.

For the purpose of this discussion, we will distinguish between user files served by the metadata server, to be referred to as User Files; vs. user files served by Data Servers, to be referred to as Component Objects.

Component Objects are addressable by their NFS filehandle. Each Component Object may store a whole User File or parts of it, in case the User File is striped across multiple Component Objects. The

striping pattern is provided by pfl_striping_pattern as defined below.

Data Servers may be accessed using different versions of the NFS protocol. It is required that the server MUST use Data Servers of the same NFS version and minor version for striping data within each layout. The NFS version and minor version define the respective security, state, and locking models to be used, as described below.

2.1. Security models

With NFSv3 Data Servers, the Metadata Server uses synthetic uids and gids for the Component Objects, where the uid owner of the Component Objects 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 (XREF pfcf_auth) to access the Object. Fencing off clients is achieved by using SETATTR by the server to change the uid and/or gid owners of the Component Objects to implicitly revoke the outstanding rpc credentials. Note: it is recommended to implement common access control methods at the Data Server filesystem exports level to allow only the Metadata Server root (super user) access to the Data Server, and to set the owner of all directories holding Component Objects 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 NFSv4.x Data Servers, the Metadata Server sets the user and group owners, mode bits, and ACL of the Component Objects to be the same as the User File. And the client must authenticate with the Data Server and go through the same authorization process it would go through via the Metadata Server.

2.2. State and Locking Models

User File OPEN, LOCK, and DELEGATION operations are always executed only against the Metadata Server.

With NFSv4 Data Servers, the Metadata Server, in response to the state changing operation, executes them against the respective Component Objects on the Data Server(s). It then sends the Data Server open stateid as part of the layout (see the pfcf_stateid in Section 5.1) and it is then used by the client for executing READ/WRITE operations against the Data Server.

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

NFSv4.1 Clustered Data Servers 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 Files Layout Protocol

This document contains the external data representation (XDR) [RFC4506] description of the NFSv4.1 flexible files layout protocol. 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 NFSv4.1 objects layout protocol:

```
#!/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 > pnfs_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 draft-bhalevy-nfsv4-flex-files-01.
   [[RFC Editor: please insert RFC number if needed]]
   ///  * Please reproduce this note if possible.
   ///  */
   ///
   /// /*
   ///  * pnfs_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 <nfs4_prot.x>
/// %#include <rpc_prot.x>
///
```

4. Device Addressing and Discovery

Data operations to a data server require the client to know the network address of the data server. The GETDEVICEINFO NFSv4.1 operation is used by the client to retrieve that information.

4.1. pnfs_ff_device_addr

The pnfs_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 [RFC5661].

```
/// struct pnfs_ff_device_addr {
///     multipath_list4         pfda_netaddrs;
///     uint32_t                pfda_version;
///     uint32_t                pfda_minorversion;
///     pathname4               pfda_path;
/// };
///
```

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

The pfda_version and pfda_minorversion represent the NFS protocol to be used to access the data server. This layout specification defines the semantics for pfda_versions 3 and 4. If pfda_version equals 3 then server MUST set pfda_minorversion to 0 and the client MUST access the data server using the NFSv3 protocol [RFC1813]. If pfda_version equals 4 then the server MUST set pfda_minorversion to either 0 or 1 and the client MUST access the data server using NFSv4 [RFC3530] or NFSv4.1 [RFC5661], respectively.

The pfda_path MAY be set by the server to an exported path on the data server for device identification. If provided, the path MUST exist and be accessible to the client. If the path does not exist, the client MUST ignore this device information and any layouts referring to the respective deviceid until valid device information is acquired.

4.2. Data Server Multipathing

The flexible file layout supports multipathing to multiple data server addresses. Data-server-level multipathing is used for bandwidth scaling via trunking and for higher availability of use in the case of a data-server failure. Multipathing allows the client to switch to another data server address which may be that of another data server that is exporting the same data stripe unit, without having to contact the metadata server for a new layout.

