NFSv4 Working Group
Internet-Draft
Intended status: draft
Expires: April 19, 2013

S. Faibish
P. Tao
EMC Corporation

October 19, 2012

Parallel NFS (pNFS) Lustre Layout Operations
draft-faibish-nfsv4-pnfs-lustre-layout-01

Status of this Memo

Copyright Notice

Abstract

Parallel NFS (pNFS) extends Network File System version 4.1(NFSv4.1) to allow clients to directly access file data on the storage used by the NFSv4.1 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, a.k.a. 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 Lustre pNFS Layout Type as a companion to the main NFSv4 Minor version 1 specification.

Table of Contents

1. Introduction

1.1. pNFS Lustre Layout Protocol

Figure 1 shows the overall architecture of a Parallel NFS (pNFS) Protocol ([8]) system:

```
          +-----------+
          |+-----------+                                 +-----------+
          ||+-----------+                                |           |
          |||           |       NFSv4.1 + pNFS           |           |
          +||  Clients  |<------------------------------>|    MDS    |
           +|           |                                |           |
            +-----------+                                |           |
                 |||                                     +-----------+
                 |||                                           |
                 |||                                           |
                 ||| Storage        +-----------+              |
                 ||| Protocol       |+-----------+             |
                 ||+----------------||+-----------+  Control   |
                 |+-----------------|||           |  Protocol  |
                 +------------------+||  Storage  |------------+
                                     +|  Devices  |
                                      +-----------+
```

Figure 1 pNFS Architecture

In this document, "storage device" is used as a general term for a data server and/or storage server for all pNFS layouts. The MetaData Server (MDS) is the NFSv4.1 server that provides pNFS layouts to clients and handles operations on file metadata (e.g., names, attributes).

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 describes the layouts used with Lustre object storage servers (OSSs) that are accessed according to the Lustre storage protocol ([1]).

1.2. General Definitions

The following definitions provide an appropriate context for the reader.

Lustre module	Description
OST	Object Storage Targets are SCSI LUNs store file data objects
OSS	Object Storage Sever implementing Lustre data protocol and serves data
OSC	Object Storage Client is a client of the Lustre services
MDT	Metadata Target is a SCSI LUN that stores files metadata
MDS	Metadata Severs implementing Lustre metadata server control protocol
MDC	Metadata Clients of Lustre protocol services protocol services
PTLRPC	Portal RPC implements Lustre communications over LNET data requests
LNET	Lustre network direct communication without IP routing

1.3. Lustre Protocol description

Lustre is an object-based file system. It is composed of three components: Metadata servers (MDSs), object storage servers (OSSs), and clients.

Lustre uses block devices (SCSI LUNs) for file data storage (OST) and metadata storages (MDT) and each block device can be managed by only one Lustre server (OSS). The total data capacity of the Lustre filesystem is the sum of all individual OST capacities. Lustre clients access and concurrently use data through the standard POSIX I/O system calls.

A Lustre MDS provides metadata services. One Lustre MDS manages one metadata target (MDT). Each MDT stores file metadata, such as file names, directory structures, and access permissions. An OSS exposes block devices and serves data. Each OSS manages one or more object storage targets (OSTs), and OSTs store file data "objects".

The Lustre protocol specifies several operations on objects, including OPEN, READ, WRITE, GET ATTRIBUTES, SET ATTRIBUTES, CREATE, and DELETE. However, using the Lustre layout the Lustre client only uses the OPEN, READ, WRITE and GET ATTRIBUTES commands. The other commands are only used by the Lustre server.

A Lustre file object's layout information is defined in the extended attribute (EA) of the inode. Essentially, EA describes the mapping between file object identifier and its corresponding OSTs. This information is also known as striping. A Lustre-based layout for pNFS includes object identifiers, capabilities that allow clients to READ or WRITE those objects, and various parameters that control how file data is striped across OSTs.

This document specifies the layout protocol and operations using as data and control protocols the Lustre protocol ([1]).

