                  Parallel NFS (pNFS) Block/Volume Layout

Abstract

   Parallel NFS (pNFS) extends Network File Sharing 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. The main pNFS operations document
   specifies storage-class-independent extensions to NFS; this document
   specifies the additional extensions (primarily data structures) for
   use of pNFS with block- and volume-based storage.

Status of This Memo

   This is an Internet Standards Track document.

Table of Contents

1. Introduction

   Figure 1 shows the overall architecture of a Parallel NFS (pNFS)
   system:

```
    +-----------+
    |+-----------+                                +-----------+
    ||+-----------+                               |           |
    |||           |      NFSv4.1 + pNFS           |           |
    +|| Clients   |<----------------------------->|  Server   |
     +|           |                               |           |
      +-----------+                               |           |
          |||                                     +-----------+
          |||                                           |
          |||                                           |
          ||| Storage          +-----------+            |
          ||| Protocol         |+-----------+           |
          ||+----------------||+-----------+  Control   |
          |+----------------|||           | Protocol|
    +-----------------+|| Storage  |-----------+
                      +| Systems  |
                       +-----------+
```
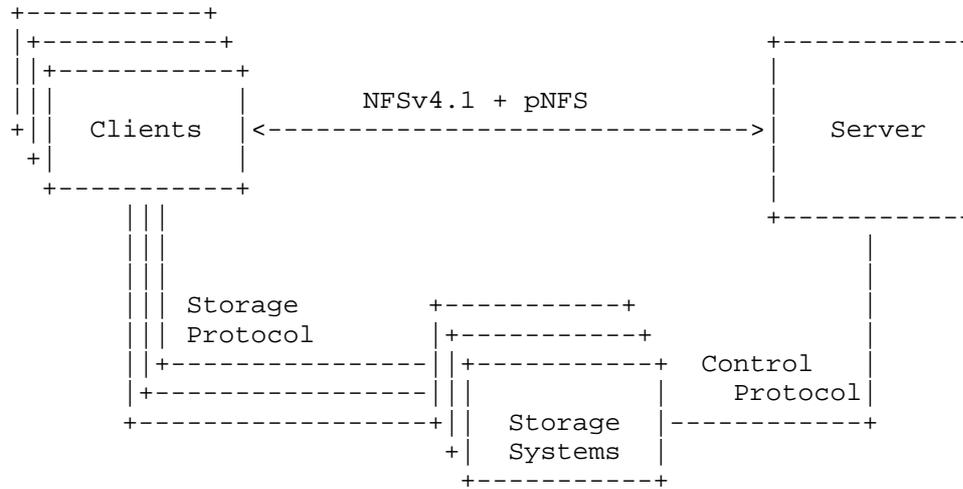
                    Figure 1: pNFS Architecture

   The overall approach is that pNFS-enhanced clients obtain sufficient
   information from the server to enable them to access the underlying
   storage (on the storage systems) directly. See the pNFS portion of
   [NFSv4.1] for more details. This document is concerned with access
   from pNFS clients to storage systems over storage protocols based on
   blocks and volumes, such as the Small Computer System Interface
   (SCSI) protocol family (e.g., parallel SCSI, Fibre Channel Protocol
   (FCP) for Fibre Channel, Internet SCSI (iSCSI), Serial Attached SCSI
   (SAS), and Fibre Channel over Ethernet (FCoE)). This class of
   storage is referred to as block/volume storage. While the Server to
   Storage System protocol, called the "Control Protocol", is not of
   concern for interoperability here, it will typically also be a
   block/volume protocol when clients use block/ volume protocols.

1.1. 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 [RFC2119].

1.2. General Definitions

   The following definitions are provided for the purpose of providing
   an appropriate context for the reader.

   Byte

      This document defines a byte as an octet, i.e., a datum exactly 8
      bits in length.

   Client

      The "client" is the entity that accesses the NFS server's
      resources. The client may be an application that contains the
      logic to access the NFS server directly. The client may also be
      the traditional operating system client that provides remote file
      system services for a set of applications.

   Server

      The "server" is the entity responsible for coordinating client
      access to a set of file systems and is identified by a server
      owner.

1.3. Code Components Licensing Notice

   The external data representation (XDR) description and scripts 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 Section
   4 of "Legal Provisions Relating to IETF Documents".

1.4. XDR Description

   This document contains the XDR ([XDR]) description of the NFSv4.1
   block 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 block layout:

   #!/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 > nfs4_block_layout_spec.x

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

The embedded XDR file header follows, with subsequent pieces embedded
throughout the document:

```
/// /*
///  * This code was derived from RFC 5663.
///  * Please reproduce this note if possible.
///  */
/// /*
///  * Copyright (c) 2010 IETF Trust and the persons identified
///  * as the document authors.  All rights reserved.
///  *
///  * Redistribution and use in source and binary forms, with
///  * or without modification, are permitted provided that the
///  * following conditions are met:
///  *
///  * - Redistributions of source code must retain the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer.
///  *
///  * - 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.
///  *
///  * - 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.
///  */
///
/// /*
///  *         nfs4_block_layout_prot.x
///  */
///
/// %#include "nfsv41.h"
///
```

The XDR code contained in this document depends on types from the
nfsv41.x file. 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.

2. Block Layout Description

2.1. Background and Architecture

The fundamental storage abstraction supported by block/volume storage
is a storage volume consisting of a sequential series of fixed-size
blocks. This can be thought of as a logical disk; it may be realized
by the storage system as a physical disk, a portion of a physical
disk, or something more complex (e.g., concatenation, striping, RAID,
and combinations thereof) involving multiple physical disks or
portions thereof.

A pNFS layout for this block/volume class of storage is responsible
for mapping from an NFS file (or portion of a file) to the blocks of
storage volumes that contain the file. The blocks are expressed as
extents with 64-bit offsets and lengths using the existing NFSv4
offset4 and length4 types. Clients must be able to perform I/O to
the block extents without affecting additional areas of storage
(especially important for writes); therefore, extents MUST be aligned
to 512-byte boundaries, and writable extents MUST be aligned to the
block size used by the NFSv4 server in managing the actual file
system (4 kilobytes and 8 kilobytes are common block sizes). This
block size is available as the NFSv4.1 layout_blksize attribute.
[NFSv4.1]. Readable extents SHOULD be aligned to the block size used
by the NFSv4 server, but in order to support legacy file systems with
fragments, alignment to 512-byte boundaries is acceptable.