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

The client is free to use any of the network addresses as a destination to send data server requests. If some network addresses are less optimal paths to the data than others, then the MDS SHOULD NOT include those network addresses in pfda_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 data server using all addresses available in pfda_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 pfda_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 pfda_netaddrs, they will designate the same data server. When the data server is accessed over NFSv4.1 or higher minor version the two data server addresses will support the implementation of client ID or session trunking (the latter is RECOMMENDED) as defined in [RFC5661]. The two data server addresses will share the same server owner or major ID of the server owner. It is not always necessary for the two data server addresses to designate the same server with trunking being used. For example, the data could be read-only, and the data consist of exact replicas.

5. Flexible Files Layout

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      = 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;
/// };
```

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 files layout driver. This section defines the structure of this opaque value, pnfs_ff_layout4.

5.1. pnfs_ff_layout

```
/// enum pnfs_ff_striping_pattern {
///     PFSP_SPARSE_STRIPING = 1,
///     PFSP_DENSE_STRIPING  = 2,
///     PFSP_RAID_4          = 4,
///     PFSP_RAID_5          = 5,
///     PFSP_RAID_PQ         = 6
/// };
///
/// enum pnfs_ff_comp_type {
///     PNFS_FF_COMP_MISSING = 0,
///     PNFS_FF_COMP_PACKED  = 1,
///     PNFS_FF_COMP_FULL    = 2
/// };
///
/// struct pnfs_ff_comp_full {
///     deviceid4               pfcf_deviceid;
///     nfs_fh4                 pfcf_fhandle;
///     stateid4                pfcf_stateid;
///     opaque_auth             pfcf_auth;
///     uint32_t                pfcf_metric;
/// };
///
/// union pnfs_ff_comp switch (pnfs_ff_comp_type pfc_type) {
///    case PNFS_FF_COMP_MISSING:
///         void;
///
///    case PNFS_FF_COMP_PACKED:
///         deviceid4               pfcp_deviceid;
///
///    case PNFS_FF_COMP_FULL:
///         pnfs_ff_comp_full       pfcp_full;
/// };
///
/// struct pnfs_ff_layout {
///     pnfs_ff_striping_pattern    pfl_striping_pattern;
///     uint32_t                    pfl_num_comps;
///     uint32_t                    pfl_mirror_cnt;
///     length4                     pfl_stripe_unit;
///     nfs_fh4                     pfl_global_fh;
///     uint32_t                    pfl_comps_index;
///     pnfs_ff_comp                pfl_comps<>;
/// };
///
```

The pnfs_ff_layout structure specifies a layout over a set of Component Objects. The layout parameterizes the algorithm that maps the file's contents within the returned byte range, as represented by lo_offset and lo_length, over the Component Objects.

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.

This section provides a brief introduction to the layout parameters. See Section 5.2 for a more detailed description of the different striping schemes and the respective interpretation of the layout parameters for each striping scheme.

In addition to mapping data using simple striping schemes where loss of a single component object results in data loss, the layout parameters support mirroring and more advanced redundancy schemes that protect against loss of component objects. pfl_striping_pattern represents the algorithm to be used for mapping byte offsets in the file address space to corresponding component objects in the returned layout and byte offsets in the component's address space. pfl_striping_pattern also represents methods for storing and retrieving redundant data that can be used to recover from failure or loss of component objects.

pfl_num_comps is the total number of component objects the file is striped over within the returned byte range, not counting mirrored components (See pfl_mirror_cnt below). Note that the server MAY grow the file by adding more components to the stripe while clients hold valid layouts until the file has reached its final stripe width.