2. Conventions used in this document

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 RFC-2119 [6].

3. XDR Description of the Lustre-Based Layout Protocol

This document contains the external data representation (XDR [2]) description of the NFSv4.1 objects 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 Lustre 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_lustre_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 ([3]). 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

The XDR description, marked with lines beginning with the sequence "///", as well as scripts for extracting the XDR description are Code Components as described in Section 4 of "Legal Provisions Relating to IETF Documents" ([4]). These Code Components are licensed according to the terms of Section 4 of "Legal Provisions Relating to IETF Documents".

```
/// /*
/// * 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.
   /// *
   /// * Please reproduce this note if possible.
   /// */
   ///
   /// /*
   /// *pnfs_lustre_prot.x
   /// */
   ///
   /// %#include <nfs4_prot.x>
   ///
```

4. Basic Data Type Definitions

The following sections define basic data types and constants used by the Lustre Layout protocol.

4.1. pnfs_lov_magic

Lustre uses two magic numbers to identify different lov_mds_md versions.

```
   /// enum pnfs_lov_magic {
   /// LOV_MAGIC_V1 = 0x0BD10BD0, /* to identify lov_mds_md_v1 */
   /// LOV_MAGIC_V3 = 0x0BD30BD0  /* to identify lov_mds_md_v3 */
```

```
/// };
```

pnfs_lov_magic is used to indicate the Lustre protocol MDS metadata version. The magic number is used to identify the protocol version and to detect the byte order of the request sent by the client.

At this time, the Lustre protocol is using LOV_MAGIC_V1/3 to mark different version of lov_mds_md. If OST pooling is used the server will return LOV_MAGIC_V3. If OST pooling is not configured, the MDS server SHOULD return LOV_MAGIC_V1. So the versioning is used just for feature matching. The latest Lustre protocol V3 matches the relevant data structures, APIs, protocols, and algorithms involved for the Lustre version 1.6 source code base.

Therefore, the Lustre protocol version is explicitly called out in the information returned in the layout. (The format value is 0x0BD10BD0 for version V1 capability.)

4.2. pnfs_los_object_cred4

```
/// enum pnfs_los_cap_key_sec4 {
///   PNFS_OSS_CAP_KEY_SEC_NONE = 0,
///   PNFS_OSS_CAP_KEY_SEC_SSV  = 1
/// };
///
/// struct pnfs_los_object_cred4 {
///   pnfs_los_objid4         ploc_object_id;
///   pnfs_los_cap_key_sec4   ploc_cap_key_sec;
///   opaque                  ploc_capability_key<>;
///   opaque                  ploc_capability<>;
/// };
///
```

Lustre PTLRPC supports GSS authentication. PTLRPC implements Lustre communications over LNET ([1]). So pnfs_los_object_cred4 is put inside pnfs_los_layout4 so that if network requires security, credentials can be passed around.

The pnfs_los_object_cred4 structure is used to identify each component comprising the file. The "ploc_object_id" identifies the component object, the "ploc_capability_key" provide the OSS security credentials needed to access that object. The "ploc_cap_key_sec" value denotes the method used to secure the "ploc_capability_key".

To comply with the Lustre security requirements, the capability key SHOULD be transferred securely to prevent eavesdropping. Therefore, a client SHOULD either issue the LAYOUTGET or GETDEVICEINFO operations via RPCSEC_GSS with the privacy service or previously establish a secret state verifier (SSV) for the sessions via the NFSv4.1 SET_SSV operation. The pnfs_los_cap_key_sec4 type is used to identify the method used by the server to secure the capability key.

- PNFS_OSS_CAP_KEY_SEC_NONE denotes that the "ploc_capability_key" is not encrypted, in which case the client SHOULD issue the LAYOUTGET or GETDEVICEINFO operations with RPCSEC_GSS with the privacy service or the NFSv4.1 transport should be secured by using methods that are external to NFSv4.1 like the use of IPsec ([5]) for transporting the NFSV4.1 protocol.