The pNFS operation for requesting a layout (LAYOUTGET) includes the
"layoutiomode4 loga_iomode" argument, which indicates whether the
requested layout is for read-only use or read-write use. A read-only
layout may contain holes that are read as zero, whereas a read-write
layout will contain allocated, but un-initialized storage in those
holes (read as zero, can be written by client). This document also
supports client participation in copy-on-write (e.g., for file
systems with snapshots) by providing both read-only and un-
initialized storage for the same range in a layout. Reads are
initially performed on the read-only storage, with writes going to
the un-initialized storage. After the first write that initializes
the un-initialized storage, all reads are performed to that now-
initialized writable storage, and the corresponding read-only storage
is no longer used.

The block/volume layout solution expands the security
responsibilities of the pNFS clients, and there are a number of
environments where the mandatory to implement security properties for
NFS cannot be satisfied. The additional security responsibilities of
the client follow, and a full discussion is present in Section 3,
"Security Considerations".

o Typically, storage area network (SAN) disk arrays and SAN
   protocols provide access control mechanisms (e.g., Logical Unit
   Number (LUN) mapping and/or masking), which operate at the
   granularity of individual hosts, not individual blocks. For this
   reason, block-based protection must be provided by the client
   software.

o Similarly, SAN disk arrays and SAN protocols typically are not
   able to validate NFS locks that apply to file regions. For
   instance, if a file is covered by a mandatory read-only lock, the
   server can ensure that only readable layouts for the file are
   granted to pNFS clients. However, it is up to each pNFS client to
   ensure that the readable layout is used only to service read
   requests, and not to allow writes to the existing parts of the
   file.

Since block/volume storage systems are generally not capable of
enforcing such file-based security, in environments where pNFS
clients cannot be trusted to enforce such policies, pNFS block/volume
storage layouts SHOULD NOT be used.

2.2. GETDEVICELIST and GETDEVICEINFO

2.2.1. Volume Identification

   Storage systems such as storage arrays can have multiple physical
   network ports that need not be connected to a common network,
   resulting in a pNFS client having simultaneous multipath access to
   the same storage volumes via different ports on different networks.

   The networks may not even be the same technology -- for example,
   access to the same volume via both iSCSI and Fibre Channel is
   possible, hence network addresses are difficult to use for volume
   identification. For this reason, this pNFS block layout identifies
   storage volumes by content, for example providing the means to match
   (unique portions of) labels used by volume managers. Volume
   identification is performed by matching one or more opaque byte
   sequences to specific parts of the stored data. Any block pNFS
   system using this layout MUST support a means of content-based unique
   volume identification that can be employed via the data structure
   given here.

   /// struct pnfs_block_sig_component4 { /* disk signature component */
   ///   int64_t bsc_sig_offset;   /* byte offset of component
   ///   on volume*/
   ///   opaque bsc_contents<>;   /* contents of this component
   ///   of the signature */
   /// };
   ///

   Note that the opaque "bsc_contents" field in the
   "pnfs_block_sig_component4" structure MUST NOT be interpreted as a
   zero-terminated string, as it may contain embedded zero-valued bytes.
   There are no restrictions on alignment (e.g., neither bsc_sig_offset
   nor the length are required to be multiples of 4). The
   bsc_sig_offset is a signed quantity, which, when positive, represents
   an byte offset from the start of the volume, and when negative
   represents an byte offset from the end of the volume.

   Negative offsets are permitted in order to simplify the client
   implementation on systems where the device label is found at a fixed
   offset from the end of the volume. If the server uses negative
   offsets to describe the signature, then the client and server MUST
   NOT see different volume sizes. Negative offsets SHOULD NOT be used
   in systems that dynamically resize volumes unless care is taken to
   ensure that the device label is always present at the offset from the
   end of the volume as seen by the clients.

A signature is an array of up to "PNFS_BLOCK_MAX_SIG_COMP" (defined
below) signature components. The client MUST NOT assume that all
signature components are co-located within a single sector on a block
device.

The pNFS client block layout driver uses this volume identification
to map pnfs_block_volume_type4 PNFS_BLOCK_VOLUME_SIMPLE deviceid4s to
its local view of a LUN.

2.2.2. Volume Topology

The pNFS block server volume topology is expressed as an arbitrary
combination of base volume types enumerated in the following data
structures. The individual components of the topology are contained
in an array and components may refer to other components by using
array indices.

```
/// enum pnfs_block_volume_type4 {
///      PNFS_BLOCK_VOLUME_SIMPLE = 0,  /* volume maps to a single
///                                        LU */
///      PNFS_BLOCK_VOLUME_SLICE  = 1,  /* volume is a slice of
///                                        another volume */
///      PNFS_BLOCK_VOLUME_CONCAT = 2,  /* volume is a
///                                        concatenation of
///                                        multiple volumes */
///      PNFS_BLOCK_VOLUME_STRIPE = 3   /* volume is striped across
///                                        multiple volumes */
/// };
///
/// const PNFS_BLOCK_MAX_SIG_COMP = 16;/* maximum components per
///                                        signature */
/// struct pnfs_block_simple_volume_info4 {
///     pnfs_block_sig_component4 bsv_ds<PNFS_BLOCK_MAX_SIG_COMP>;
///                                      /* disk signature */
/// };
///
///
/// struct pnfs_block_slice_volume_info4 {
///     offset4  bsv_start;             /* offset of the start of the
///                                        slice in bytes */
///     length4  bsv_length;            /* length of slice in bytes */
///     uint32_t bsv_volume;            /* array index of sliced
///                                        volume */
/// };
///
/// struct pnfs_block_concat_volume_info4 {
///     uint32_t  bcv_volumes<>;        /* array indices of volumes
///                                        which are concatenated */
```

```
/// };
///
/// struct pnfs_block_stripe_volume_info4 {
///     length4  bsv_stripe_unit;      /* size of stripe in bytes */
///     uint32_t bsv_volumes<>;        /* array indices of volumes
///                                       which are striped across --
///                                       MUST be same size */
/// };
///
/// union pnfs_block_volume4 switch (pnfs_block_volume_type4 type) {
///     case PNFS_BLOCK_VOLUME_SIMPLE:
///         pnfs_block_simple_volume_info4 bv_simple_info;
///     case PNFS_BLOCK_VOLUME_SLICE:
///         pnfs_block_slice_volume_info4 bv_slice_info;
///     case PNFS_BLOCK_VOLUME_CONCAT:
///         pnfs_block_concat_volume_info4 bv_concat_info;
///     case PNFS_BLOCK_VOLUME_STRIPE:
///         pnfs_block_stripe_volume_info4 bv_stripe_info;
/// };
///
/// /* block layout specific type for da_addr_body */
/// struct pnfs_block_deviceaddr4 {
///     pnfs_block_volume4 bda_volumes<>; /* array of volumes */
/// };
///
```