pfl_mirror_cnt represents the number of mirrors each component in the stripe has. If there is no mirroring then pfm_mirror_cnt MUST be 0. Otherwise, the number of entries listed in pfl_comps MUST be a multiple of (pfl_mirror_cnt + 1).

pfl_stripe_unit is the number of bytes placed on one component before advancing to the next one in the list of components. When the file is striped over a single component object (pfl_num_comps equals to 1), the stripe unit has no use and the server SHOULD set it to the server default value or to zero; otherwise, pfl_stripe_unit MUST NOT be set to zero.

The pfl_comps field represents an array of component objects. The data placement algorithm that maps file data onto component objects assumes that each component object occurs exactly once in the array of components. Therefore, component objects MUST appear in the pfl_comps array only once. The components array may represent all objects comprising the file, in which case pfl_comps_index is set to zero and the number of entries in the pfl_comps array is equal to pfl_num_comps * (pfl_mirror_cnt + 1). The server MAY return fewer components than pfl_num_comps, provided that the returned byte range

represented by lo_offset and lo_count maps in whole into the set of returned component objects. In this case, pfl_comps_index represents the logical position of the returned components array, pfl_comps, within the full array of components that comprise the file. pfl_comps_index MUST be a multiple of (pfl_mirror_cnt + 1).

Each component object in the pfl_comps array is described by the pnfs_ff_comp type.

When a component object is unavailable pfc_type is set to PNFS_FF_COMP_MISSING and no other information for this component is returned. When a data redundancy scheme is being used, as represented by pfl_striping_pattern, the client MAY use a respective data recovery algorithm to reconstruct data that is logically stored on the missing component using user data and redundant data stored on the available components in the containing stripe.

The server MUST set the same pfc_type for all available components to either PNFS_FF_COMP_PACKED or PNFS_FF_COMP_FULL.

When NFSv4.1 Clustered Data Servers are used, the metadata server implements the global state model where all data servers share the same stateid and filehandle for the file. In such case, the client MUST use the open, delegation, or lock stateid returned by the metadata server for the file for accessing the Data Servers for READ and WRITE; the global filehandle to be used by the client is provided by pfl_global_fh. If the metadata server filehandle for the file is being used by all data servers then pfl_global_fh MAY be set to an empty filehandle.

pfcp_deviceid or pfcf_deviceid provide the deviceid of the data server holding the Component Object.

When standalone data servers are used, either over NFSv4 or NFSv4.1, pfl_global_fh SHOULD be set to an empty filehandle and it MUST be ignored by the client and pfcf_fhandle provides the filehandle of the Data Server file holding the Component Object, and pfcf_stateid provides the stateid to be used by the client to access the file.

For NFSv3 Data Servers, pfcf_auth provides the RPC credentials to be used by the client to access the Component Objects. For NFSv4.x Data Servers, the server SHOULD use the AUTH_NONE flavor and a zero length opaque body to minimize the returned structure length. The client MUST ignore pfxf_auth in this case.

When pfl_mirror_cnt is not zero pfcf_metric indicates the distance to the client the distance of the respective component object, otherwise the server MUST set pfcf_metric to zero. When reading data, the

client the client is advised to read from components with the lowest pfcf_metric. When there are several components with the same pfcf_metric client implementations may implement a load distribution algorithm to evenly distribute the read load across several devices and by so provide larger bandwidth.

5.2. Striping Topologies

This section describes the different data mapping schemes in detail.

pnfs_ff_striping_pattern determines the algorithm and placement of redundant data. This section defines the different redundancy algorithms. Note: The term "RAID" (Redundant Array of Independent Disks) is used in this document to represent an array of Component Objects that store data for an individual User File. The objects are stored on independent Data Servers. User File data is encoded and striped across the array of Component Objects using algorithms developed for block-based RAID systems.