- PNFS_OSS_CAP_KEY_SEC_SSV denotes that the "ploc_capability_key" contents are encrypted using the SSV GSS context and the capability key as inputs to the GSS_Wrap() function (see GSS-API [7]) with the conf_req_flag set to TRUE. The client MUST use the secret SSV key as part of the client's GSS context to decrypt the capability key using the value of the lc_capability_key field as the input_message to the GSS_unwrap() function. Note that to prevent eavesdropping of the SSV key, the client SHOULD issue SET_SSV via RPCSEC_GSS with the privacy service.

The actual method chosen depends on whether the client established a SSV key with the server and whether it issued the operation with the RPCSEC_GSS privacy method. Naturally, if the client did not establish an SSV key via SET_SSV, the server MUST use the PNFS_OSS_CAP_KEY_SEC_NONE method. Otherwise, if the operation was not issued with the RPCSEC_GSS privacy method, the server SHOULD secure the "ploc_capability_key" with the PNFS_OSS_CAP_KEY_SEC_SSV method. The server MAY use the PNFS_OSS_CAP_KEY_SEC_SSV method also when the operation was issued with the RPCSEC_GSS privacy method.

4.3. Data Stripping Algorithms

Currently, only RAID0 is supported but Lustre defines RAID1 as well.

```
/// const LOV_PATTERN_RAID0 = 0x001
///                         /* stripes are used round-robin */
/// const LOV_PATTERN_RAID1 = 0x002
///                         /* stripes are mirrors of each other */
```

5. Object Storage Server Addressing and Discovery

Data operations to an OSS require the client to know the "address" of each OSS's root object. The OSS exposes block devices and serves data. Correspondingly, OSC is client of the services. Each OSS manages one or more OSTs, and OSTs store file data objects. Because these representations are local, GETDEVICEINFO must return information that can be used by the client to select the correct local representation.

5.1. pnfs_los_targetid_type4

The following enum specifies the manner in which a OST (OSS target) can be specified. The target can be specified by the network access protocol type used.

```
/// enum pnfs_los_targetid_type4 {
///   LOS_TARGET_TCP = 1,
///   LOS_TARGET_IB = 2
/// };
```

Where:

- LOS_TARGET_TCP denotes use of the TCP protocol

- LOS_TARGET_IB denotes use of the IB protocol

5.2. pnfs_los_deviceaddr4

The specification (according to [9]) for an object device address is as follows:

```
/// union pnfs_los_targetid4 switch(pnfs_los_targetid_type4 oti_type) {
///   case LOS_TARGET_TCP:
///        netaddr4 tcp_addr<>;
///
///   case LOS_TARGET_IB:
///        netaddr4 ib_addr<>;
///
///   default:
///         void;
/// };
///
/// struct pnfs_los_deviceaddr4 {
///   pnfs_los_targetid4   lda_targetid;
///   opaque               lda_ossname<>;
/// };
```

5.2.1. OSS Target Identifier

When "lda_targetid" is specified as an LOS_TARGET_TCP, if the TCIP network protocol is used, or as the LOS_TARGET_IB, if Infiniband protocol is used.

When "lda_targetid" is specified the opaque field MUST be formatted as the LOS name.

5.2.2. Device Network Address

The network address is given with the netaddr4 type, which specifies a TCP/IP or IB based endpoint (as specified in NFSv4.1 [3]). When given, the client SHOULD use it to probe for the OSS device at the given network address. The client MAY still use other discovery mechanisms to locate the device using the "lda_targetid". In particular, an external name service (external to data protocol coming from LNET) SHOULD be used when the devices may be attached to the network using multiple connections, and/or multiple storage fabrics (e.g., TCP or IB).