The "pnfs_block_deviceaddr4" data structure is a structure that
allows arbitrarily complex nested volume structures to be encoded.
The types of aggregations that are allowed are stripes,
concatenations, and slices. Note that the volume topology expressed
in the pnfs_block_deviceaddr4 data structure will always resolve to a
set of pnfs_block_volume_type4 PNFS_BLOCK_VOLUME_SIMPLE. The array
of volumes is ordered such that the root of the volume hierarchy is
the last element of the array. Concat, slice, and stripe volumes
MUST refer to volumes defined by lower indexed elements of the array.

The "pnfs_block_device_addr4" data structure is returned by the
server as the storage-protocol-specific opaque field da_addr_body in
the "device_addr4" structure by a successful GETDEVICEINFO operation
[NFSv4.1].

As noted above, all device_addr4 structures eventually resolve to a
set of volumes of type PNFS_BLOCK_VOLUME_SIMPLE. These volumes are
each uniquely identified by a set of signature components.
Complicated volume hierarchies may be composed of dozens of volumes
each with several signature components; thus, the device address may
require several kilobytes. The client SHOULD be prepared to allocate
a large buffer to contain the result. In the case of the server

returning NFS4ERR_TOOSMALL, the client SHOULD allocate a buffer of at
least gdir_mincount_bytes to contain the expected result and retry
the GETDEVICEINFO request.

2.2.3. GETDEVICELIST and GETDEVICEINFO deviceid4

The server in response to a GETDEVICELIST request typically will
return a single "deviceid4" in the gdlr_deviceid_list array. This is
because the deviceid4 when passed to GETDEVICEINFO will return a
"device_addr4", which encodes the entire volume hierarchy. In the
case of copy-on-write file systems, the "gdlr_deviceid_list" array
may contain two deviceid4's, one referencing the read-only volume
hierarchy, and one referencing the writable volume hierarchy. There
is no required ordering of the readable and writable IDs in the array
as the volumes are uniquely identified by their deviceid4, and are
referred to by layouts using the deviceid4. Another example of the
server returning multiple device items occurs when the file handle
represents the root of a namespace spanning multiple physical file
systems on the server, each with a different volume hierarchy. In
this example, a server implementation may return either a list of
device IDs used by each of the physical file systems, or it may
return an empty list.

Each deviceid4 returned by a successful GETDEVICELIST operation is a
shorthand id used to reference the whole volume topology. These
device IDs, as well as device IDs returned in extents of a LAYOUTGET
operation, can be used as input to the GETDEVICEINFO operation.
Decoding the "pnfs_block_deviceaddr4" results in a flat ordering of
data blocks mapped to PNFS_BLOCK_VOLUME_SIMPLE volumes. Combined
with the mapping to a client LUN described in Section 2.2.1 "Volume
Identification", a logical volume offset can be mapped to a block on
a pNFS client LUN [NFSv4.1].

2.3. Data Structures: Extents and Extent Lists

A pNFS block layout is a list of extents within a flat array of data
blocks in a logical volume. The details of the volume topology can
be determined by using the GETDEVICEINFO operation (see discussion of
volume identification, Section 2.2 above). The block layout
describes the individual block extents on the volume that make up the
file. The offsets and length contained in an extent are specified in
units of bytes.

```
///  enum pnfs_block_extent_state4 {
///      PNFS_BLOCK_READ_WRITE_DATA = 0,/* the data located by this
///                                         extent is valid
///                                         for reading and writing. */
///      PNFS_BLOCK_READ_DATA       = 1, /* the data located by this
///                                         extent is valid for reading
///                                         only; it may not be
///                                         written. */
///      PNFS_BLOCK_INVALID_DATA   = 2, /* the location is valid; the
///                                         data is invalid.  It is a
///                                         newly (pre-) allocated
///                                         extent.  There is physical
///                                         space on the volume. */
///      PNFS_BLOCK_NONE_DATA      = 3  /* the location is invalid.
///                                         It is a hole in the file.
///                                         There is no physical space
///                                         on the volume. */
///  };


///
///  struct pnfs_block_extent4 {
///      deviceid4    bex_vol_id;       /* id of logical volume on
///                                         which extent of file is
///                                         stored. */
///      offset4      bex_file_offset;  /* the starting byte offset in
///                                         the file */
///      length4      bex_length;       /* the size in bytes of the
///                                         extent */
///      offset4      bex_storage_offset; /* the starting byte offset
///                                         in the volume */
///      pnfs_block_extent_state4 bex_state;
///                                      /* the state of this extent */
///  };
///
///  /* block layout specific type for loc_body */
///  struct pnfs_block_layout4 {
///      pnfs_block_extent4 blo_extents<>;
///                                      /* extents which make up this
///                                         layout. */
///  };
///
```

   The block layout consists of a list of extents that map the logical
   regions of the file to physical locations on a volume. The
   "bex_storage_offset" field within each extent identifies a location
   on the logical volume specified by the "bex_vol_id" field in the
   extent. The bex_vol_id itself is shorthand for the whole topology of

the logical volume on which the file is stored. The client is
responsible for translating this logical offset into an offset on the
appropriate underlying SAN logical unit. In most cases, all extents
in a layout will reside on the same volume and thus have the same
bex_vol_id. In the case of copy-on-write file systems, the
PNFS_BLOCK_READ_DATA extents may have a different bex_vol_id from the
writable extents.