5.2.1. PFSP_SPARSE_STRIPING

The mapping from the logical offset within a file (L) to the Component Object C and object-specific offset O is direct and straight forward as defined by the following equations:

L: logical offset into the file

W: stripe width
W = pfl_num_comps

S: number of bytes in a stripe
S = W * pfl_stripe_unit

N: stripe number
N = L / S

C: component index corresponding to L
C = (L % S) / pfl_stripe_unit

O: The component offset corresponding to L
O = L

Note that this computation does not accommodate the same object appearing in the pfl_comps array multiple times. Therefore the server may not return layouts with the same object appearing multiple times. If needed the server can return multiple layout segments each covering a single instance of the object.

PFSP_SPARSE_STRIPING means there is no parity data, so all bytes in the component objects are data bytes located by the above equations for C and O. If a component object is marked as PNFS_FF_COMP_MISSING, the pNFS client MUST either return an I/O error if this component is attempted to be read or, alternatively, it can retry the READ against the pNFS server.

5.2.2. PFSP_DENSE_STRIPING

The mapping from the logical offset within a file (L) to the component object C and object-specific offset O is defined by the following equations:

L: logical offset into the file

W: stripe width
W = pfl_num_comps

S: number of bytes in a stripe
S = W * pfl_stripe_unit

N: stripe number
N = L / S

C: component index corresponding to L
C = (L % S) / pfl_stripe_unit

O: The component offset corresponding to L
O = (N * pfl_stripe_unit) + (L % pfl_stripe_unit)

Note that this computation does not accommodate the same object appearing in the pfl_comps array multiple times. Therefore the server may not return layouts with the same object appearing multiple times. If needed the server can return multiple layout segments each covering a single instance of the object.

PFSP_DENSE_STRIPING means there is no parity data, so all bytes in the component objects are data bytes located by the above equations for C and O. If a component object is marked as PNFS_FF_COMP_MISSING, the pNFS client MUST either return an I/O error if this component is attempted to be read or, alternatively, it can retry the READ against the pNFS server.

Note that the layout depends on the file size, which the client learns from the generic return parameters of LAYOUTGET, by doing GETATTR commands to the Metadata Server. The client uses the file size to decide if it should fill holes with zeros or return a short read. Striping patterns can cause cases where Component Objects are

shorter than other components because a hole happens to correspond to the last part of the Component Object.

5.2.3. PFSP_RAID_4

PFSP_RAID_4 means that the last component object in the stripe contains parity information computed over the rest of the stripe with an XOR operation. If a Component Object is unavailable, the client can read the rest of the stripe units in the damaged stripe and recompute the missing stripe unit by XORing the other stripe units in the stripe. Or the client can replay the READ against the pNFS server that will presumably perform the reconstructed read on the client's behalf.

When parity is present in the file, then the number of parity devices is taken into account in the above equations when calculating (D), the number of data devices in a stripe, as follows:

P: number of parity devices in each stripe
   P = 1

D: number of data devices in a stripe
   D = W - P

I: parity device index
   I = D

5.2.4. PFSP_RAID_5

PNFS_OBJ_RAID_5 means that the position of the parity data is rotated on each stripe. In the first stripe, the last component holds the parity. In the second stripe, the next-to-last component holds the parity, and so on. In this scheme, all stripe units are rotated so that I/O is evenly spread across objects as the file is read sequentially. The rotated parity layout is illustrated here, with hexadecimal numbers indicating the stripe unit.