6. Lustre-Based Layout

The layout4 type is defined in the NFSv4.1 ([3]) as follows:

```
enum layouttype4 {
     LAYOUT4_NFSV4_1_FILES= 0x1,
     LAYOUT4_OSD2_OBJECTS = 0x2,
     LAYOUT4_BLOCK_VOLUME = 0x3,
     LAYOUT4_OSS_OBJECTS  = 0x0BD30BD4 /* Tentatively */
};

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_OSS_OBJECTS. The NFSv4.1 ([3]) 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 Lustre storage layout driver. This section defines the structure of this opaque value, pnfs_oss_layout4.

6.1. pnfs_lov_mds_md

These are the key file mapping data structures. pnfs_lov_ost_data is per-stripe data structure. lov_mds_md is per file data structure. The difference between v1 and v3 is that, v3 supports OST pooling.

```
/// struct pnfs_lov_ost_data { /* per-stripe data structure */
///   uint64_t l_object_id;    /* OST object ID */
///   uint64_t l_object_seq;   /* OST object seq number */
///   uint32_t l_ost_gen;
///                            /* generation of this l_ost_idx */
///   uint32_t l_ost_idx;
///                   /* OST index in LOV (lov_tgt_desc->tgts) */
/// };
///
/// struct pnfs_lov_mds_md_v1 { /* LOV EA mds/wire data */
///   uint32_t lmm_pattern;
///                   /* LOV_PATTERN_RAID0, LOV_PATTERN_RAID1 */
///   uint64_t lmm_object_id; /* LOV object ID */
///   uint64_t lmm_object_seq;/* LOV object seq number */
///   uint32_t lmm_stripe_size;  /* size of stripe in bytes */
///   uint16_t lmm_stripe_count;
///                      /* num stripes in use for this object */
///   uint16_t lmm_layout_gen;  /* layout generation number */
///
///   pnfs_lov_ost_data lmm_objects[0]; /* per-stripe data */
/// };
///
/// struct pnfs_lov_mds_md_v3 { /* LOV EA mds/wire data */
///   uint32_t lmm_pattern;
///                /* LOV_PATTERN_RAID0, LOV_PATTERN_RAID1 */
///   uint64_t lmm_object_id; /* LOV object ID */
///   uint64_t lmm_object_seq; /* LOV object seq number */
///   uint32_t lmm_stripe_size;  /* size of stripe in bytes */
///   uint16_t lmm_stripe_count;
///                      /* num stripes in use for this object */
///   uint16_t lmm_layout_gen;  /* layout generation number */
///   char  lmm_pool_name[LOV_MAXPOOLNAME];
///                                /* must be 32bit aligned */
///   pnfs_lov_ost_data lmm_objects[0]; /*per-stripe data*/
/// };
///
/// union pnfs_lov_mds_md switch (pnfs_lov_magic lmm_magic) {
///   case LOV_MAGIC_V1:
///        pnfs_lov_mds_md_v1mds_md;
///   case LOV_MAGIC_V3:
///        pnfs_lov_mds_md_v3mds_md;
///   default:
///        void;
/// };
///
```

The pnfs_lov_ost_data structure parameterizes the algorithm that maps a file's contents over the component OST's.

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. The file length in this case MUST be limited to the number of bytes in a full stripe.

The "pnfs_lov_ost_data" is a per stripe data structure that defines the location of the stripe in OST and which OST holds the data.

"l_object_id" holds the file data's object ID on the OST. "l_object_seq" holds the object sequence number which is always 0. "l_ost_idx" holds the OST's index in LOV, and "l_ost_gen" holds the OST's index generation.

The "lmm_magic" specifies the format of the returned stripping information. LOV_MAGIC_V1 is used for pnfs_lov_mds_md_v1, and LOV_MAGIC_V3 is used for "pnfs_lov_mds_md_v3".

The "lmm_pattern" holds the file's stripping pattern. It can be either LOV_PATTERN_RAID0 or LOV_PATTERN_RAID1. "lmm_object_id" holds the MDS object ID. "lmm_object_seq" holds the LOV object sequence number.

"lmm_stripe_size" holds the stripe size in bytes. A file is striped across multiple OSTs in the same stripe size. The "lmm_stripe_count" holds the number of OSTs over which the file is striped.

"llm_layout_gen" holds the generation of current layout information. Clients need to obtain layout generation before IO and check layout generation after IO. If layout generation is changed, client needs to redo the operations.

The "lmm_objects" is an array of "lmm_stripe_count" members containing per OST file information. Each element is in form of struct pnfs_lov_ost_data.

6.2. pnfs_los_layout4

The following is the opaque data in generic layout.