Each extent maps a logical region of the file onto a portion of the
specified logical volume. The bex_file_offset, bex_length, and
bex_state fields for an extent returned from the server are valid for
all extents. In contrast, the interpretation of the
bex_storage_offset field depends on the value of bex_state as follows
(in increasing order):

o PNFS_BLOCK_READ_WRITE_DATA means that bex_storage_offset is valid,
   and points to valid/initialized data that can be read and written.

o PNFS_BLOCK_READ_DATA means that bex_storage_offset is valid and
   points to valid/ initialized data that can only be read. Write
   operations are prohibited; the client may need to request a
   read-write layout.

o PNFS_BLOCK_INVALID_DATA means that bex_storage_offset is valid,
   but points to invalid un-initialized data. This data must not be
   physically read from the disk until it has been initialized. A
   read request for a PNFS_BLOCK_INVALID_DATA extent must fill the
   user buffer with zeros, unless the extent is covered by a
   PNFS_BLOCK_READ_DATA extent of a copy-on-write file system. Write
   requests must write whole server-sized blocks to the disk; bytes
   not initialized by the user must be set to zero. Any write to
   storage in a PNFS_BLOCK_INVALID_DATA extent changes the written
   portion of the extent to PNFS_BLOCK_READ_WRITE_DATA; the pNFS
   client is responsible for reporting this change via LAYOUTCOMMIT.

o PNFS_BLOCK_NONE_DATA means that bex_storage_offset is not valid,
   and this extent may not be used to satisfy write requests. Read
   requests may be satisfied by zero-filling as for
   PNFS_BLOCK_INVALID_DATA. PNFS_BLOCK_NONE_DATA extents may be
   returned by requests for readable extents; they are never returned
   if the request was for a writable extent.

An extent list contains all relevant extents in increasing order of
the bex_file_offset of each extent; any ties are broken by increasing
order of the extent state (bex_state).

2.3.1. Layout Requests and Extent Lists

   Each request for a layout specifies at least three parameters: file
   offset, desired size, and minimum size. If the status of a request
   indicates success, the extent list returned must meet the following
   criteria:

   o A request for a readable (but not writable) layout returns only
      PNFS_BLOCK_READ_DATA or PNFS_BLOCK_NONE_DATA extents (but not
      PNFS_BLOCK_INVALID_DATA or PNFS_BLOCK_READ_WRITE_DATA extents).

   o A request for a writable layout returns PNFS_BLOCK_READ_WRITE_DATA
      or PNFS_BLOCK_INVALID_DATA extents (but not PNFS_BLOCK_NONE_DATA
      extents). It may also return PNFS_BLOCK_READ_DATA extents only
      when the offset ranges in those extents are also covered by
      PNFS_BLOCK_INVALID_DATA extents to permit writes.

   o The first extent in the list MUST contain the requested starting
      offset.

   o The total size of extents within the requested range MUST cover at
      least the minimum size. One exception is allowed: the total size
      MAY be smaller if only readable extents were requested and EOF is
      encountered.

   o Extents in the extent list MUST be logically contiguous for a
      read-only layout. For a read-write layout, the set of writable
      extents (i.e., excluding PNFS_BLOCK_READ_DATA extents) MUST be
      logically contiguous. Every PNFS_BLOCK_READ_DATA extent in a
      read-write layout MUST be covered by one or more
      PNFS_BLOCK_INVALID_DATA extents. This overlap of
      PNFS_BLOCK_READ_DATA and PNFS_BLOCK_INVALID_DATA extents is the
      only permitted extent overlap.

   o Extents MUST be ordered in the list by starting offset, with
      PNFS_BLOCK_READ_DATA extents preceding PNFS_BLOCK_INVALID_DATA
      extents in the case of equal bex_file_offsets.

   If the minimum requested size, loga_minlength, is zero, this is an
   indication to the metadata server that the client desires any layout
   at offset loga_offset or less that the metadata server has "readily
   available". Readily is subjective, and depends on the layout type
   and the pNFS server implementation. For block layout servers,
   readily available SHOULD be interpreted such that readable layouts
   are always available, even if some extents are in the
   PNFS_BLOCK_NONE_DATA state. When processing requests for writable
   layouts, a layout is readily available if extents can be returned in
   the PNFS_BLOCK_READ_WRITE_DATA state.

2.3.2. Layout Commits

```
///  /* block layout specific type for lou_body */
///  struct pnfs_block_layoutupdate4 {
///       pnfs_block_extent4 blu_commit_list<>;
///                                    /* list of extents which
///                                     * now contain valid data.
///                                     */
///  };
///
```

   The "pnfs_block_layoutupdate4" structure is used by the client as the
   block-protocol specific argument in a LAYOUTCOMMIT operation. The
   "blu_commit_list" field is an extent list covering regions of the
   file layout that were previously in the PNFS_BLOCK_INVALID_DATA
   state, but have been written by the client and should now be
   considered in the PNFS_BLOCK_READ_WRITE_DATA state. The bex_state
   field of each extent in the blu_commit_list MUST be set to
   PNFS_BLOCK_READ_WRITE_DATA. The extents in the commit list MUST be
   disjoint and MUST be sorted by bex_file_offset. The
   bex_storage_offset field is unused. Implementors should be aware
   that a server may be unable to commit regions at a granularity
   smaller than a file-system block (typically 4 KB or 8 KB). As noted
   above, the block-size that the server uses is available as an NFSv4
   attribute, and any extents included in the "blu_commit_list" MUST be
   aligned to this granularity and have a size that is a multiple of
   this granularity. If the client believes that its actions have moved
   the end-of-file into the middle of a block being committed, the
   client MUST write zeroes from the end-of-file to the end of that
   block before committing the block. Failure to do so may result in
   junk (un-initialized data) appearing in that area if the file is
   subsequently extended by moving the end-of-file.

2.3.3. Layout Returns

   The LAYOUTRETURN operation is done without any block layout specific
   data. When the LAYOUTRETURN operation specifies a
   LAYOUTRETURN4_FILE_return type, then the layoutreturn_file4 data
   structure specifies the region of the file layout that is no longer
   needed by the client. The opaque "lrf_body" field of the
   "layoutreturn_file4" data structure MUST have length zero. A
   LAYOUTRETURN operation represents an explicit release of resources by
   the client, usually done for the purpose of avoiding unnecessary
   CB_LAYOUTRECALL operations in the future. The client may return
   disjoint regions of the file by using multiple LAYOUTRETURN
   operations within a single COMPOUND operation.