```
0 1 2 P
4 5 P 3
8 P 6 7
P 9 a b
```

Note that the math for RAID_5 is similar to RAID_4 only that the device indices for each stripe are rotated backwards. So start with the equations above for RAID_4, then compute the rotation as described below.

P: number of parity devices in each stripe
P = 1

PC: Parity Cycle
PC = W

R: The parity rotation index
(N is as computed in above equations for RAID-4)
R = N % PC

I: parity device index
I = (W + W - (R + 1) * P) % W

Cr: The rotated device index
(C is as computed in the above equations for RAID-4)
Cr = (W + C - (R * P)) % W

Note: W is added above to avoid negative numbers modulo math.

5.2.5. PFSP_RAID_PQ

PFSP_RAID_PQ is a double-parity scheme that uses the Reed-Solomon P+Q encoding scheme [ErrorCorrectingCodes]. In this layout, the last two component objects hold the P and Q data, respectively. P is parity computed with XOR. The Q computation is described in detail in [MathOfRAID-6]. The same polynomial "x^8+x^4+x^3+x^2+1" and Galois field size of 2^8 are used here. Clients may simply choose to read data through the metadata server if two or more components are missing or damaged.

The equations given above for embedded parity can be used to map a file offset to the correct component object by setting the number of parity components (P) to 2 instead of 1 for RAID-5 and computing the Parity Cycle length as the Lowest Common Multiple of pfl_num_comps and P, divided by P, as described below. Note: This algorithm can be used also for RAID-5 where P=1.

P: number of parity devices
P = 2

PC: Parity cycle:
PC = LCM(W, P) / P

Q: The device index holding the Q component
(I is as computed in the above equations for RAID-5)
Qdev = (I + 1) % W

5.2.6. RAID Usage and Implementation Notes

RAID layouts with redundant data in their stripes require additional serialization of updates to ensure correct operation. Otherwise, if two clients simultaneously write to the same logical range of an object, the result could include different data in the same ranges of mirrored tuples, or corrupt parity information. It is the responsibility of the metadata server to enforce serialization requirements such as this. For example, the metadata server may do so by not granting overlapping write layouts within mirrored objects.

Many alternative encoding schemes exist for P >= 2 [ErasureCodingLibraries]. These involve P or Q equations different than those used in PFSP_RAID_PQ. Thus, if one of these schemes is to be used in the future, a distinct value must be added to pnfs_ff_striping_pattern for it. While Reed-Solomon codes are well understood, recently discovered schemes such as Liberation codes are more computationally efficient for small group_widths, and Cauchy Reed-Solomon codes are more computationally efficient for higher values of P.

5.3. Mirroring

The pfl_mirror_cnt is used to replicate a file by replicating its Component Objects. If there is no mirroring, then pfs_mirror_cnt MUST be 0. If pfl_mirror_cnt is greater than zero, then the size of the pfl_comps array MUST be a multiple of (pfl_mirror_cnt + 1). Thus, for a classic mirror on two objects, pfl_mirror_cnt is one. Note that mirroring can be defined over any striping pattern.

Replicas are adjacent in the olo_components array, and the value C produced by the above equations is not a direct index into the pfl_comps array. Instead, the following equations determine the replica component index RCi, where i ranges from 0 to pfl_mirror_cnt.