```
/// struct pnfs_los_layout4 {
///  pnfs_lov_magic  lmm_magic;
///  pnfs_lov_mds_md lov_mds_md;
///  uint32_t llo_comps_index;
///  pnfs_los_object_cred4llo_components<>;
/// };
///
```

pnfs_lov_magic and lov_mds_md are defined as above [section 6.1].

The "llo_components" is an array of "pnfs_los_object_cred4", containing credentials that client need to use to connect to OSS's. The "llo_components" may present all credentials that the client needs to access each object of the file, in which case "llo_comps_index" is set to zero. Otherwise if a file has multiple layout segments and different stripping patterns, "llo_comps_index" is set to the index of the object composing the file, and "llo_components" MUST have exactly one entry.

Note that the layout depends on the file size, which the client learns, by doing GETATTR commands to the pNFS metadata server.

A hole, no matter how big it is, can be represented with a single layout extent. The pnfs client uses the file size to decide if it should fill holes with zeros or return a short read of the hole extent. 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.

6.3. Data Mapping Schemes

This section describes the different data mapping schemes in detail. The Lustre layout always uses a "dense" layout as described in NFSv4.1 ([3]). This means that the second stripe unit of the file starts at offset 0 of the second component, rather than at offset stripe_unit bytes. After a full stripe has been written, the next stripe unit is appended to the first component object in the list without any holes in the component objects. From the MDS point of view, each file is composed of multiple data objects striped on one or more OSTs.

6.3.1. Simple Striping

A file object's layout information is defined in the extended attribute (EA) of the inode. Essentially, EA describes the mapping between file object id and its corresponding OSTs.

For example, if file A has a stripe count of three, then its EA will look like:

```
EA ---> <obj id x, ost p>
        <obj id y, ost q>
        <obj id z, ost r>
        stripe size and stripe width
```

In the above equation obj_id is the object identifier of a file fragment on the ost p, "stripe size" is the size of each file segment on one OST and "stripe width" is the number of OST's used. So if the "stripe size" is 1MB, and the "stripe width" is 3, then this would mean that: [0,1M), [4M,5M), ... are stored as object x, which is on OST p; [1M, 2M), [5M, 6M), ... are stored as object y, which is on OST q; [2M,3M), [6M, 7M), ... are stored as object z, which is on OST r.

Before reading the file, client will query the MDS and be informed that it should talk to <ost p, ost q, ost r> for this operation. This information is structured in so-called LSM, and client side LOV (logical object volume) is to interpret this information so client can send requests to OSTs. Here again, the client communicates with OST through a client module interface known as OSC. Depending on the context, OSC can also be used to refer to an OSS client by itself.

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
S = stripe size
C = (L-L%S)%W
O = L/W/S+L%S

In these equations, S is the number of bytes in a full stripe or stripe size. C is an index into the array of components, so it selects a particular OST device. C count starts from zero. O is the offset within the OST that corresponds to the file offset. Note that this computation does accommodate the fact that an object includes all the file segments that are located on same OST.

For example, consider an object striped over three devices, <OST0 OST1 OST2>. The stripe size is 1024KB. The stripe width W is thus 3.