   Note that the block/volume layout supports unilateral layout
   revocation. When a layout is unilaterally revoked by the server,
   usually due to the client's lease time expiring, or a delegation
   being recalled, or the client failing to return a layout in a timely
   manner, it is important for the sake of correctness that any in-
   flight I/Os that the client issued before the layout was revoked are
   rejected at the storage. For the block/volume protocol, this is
   possible by fencing a client with an expired layout timer from the
   physical storage. Note, however, that the granularity of this
   operation can only be at the host/logical-unit level. Thus, if one
   of a client's layouts is unilaterally revoked by the server, it will
   effectively render useless *all* of the client's layouts for files
   located on the storage units comprising the logical volume. This may
   render useless the client's layouts for files in other file systems.

2.3.4. Client Copy-on-Write Processing

   Copy-on-write is a mechanism used to support file and/or file system
   snapshots. When writing to unaligned regions, or to regions smaller
   than a file system block, the writer must copy the portions of the
   original file data to a new location on disk. This behavior can
   either be implemented on the client or the server. The paragraphs
   below describe how a pNFS block layout client implements access to a
   file that requires copy-on-write semantics.

   Distinguishing the PNFS_BLOCK_READ_WRITE_DATA and
   PNFS_BLOCK_READ_DATA extent types in combination with the allowed
   overlap of PNFS_BLOCK_READ_DATA extents with PNFS_BLOCK_INVALID_DATA
   extents allows copy-on-write processing to be done by pNFS clients.
   In classic NFS, this operation would be done by the server. Since
   pNFS enables clients to do direct block access, it is useful for
   clients to participate in copy-on-write operations. All block/volume
   pNFS clients MUST support this copy-on-write processing.

   When a client wishes to write data covered by a PNFS_BLOCK_READ_DATA
   extent, it MUST have requested a writable layout from the server;
   that layout will contain PNFS_BLOCK_INVALID_DATA extents to cover all
   the data ranges of that layout's PNFS_BLOCK_READ_DATA extents. More
   precisely, for any bex_file_offset range covered by one or more
   PNFS_BLOCK_READ_DATA extents in a writable layout, the server MUST
   include one or more PNFS_BLOCK_INVALID_DATA extents in the layout
   that cover the same bex_file_offset range. When performing a write
   to such an area of a layout, the client MUST effectively copy the
   data from the PNFS_BLOCK_READ_DATA extent for any partial blocks of
   bex_file_offset and range, merge in the changes to be written, and
   write the result to the PNFS_BLOCK_INVALID_DATA extent for the blocks
   for that bex_file_offset and range. That is, if entire blocks of
   data are to be overwritten by an operation, the corresponding

PNFS_BLOCK_READ_DATA blocks need not be fetched, but any partial-
block writes must be merged with data fetched via
PNFS_BLOCK_READ_DATA extents before storing the result via
PNFS_BLOCK_INVALID_DATA extents. For the purposes of this
discussion, "entire blocks" and "partial blocks" refer to the
server's file-system block size. Storing of data in a
PNFS_BLOCK_INVALID_DATA extent converts the written portion of the
PNFS_BLOCK_INVALID_DATA extent to a PNFS_BLOCK_READ_WRITE_DATA
extent; all subsequent reads MUST be performed from this extent; the
corresponding portion of the PNFS_BLOCK_READ_DATA extent MUST NOT be
used after storing data in a PNFS_BLOCK_INVALID_DATA extent. If a
client writes only a portion of an extent, the extent may be split at
block aligned boundaries.

When a client wishes to write data to a PNFS_BLOCK_INVALID_DATA
extent that is not covered by a PNFS_BLOCK_READ_DATA extent, it MUST
treat this write identically to a write to a file not involved with
copy-on-write semantics. Thus, data must be written in at least
block-sized increments, aligned to multiples of block-sized offsets,
and unwritten portions of blocks must be zero filled.

In the LAYOUTCOMMIT operation that normally sends updated layout
information back to the server, for writable data, some
PNFS_BLOCK_INVALID_DATA extents may be committed as
PNFS_BLOCK_READ_WRITE_DATA extents, signifying that the storage at
the corresponding bex_storage_offset values has been stored into and
is now to be considered as valid data to be read.
PNFS_BLOCK_READ_DATA extents are not committed to the server. For
extents that the client receives via LAYOUTGET as
PNFS_BLOCK_INVALID_DATA and returns via LAYOUTCOMMIT as
PNFS_BLOCK_READ_WRITE_DATA, the server will understand that the
PNFS_BLOCK_READ_DATA mapping for that extent is no longer valid or
necessary for that file.

2.3.5. Extents are Permissions

Layout extents returned to pNFS clients grant permission to read or
write; PNFS_BLOCK_READ_DATA and PNFS_BLOCK_NONE_DATA are read-only
(PNFS_BLOCK_NONE_DATA reads as zeroes), PNFS_BLOCK_READ_WRITE_DATA
and PNFS_BLOCK_INVALID_DATA are read/write, (PNFS_BLOCK_INVALID_DATA
reads as zeros, any write converts it to PNFS_BLOCK_READ_WRITE_DATA).
This is the only means a client has of obtaining permission to
perform direct I/O to storage devices; a pNFS client MUST NOT perform
direct I/O operations that are not permitted by an extent held by the
client. Client adherence to this rule places the pNFS server in
control of potentially conflicting storage device operations,
enabling the server to determine what does conflict and how to avoid
conflicts by granting and recalling extents to/from clients.

Block/volume class storage devices are not required to perform read
and write operations atomically. Overlapping concurrent read and
write operations to the same data may cause the read to return a
mixture of before-write and after-write data. Overlapping write
operations can be worse, as the result could be a mixture of data
from the two write operations; data corruption can occur if the
underlying storage is striped and the operations complete in
different orders on different stripes. When there are multiple
clients who wish to access the same data, a pNFS server can avoid
these conflicts by implementing a concurrency control policy of
single writer XOR multiple readers. This policy MUST be implemented
when storage devices do not provide atomicity for concurrent
read/write and write/write operations to the same data.