```
FW = size of pfl_comps array / (pfl_mirror_cnt+1)

C = component index for striping or two-level striping
    as calculated using above equations

i ranges from 0 to pfl_mirror_cnt, inclusive
RCi = C * (pfl_mirror_cnt+1) + i
```

6. Recovering from Client I/O Errors

The pNFS client may encounter errors when directly accessing the Data Servers. 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 pflr_ioerr4 structure (see Section 7.1).

The metadata server analyzes the error and determines the required recovery operations such as repairing any parity inconsistencies, recovering media failures, or reconstructing missing objects.

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 Data Server first and only if the error persists, retry the I/O operation via the metadata server.

7. Flexible Files Layout 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 pnfs_ff_layoutreturn4 type.

The pnfs_ff_layoutreturn4 type allows the client to report I/O error information or layout usage statistics back to the metadata server as defined below.

7.1. pflr_errno

```
/// enum pflr_errno {
///     PNFS_FF_ERR_EIO             = 1,
///     PNFS_FF_ERR_NOT_FOUND       = 2,
///     PNFS_FF_ERR_NO_SPACE        = 3,
///     PNFS_FF_ERR_BAD_STATEID     = 4,
///     PNFS_FF_ERR_NO_ACCESS       = 5,
///     PNFS_FF_ERR_UNREACHABLE     = 6,
///     PNFS_FF_ERR_RESOURCE        = 7
/// };
///
```

pflr_errno4 is used to represent error types when read/write errors are reported to the metadata server. The error codes serve as hints

to the metadata server that may help it in diagnosing the exact reason for the error and in repairing it.

PNFS_FF_ERR_EIO indicates the operation failed because the Data Server experienced a failure trying to access the object. The most common source of these errors is media errors, but other internal errors might cause this as well. In this case, the metadata server should go examine the broken object more closely; hence, it should be used as the default error code.

PNFS_FF_ERR_NOT_FOUND indicates the object ID specifies a Component Object that does not exist on the Data Server.

PNFS_FF_ERR_NO_SPACE indicates the operation failed because the Data Server ran out of free capacity during the operation.

PNFS_FF_ERR_BAD_STATEID indicates the stateid is not valid.

PNFS_FF_ERR_NO_ACCESS indicates the RPC credentials do not allow the requested operation. This may happen when the client is fenced off. The client will need to return the layout and get a new one with fresh credentials.

PNFS_FF_ERR_UNREACHABLE indicates the client did not complete the I/O operation at the Data Server due to a communication failure. Whether or not the I/O operation was executed by the Data Server is undetermined.

PNFS_FF_ERR_RESOURCE indicates the client did not issue the I/O operation due to a local problem on the initiator (i.e., client) side, e.g., when running out of memory. The client MUST guarantee that the Data Server WRITE operation was never sent.

7.2. pnfs_ff_ioerr

```
/// struct pnfs_ff_ioerr {
///     deviceid4           ioe_deviceid;
///     nfs_fh4             ioe_fhandle;
///     offset4             ioe_comp_offset;
///     length4             ioe_comp_length;
///     bool                ioe_iswrite;
///     pnfs_ff_errno       ioe_errno;
/// };
///
```

The pnfs_ff_ioerr4 structure is used to return error indications for Component Objects that generated errors during data transfers. These are hints to the metadata server that there are problems with that

object. For each error, "ioe_deviceid", "ioe_fhandle", "ioe_comp_offset", and "ioe_comp_length" represent the Component Object and byte range within the object in which the error occurred; "ioe_iswrite" is set to "true" if the failed Data Server operation was data modifying, and "ioe_errno" represents the type of error.

Component byte ranges in the optional pnfs_ff_ioerr4 structure are used for recovering the object and MUST be set by the client to cover all failed I/O operations to the component.

7.3. pnfs_ff_iostats

```
/// struct pnfs_ff_iostats {
///     offset4             ios_offset;
///     length4             ios_length;
///     uint32_t            ios_duration;
///     uint32_t            ios_rd_count;
///     uint64_t            ios_rd_bytes;
///     uint32_t            ios_wr_count;
///     uint64_t            ios_wr_bytes;
/// };
///
```

With pNFS, the data transfers are performed directly between the pNFS client and the data servers. 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. pnfs_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, ios_offset and ios_length represent the starting offset of the range and the range length in bytes. ios_duration represents the number of seconds the reported burst of I/O lasted. ios_rd_count, ios_rd_bytes, ios_wr_count, and ios_wr_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.

7.4. pnfs_ff_layoutreturn

```
/// struct pnfs_ff_layoutreturn {
///     pnfs_ff_ioerr              pflr_ioerr_report<>;
///     pnfs_ff_iostats            pflr_iostats_report<>;
/// };
///
```

When object I/O operations failed, "pflr_ioerr_report<>" is used to report these errors to the metadata server as an array of elements of type pnfs_ff_ioerr4. Each element in the array represents an error that occurred on the Component Object identified by <ioe_deviceid, ioe_fhandle>. If no errors are to be reported, the size of the pflr_ioerr_report<> array is set to zero. The client MAY also use "pflr_iostats_report<>" to report a list of I/O statistics as an array of elements of type pnfs_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.

8. Flexible Files Creation Layout 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 pnfs_ff_layouthint type.

8.1. pnfs_ff_layouthint