```
Offset 0KB:
  C = (0-0%1)%3 = 0 (OST0)
  O = 0/3/1024 + (0%1024) = 0

Offset 1024KB:
  C = (1024-(1024%1024))%3 = 1 (OST1)
  O = 1024/3/1024 +(1024%1024) = 0

Offset 9000KB:
  C = (9000-(9000%1024))%3 = 2 (OST2)
  O = 9000/3/1024 + (9000%1024) = 810

Offset 102400KB:
  C = (102400-(102400%1024))%3 = 1 (OST0)
  O = 102400/3/1024 + (102400%4096) = 33
```

6.4. RAID Algorithms

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 OST's that store data for an individual file. The objects are stored on independent OST-based storage devices. File data is encoded and striped across the array of component OST's using algorithms developed for block-based RAID systems.

6.4.1. PNFS_OST_RAID_0

PNFS_OST_RAID_0 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_OST_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 MDS server.

6.4.2. PNFS_OST_RAID_1

PNFS_OST_RAID_1 means there is no parity data, but each OST is mirrored to another OST. In this case the component objects are data bytes still located by the above equations for C and O, defined in section 6.3.1. If a component object is marked as PNFS_OST_MISSING, the pNFS client MUST retry the mirrored OST and return an I/O error if the mirror component is missing as well and attempt to be read or, alternatively, it can retry the READ against the pNFS server.

7. Lustre-Based Creation Layout Hint

The layouthint4 type is defined in the NFSv4.1 ([3]) 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 to be created for a particular file. If the "loh_type" layout type is LAYOUT4_OSS_OBJECTS, then the "loh_body" opaque value is defined by the pnfs_oss_layouthint4 type.

7.1. pnfs_los_layouthint4

```
/// union pnfs_lov_stripe_count_hint4 switch (bool lsc_valid) {
///   case TRUE:
///        uint32_t lsc_stripe_count;
///   case FALSE:
///        void;
/// }
///
/// union pnfs_lov_stripe_size_hint4 switch (bool lss_valid) {
///   case TRUE:
///        uint32_t lss_stripe_size;
///   case FALSE:
///        void;
/// }
///
/// union pnfs_lov_stripe_offset_hint4 switch (bool lso_valid) {
///   case TRUE:
///        uint32_t lso_stripe_offset;
///   case FALSE:
///        void;
/// }
///
/// union pnfs_lov_stripe_pattern_hint4 switch (bool lsp_valid) {
///   case TRUE:
///        uint32_t lsp_stripe_pattern;
///   case FALSE:
///        void;
/// }
///
/// union pnfs_lov_pool_hint4 switch (bool lp_valid) {
///   case TRUE:
///        string    lp_pool_name<>;
///   case FALSE:
///        void;
/// }
///
/// struct pnfs_los_layouthint4 {
///   pnfs_lov_stripe_count_hint4   lov_stripe_count_hint;
///   pnfs_lov_stripe_size_hint4    lov_stripe_size_hint;
///   pnfs_lov_stripe_offset_hint4  lov_stripe_offset_hint;
///   pnfs_lov_stripe_pattern_hint4 lov_stripe_pattern_hint;
///   pnfs_lov_pool_hint4           lov_pool_hint;
/// }
///
```

This type conveys hints for the desired data map. Hints are indications of the client for preferences of the data stripe type to be used for the file. 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 RAID algorithm 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.

8. IANA Considerations

As described in NFSv4.1 ([8]), new layout type numbers have been assigned by IANA. This document defines the protocol associated with a new layout type number, LAYOUT4_OSS_OBJECTS, and it requires to be assigned a new value from IANA.

9. References

9.1. Normative References

[1] http://www.scribd.com/doc/59271212/Understanding-Lustre-File-System-Internals

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

[3] 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.

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

[5] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

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

[7] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

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

[9] Eisler, M., "IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats", RFC 5665, January 2010.

This document was prepared using 2-Word-v2.0.template.dot.

Authors' Addresses

Sorin Faibish (editor)
EMC Corporation
228 South Street
Hopkinton, MA 01748
US

Phone: +1 (508) 249-5745
Email: sfaibish@emc.com

Peng Tao
EMC Corporation
8F, Block D, SP Tower
Tsinghua Science Park
Zhongguancun Dong Road
Beijing 100084
PRC

Phone: +86 (10) 8215 8293
Email: tao.peng@emc.com