If a client makes a layout request that conflicts with an existing
layout delegation, the request will be rejected with the error
NFS4ERR_LAYOUTTRYLATER. This client is then expected to retry the
request after a short interval. During this interval, the server
SHOULD recall the conflicting portion of the layout delegation from
the client that currently holds it. This reject-and-retry approach
does not prevent client starvation when there is contention for the
layout of a particular file. For this reason, a pNFS server SHOULD
implement a mechanism to prevent starvation. One possibility is that
the server can maintain a queue of rejected layout requests. Each
new layout request can be checked to see if it conflicts with a
previous rejected request, and if so, the newer request can be
rejected. Once the original requesting client retries its request,
its entry in the rejected request queue can be cleared, or the entry
in the rejected request queue can be removed when it reaches a
certain age.

NFSv4 supports mandatory locks and share reservations. These are
mechanisms that clients can use to restrict the set of I/O operations
that are permissible to other clients. Since all I/O operations
ultimately arrive at the NFSv4 server for processing, the server is
in a position to enforce these restrictions. However, with pNFS
layouts, I/Os will be issued from the clients that hold the layouts
directly to the storage devices that host the data. These devices
have no knowledge of files, mandatory locks, or share reservations,
and are not in a position to enforce such restrictions. For this
reason the NFSv4 server MUST NOT grant layouts that conflict with
mandatory locks or share reservations. Further, if a conflicting
mandatory lock request or a conflicting open request arrives at the
server, the server MUST recall the part of the layout in conflict
with the request before granting the request.

2.3.6. End-of-file Processing

   The end-of-file location can be changed in two ways: implicitly as
   the result of a WRITE or LAYOUTCOMMIT beyond the current end-of-file,
   or explicitly as the result of a SETATTR request. Typically, when a
   file is truncated by an NFSv4 client via the SETATTR call, the server
   frees any disk blocks belonging to the file that are beyond the new
   end-of-file byte, and MUST write zeros to the portion of the new
   end-of-file block beyond the new end-of-file byte. These actions
   render any pNFS layouts that refer to the blocks that are freed or
   written semantically invalid. Therefore, the server MUST recall from
   clients the portions of any pNFS layouts that refer to blocks that
   will be freed or written by the server before processing the truncate
   request. These recalls may take time to complete; as explained in
   [NFSv4.1], if the server cannot respond to the client SETATTR request
   in a reasonable amount of time, it SHOULD reply to the client with
   the error NFS4ERR_DELAY.

   Blocks in the PNFS_BLOCK_INVALID_DATA state that lie beyond the new
   end-of-file block present a special case. The server has reserved
   these blocks for use by a pNFS client with a writable layout for the
   file, but the client has yet to commit the blocks, and they are not
   yet a part of the file mapping on disk. The server MAY free these
   blocks while processing the SETATTR request. If so, the server MUST
   recall any layouts from pNFS clients that refer to the blocks before
   processing the truncate. If the server does not free the
   PNFS_BLOCK_INVALID_DATA blocks while processing the SETATTR request,
   it need not recall layouts that refer only to the PNFS_BLOCK_INVALID
   DATA blocks.

   When a file is extended implicitly by a WRITE or LAYOUTCOMMIT beyond
   the current end-of-file, or extended explicitly by a SETATTR request,
   the server need not recall any portions of any pNFS layouts.

2.3.7. Layout Hints

   The SETATTR operation supports a layout hint attribute [NFSv4.1].
   When the client sets a layout hint (data type layouthint4) with a
   layout type of LAYOUT4_BLOCK_VOLUME (the loh_type field), the
   loh_body field contains a value of data type pnfs_block_layouthint4.

```
   /// /* block layout specific type for loh_body */
   /// struct pnfs_block_layouthint4 {
   ///     uint64_t blh_maximum_io_time;  /* maximum i/o time in seconds
   ///                                        */
   /// };
   ///
```

The block layout client uses the layout hint data structure to
communicate to the server the maximum time that it may take an I/O to
execute on the client. Clients using block layouts MUST set the
layout hint attribute before using LAYOUTGET operations.

2.3.8. Client Fencing

The pNFS block protocol must handle situations in which a system
failure, typically a network connectivity issue, requires the server
to unilaterally revoke extents from one client in order to transfer
the extents to another client. The pNFS server implementation MUST
ensure that when resources are transferred to another client, they
are not used by the client originally owning them, and this must be
ensured against any possible combination of partitions and delays
among all of the participants to the protocol (server, storage and
client). Two approaches to guaranteeing this isolation are possible
and are discussed below.

One implementation choice for fencing the block client from the block
storage is the use of LUN masking or mapping at the storage systems
or storage area network to disable access by the client to be
isolated. This requires server access to a management interface for
the storage system and authorization to perform LUN masking and
management operations. For example, the Storage Management
Initiative Specification (SMI-S) [SMIS] provides a means to discover
and mask LUNs, including a means of associating clients with the
necessary World Wide Names or Initiator names to be masked.

In the absence of support for LUN masking, the server has to rely on
the clients to implement a timed-lease I/O fencing mechanism.
Because clients do not know if the server is using LUN masking, in
all cases, the client MUST implement timed-lease fencing. In timed-
lease fencing, we define two time periods, the first, "lease_time" is
the length of a lease as defined by the server's lease_time attribute
(see [NFSv4.1]), and the second, "blh_maximum_io_time" is the maximum
time it can take for a client I/O to the storage system to either
complete or fail; this value is often 30 seconds or 60 seconds, but
may be longer in some environments. If the maximum client I/O time
cannot be bounded, the client MUST use a value of all 1s as the
blh_maximum_io_time.