```
/// union pnfs_ff_max_comps_hint switch (bool pfmx_valid) {
///     case TRUE:
///         uint32_t            omx_max_comps;
///     case FALSE:
///         void;
/// };
///
/// union pnfs_ff_stripe_unit_hint switch (bool pfsu_valid) {
///     case TRUE:
///         length4             osu_stripe_unit;
///     case FALSE:
///         void;
/// };
///
/// union pnfs_ff_mirror_cnt_hint switch (bool pfmc_valid) {
///     case TRUE:
///         uint32_t            omc_mirror_cnt;
///     case FALSE:
///         void;
/// };
///
/// union pnfs_ff_striping_pattern_hint switch (bool pfsp_valid) {
///     case TRUE:
///         pnfs_ff_striping_pattern    pfsp_striping_pattern;
///     case FALSE:
///         void;
/// };
///
/// struct pnfs_ff_layouthint {
///     pnfs_ff_max_comps_hint         pflh_max_comps_hint;
///     pnfs_ff_stripe_unit_hint       pflh_stripe_unit_hint;
///     pnfs_ff_mirror_cnt_hint        pflh_mirror_cnt_hint;
///     pnfs_ff_striping_pattern_hint  pflh_striping_pattern_hint;
/// };
///
```

This type conveys hints for the desired data map. All parameters are optional so the client can give values for only the parameters it cares about, e.g. it can provide a hint for the desired number of mirrored components, regardless of the striping pattern selected for the file. The server should make an attempt to honor the hints, but it can ignore any or all of them at its own discretion and without failing the respective CREATE operation.

9. Recalling Layouts

The Flexible Files 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. For example, the server will issue stripe-aligned layout segments for RAID-5 objects. To prevent corruption of the file's parity, multiple clients must not hold valid write layouts for the same stripes. An outstanding READ/WRITE (RW) layout should be recalled when a conflicting LAYOUTGET is received from a different client for LAYOUTIOMODE4_RW and for a byte range overlapping with the outstanding layout segment.

9.1. 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_objects_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_objects_to_keep value specifies how many of the recalled objects the client is allowed to keep. The Flexible Files layout type mask flags are defined as follows. 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_objects_to_keep object-based layouts.

```
/// enum pnfs_ff_cb_recall_any_mask {
///     PNFS_FF_RCA4_TYPE_MASK_READ = -2,
///     PNFS_FF_RCA4_TYPE_MASK_RW   = -1
[[RFC Editor: please insert assigned constants]]
/// };
///
```

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.

10. 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 objects as described in Section 2.1.

11. 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 objects 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 object 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 Component Objects 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.

12. Striping Topologies Extensibility

New striping topologies that are not specified in this document may be specified using @@@. These must be documented in the IETF by submitting an RFC augmenting this protocol provided that:

- New striping topologies MUST be wire-protocol compatible with the Flexible Files Layout protocol as specified in this document.
- Some members of the data structures specified here may be declared as optional or manadatory-not-to-be-used.
- Upon acceptance by the IETF as a RFC, new striping topology constants MUST be registered as describe in Section 13.

13. 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.

A new IANA registry should be assigned to register new data map striping topologies described by the enumerated type: @@@.

14. Normative References

[ErasureCodingLibraries]
Plank, James S., and Luo, Jianqiang and Schuman, Catherine D. and Xu, Lihao and Wilcox-O'Hearn, Zooko, , "A Performance Evaluation and Examination of Open-source Erasure Coding Libraries for Storage", 2007.

[ErrorCorrectingCodes]
MacWilliams, F. and N. Sloane, "The Theory of Error-Correcting Codes, Part I", 1977.

[LEGAL] IETF Trust, "Legal Provisions Relating to IETF Documents", November 2008, <http://trustee.ietf.org/docs/IETF-Trust-License-Policy.pdf>.

[MathOfRAID-6] Anvin, H., "The Mathematics of RAID-6", May 2009, <http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf>.

[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.

Appendix A. Acknowledgments

The pNFS Objects Layout was authored and revised by Brent Welch, Jim Zelenka, Benny Halevy, and Boaz Harrosh.

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