After a new client ID is established, the client MUST use SETATTR
with a layout hint of type LAYOUT4_BLOCK_VOLUME to inform the server
of its maximum I/O time prior to issuing the first LAYOUTGET
operation. While the maximum I/O time hint is a per-file attribute,
it is actually a per-client characteristic. Thus, the server MUST
maintain the last maximum I/O time hint sent separately for each
client. Each time the maximum I/O time changes, the server MUST

apply it to all files for which the client has a layout. If the
client does not specify this attribute on a file for which a block
layout is requested, the server SHOULD use the most recent value
provided by the same client for any file; if that client has not
provided a value for this attribute, the server SHOULD reject the
layout request with the error NFS4ERR_LAYOUTUNAVAILABLE. The client
SHOULD NOT send a SETATTR of the layout hint with every LAYOUTGET. A
server that implements fencing via LUN masking SHOULD accept any
maximum I/O time value from a client. A server that does not
implement fencing may return an error NFS4ERR_INVAL to the SETATTR
operation. Such a server SHOULD return NFS4ERR_INVAL when a client
sends an unbounded maximum I/O time (all 1s), or when the maximum I/O
time is significantly greater than that of other clients using block
layouts with pNFS.

When a client receives the error NFS4ERR_INVAL in response to the
SETATTR operation for a layout hint, the client MUST NOT use the
LAYOUTGET operation. After responding with NFS4ERR_INVAL to the
SETATTR for layout hint, the server MUST return the error
NFS4ERR_LAYOUTUNAVAILABLE to all subsequent LAYOUTGET operations from
that client. Thus, the server, by returning either NFS4ERR_INVAL or
NFS4_OK determines whether or not a client with a large, or an
unbounded-maximum I/O time may use pNFS.

Using the lease time and the maximum I/O time values, we specify the
behavior of the client and server as follows.

When a client receives layout information via a LAYOUTGET operation,
those layouts are valid for at most "lease_time" seconds from when
the server granted them. A layout is renewed by any successful
SEQUENCE operation, or whenever a new stateid is created or updated
(see the section "Lease Renewal" of [NFSv4.1]). If the layout lease
is not renewed prior to expiration, the client MUST cease to use the
layout after "lease_time" seconds from when it either sent the
original LAYOUTGET command or sent the last operation renewing the
lease. In other words, the client may not issue any I/O to blocks
specified by an expired layout. In the presence of large
communication delays between the client and server, it is even
possible for the lease to expire prior to the server response
arriving at the client. In such a situation, the client MUST NOT use
the expired layouts, and SHOULD revert to using standard NFSv41 READ
and WRITE operations. Furthermore, the client must be configured
such that I/O operations complete within the "blh_maximum_io_time"
even in the presence of multipath drivers that will retry I/Os via
multiple paths.

As stated in the "Dealing with Lease Expiration on the Client"
section of [NFSv4.1], if any SEQUENCE operation is successful, but
sr_status_flag has SEQ4_STATUS_EXPIRED_ALL_STATE_REVOKED,
SEQ4_STATUS_EXPIRED_SOME_STATE_REVOKED, or
SEQ4_STATUS_ADMIN_STATE_REVOKED is set, the client MUST immediately
cease to use all layouts and device ID to device address mappings
associated with the corresponding server.

In the absence of known two-way communication between the client and
the server on the fore channel, the server must wait for at least the
time period "lease_time" plus "blh_maximum_io_time" before
transferring layouts from the original client to any other client.
The server, like the client, must take a conservative approach, and
start the lease expiration timer from the time that it received the
operation that last renewed the lease.

2.4. Crash Recovery Issues

A critical requirement in crash recovery is that both the client and
the server know when the other has failed. Additionally, it is
required that a client sees a consistent view of data across server
restarts. These requirements and a full discussion of crash recovery
issues are covered in the "Crash Recovery" section of the NFSv41
specification [NFSv4.1]. This document contains additional crash
recovery material specific only to the block/volume layout.

When the server crashes while the client holds a writable layout, and
the client has written data to blocks covered by the layout, and the
blocks are still in the PNFS_BLOCK_INVALID_DATA state, the client has
two options for recovery. If the data that has been written to these
blocks is still cached by the client, the client can simply re-write
the data via NFSv4, once the server has come back online. However,
if the data is no longer in the client's cache, the client MUST NOT
attempt to source the data from the data servers. Instead, it should
attempt to commit the blocks in question to the server during the
server's recovery grace period, by sending a LAYOUTCOMMIT with the
"loca_reclaim" flag set to true. This process is described in detail
in Section 18.42.4 of [NFSv4.1].

2.5. Recalling Resources: CB_RECALL_ANY

The server may decide that it cannot hold all of the state for
layouts without running out of resources. In such a case, it is free
to recall individual layouts using CB_LAYOUTRECALL to reduce the
load, or it may choose to request that the client return any layout.

The NFSv4.1 spec [NFSv4.1] defines the following types:

    const RCA4_TYPE_MASK_BLK_LAYOUT = 4;

    struct CB_RECALL_ANY4args {
          uint32_t   craa_objects_to_keep;
          bitmap4   craa_type_mask;
    };

When the server sends a CB_RECALL_ANY request to a client specifying
the RCA4_TYPE_MASK_BLK_LAYOUT bit in craa_type_mask, the client
should immediately respond with NFS4_OK, and then asynchronously
return complete file layouts until the number of files with layouts
cached on the client is less than craa_object_to_keep.

2.6. Transient and Permanent Errors

The server may respond to LAYOUTGET with a variety of error statuses.
These errors can convey transient conditions or more permanent
conditions that are unlikely to be resolved soon.

The transient errors, NFS4ERR_RECALLCONFLICT and NFS4ERR_TRYLATER,
are used to indicate that the server cannot immediately grant the
layout to the client. In the former case, this is because the server
has recently issued a CB_LAYOUTRECALL to the requesting client,
whereas in the case of NFS4ERR_TRYLATER, the server cannot grant the
request possibly due to sharing conflicts with other clients. In
either case, a reasonable approach for the client is to wait several
milliseconds and retry the request. The client SHOULD track the
number of retries, and if forward progress is not made, the client
SHOULD send the READ or WRITE operation directly to the server.

The error NFS4ERR_LAYOUTUNAVAILABLE may be returned by the server if
layouts are not supported for the requested file or its containing
file system. The server may also return this error code if the
server is the progress of migrating the file from secondary storage,
or for any other reason that causes the server to be unable to supply
the layout. As a result of receiving NFS4ERR_LAYOUTUNAVAILABLE, the
client SHOULD send future READ and WRITE requests directly to the
server. It is expected that a client will not cache the file's
layoutunavailable state forever, particular if the file is closed,
and thus eventually, the client MAY reissue a LAYOUTGET operation.

3. Security Considerations

Typically, SAN disk arrays and SAN protocols provide access control
mechanisms (e.g., LUN mapping and/or masking) that operate at the
granularity of individual hosts. The functionality provided by such

mechanisms makes it possible for the server to "fence" individual
client machines from certain physical disks -- that is to say, to
prevent individual client machines from reading or writing to certain
physical disks. Finer-grained access control methods are not
generally available. For this reason, certain security
responsibilities are delegated to pNFS clients for block/volume
layouts. Block/volume storage systems generally control access at a
volume granularity, and hence pNFS clients have to be trusted to only
perform accesses allowed by the layout extents they currently hold
(e.g., and not access storage for files on which a layout extent is
not held). In general, the server will not be able to prevent a
client that holds a layout for a file from accessing parts of the
physical disk not covered by the layout. Similarly, the server will
not be able to prevent a client from accessing blocks covered by a
layout that it has already returned. This block-based level of
protection must be provided by the client software.

An alternative method of block/volume protocol use is for the storage
devices to export virtualized block addresses, which do reflect the
files to which blocks belong. These virtual block addresses are
exported to pNFS clients via layouts. This allows the storage device
to make appropriate access checks, while mapping virtual block
addresses to physical block addresses. In environments where the
security requirements are such that client-side protection from
access to storage outside of the authorized layout extents is not
sufficient, pNFS block/volume storage layouts SHOULD NOT be used
unless the storage device is able to implement the appropriate access
checks, via use of virtualized block addresses or other means. In
contrast, an environment where client-side protection may suffice
consists of co-located clients, server and storage systems in a data
center with a physically isolated SAN under control of a single
system administrator or small group of system administrators.

This also has implications for some NFSv4 functionality outside pNFS.
For instance, if a file is covered by a mandatory read-only lock, the
server can ensure that only readable layouts for the file are granted
to pNFS clients. However, it is up to each pNFS client to ensure
that the readable layout is used only to service read requests, and
not to allow writes to the existing parts of the file. Similarly,
block/volume storage devices are unable to validate NFS Access
Control Lists (ACLs) and file open modes, so the client must enforce
the policies before sending a READ or WRITE request to the storage
device. Since block/volume storage systems are generally not capable
of enforcing such file-based security, in environments where pNFS
clients cannot be trusted to enforce such policies, pNFS block/volume
storage layouts SHOULD NOT be used.

   Access to block/volume storage is logically at a lower layer of the
   I/O stack than NFSv4, and hence NFSv4 security is not directly
   applicable to protocols that access such storage directly. Depending
   on the protocol, some of the security mechanisms provided by NFSv4
   (e.g., encryption, cryptographic integrity) may not be available or
   may be provided via different means. At one extreme, pNFS with
   block/volume storage can be used with storage access protocols (e.g.,
   parallel SCSI) that provide essentially no security functionality.
   At the other extreme, pNFS may be used with storage protocols such as
   iSCSI that can provide significant security functionality. It is the
   responsibility of those administering and deploying pNFS with a
   block/volume storage access protocol to ensure that appropriate
   protection is provided to that protocol (physical security is a
   common means for protocols not based on IP). In environments where
   the security requirements for the storage protocol cannot be met,
   pNFS block/volume storage layouts SHOULD NOT be used.

   When security is available for a storage protocol, it is generally at
   a different granularity and with a different notion of identity than
   NFSv4 (e.g., NFSv4 controls user access to files, iSCSI controls
   initiator access to volumes). The responsibility for enforcing
   appropriate correspondences between these security layers is placed
   upon the pNFS client. As with the issues in the first paragraph of
   this section, in environments where the security requirements are
   such that client-side protection from access to storage outside of
   the layout is not sufficient, pNFS block/volume storage layouts
   SHOULD NOT be used.

4. Conclusions

   This document specifies the block/volume layout type for pNFS and
   associated functionality.

5. IANA Considerations

   There are no IANA considerations in this document. All pNFS IANA
   Considerations are covered in [NFSv4.1].

6. Acknowledgments

   This document draws extensively on the authors' familiarity with the
   mapping functionality and protocol in EMC's Multi-Path File System
   (MPFS) (previously named HighRoad) system [MPFS]. The protocol used
   by MPFS is called FMP (File Mapping Protocol); it is an add-on
   protocol that runs in parallel with file system protocols such as
   NFSv3 to provide pNFS-like functionality for block/volume storage.
   While drawing on FMP, the data structures and functional
   considerations in this document differ in significant ways, based on

lessons learned and the opportunity to take advantage of NFSv4
features such as COMPOUND operations. The design to support pNFS
client participation in copy-on-write is based on text and ideas
contributed by Craig Everhart.

Andy Adamson, Ben Campbell, Richard Chandler, Benny Halevy, Fredric
Isaman, and Mario Wurzl all helped to review versions of this
specification.

7. References

7.1. Normative References

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

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

   [NFSv4.1] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed.,
              "Network File System (NFS) Version 4 Minor Version 1
              Protocol", RFC 5661, January 2010.

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

7.2. Informative References

   [MPFS]   EMC Corporation, "EMC Celerra Multi-Path File System
              (MPFS)", EMC Data Sheet,
              http://www.emc.com/collateral/software/data-sheet/
              h2006-celerra-mpfs-mpfsi.pdf.

   [SMIS]   SNIA, "Storage Management Initiative Specification (SMI-S)
              v1.4", http://www.snia.org/tech_activities/standards/
              curr_standards/smi/SMI-S_Technical_Position_v1.4.0r4.zip.

Authors' Addresses

   David L. Black
   EMC Corporation
   176 South Street
   Hopkinton, MA 01748

   Phone: +1 (508) 293-7953
   EMail: black_david@emc.com

   Stephen Fridella
   Nasuni Inc
   313 Speen St
   Natick MA 01760

   EMail: stevef@nasuni.com

   Jason Glasgow
   Google
   5 Cambridge Center
   Cambridge, MA 02142

   Phone: +1 (617) 575 1599
   EMail: jglasgow@aya.yale.edu