Linux pNFS Client Design Document for Pluggable Layout Drivers

(DRAFT)

Dean Hildebrand Andy Adamson Bruce Fields Peter Honeyman
Center for Information Technology Integration
University of Michigan
{dhildebz, andros, bfields, honey}@umich.edu

Marc Eshel
IBM Almaden Research Center
eshel@almaden.ibm.com

Trond Myklebust
Network Appliance, Inc.
trond.myklebust@fys.uio.no

1. Introduction

This document provides an overview of the current Linux pNFS client architecture. This architecture enables pluggable layout drivers, allowing the layout driver for a particular file system to be chosen at mount-time. Layout drivers implement a standard functional interface to manage layout specific I/O operations as well as a standard policy interface to answer runtime policy decisions.

2. Overall pNFS architecture

In pNFS, the NFS client and server continue to perform control and file management operations and relegate the responsibility for achieving scalable I/O throughput to a storage-specific driver. By separating control and data flows, pNFS allows data to transfer in parallel from many clients to many storage endpoints. This removes the single server bottleneck by distributing I/O across the bisectional bandwidth of the storage network between the clients and storage devices.

Figure 1 depicts the architecture of pNFS, which adds a layout driver, an I/O driver, and a file layout retrieval interface (export operations) to the standard NFSv4 architecture.

The *layout driver* understands the file layout of the storage system. A layout consists of all information required to access any byte range of a file. For example, a block layout may contain information about block size, offset of the first block on each storage device, and an array of tuples that contains device identifiers, block numbers, and block counts. An object layout specifies the storage devices for a file and the information necessary to translate a logical byte sequence into a collection of objects. A file layout is similar to an object layout but uses file handles instead of object identifiers. The layout driver uses the layout to translate read and write requests from the pNFS client into I/O requests understood by the storage devices. The I/O driver performs raw I/O, e.g., Myrinet GM, Infiniband, TCP/IP, to the storage nodes.

To ensure support for all I/O protocols, the pNFS implementation for each operation system must include a standard interface that the layout driver implements. The layout driver can be specialized or (preferably) implement a standard protocol such as the Fibre Channel Protocol (FCP), allowing multiple file systems to all share the same layout driver. Storage systems adopting this architecture reduce development and management obligations by obviating a specialized file system client, which reduces the cost of high-end storage systems.

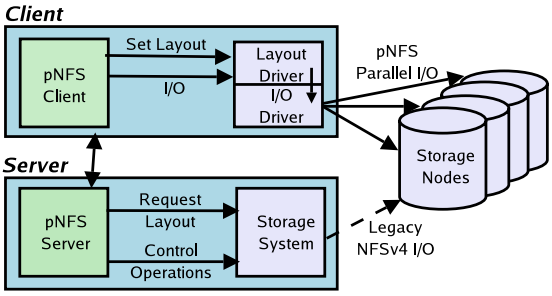

Figure 1. pNFS architecture

pNFS extends NFSv4 with the addition of a layout driver, an I/O driver, and a file layout retrieval interface. The pNFS server obtains an opaque file layout map from the storage system and transfers it to the pNFS client and subsequently to its layout driver for direct and parallel data access.

3. pNFS Design

3.1. Design goals

The goals of this pNFS design are:

- Support any storage protocol, including but not limited to block, object, and file storage protocols.
- Pluggable layout drivers with a standard interface.
- A clear division of responsibilities between the pNFS client and layout driver.
- Enable layout driver specific policies that govern its runtime behavior.
- Allow the pNFS client to export a set of optional features to a layout driver, e.g., data cache, writeback cache. The pNFS client implements all pNFS operations, providing the layout driver with an interface for their execution, e.g., GETDEVICEINFO
- Allow layout drivers to provide custom client feature implemtations such as data cache management, writeback cache, etc.

3.2. pNFS Client Design

An NFSv4 client no longer assumes SUNRPC is the only I/O protocol. If a layout driver exists for a given file system, it provides the I/O capabilities.

For NFSv2 and v3, the client follows the standard I/O code path. With NFSv4, we have instrumented the client I/O code path with several hooks to query the policies of the layout driver for the file system. Depending on these policies, the pNFS client will call operations on the layout driver as necessary. If no layout driver exists for a given file system (superblock), the NFSv4 client also follows the standard I/O path.

We currently envision three sets of interfaces that enable communication between the pNFS client and a layout driver. The first is the layout driver operations interface, which gives a set of operations required to set layout information and perform I/O to the storage nodes. The second is a layout driver policy interface, which provides a set of operations that allow a pNFS client to retrieve the runtime characteristics of a layout driver, e.g., file system block size, whether the layout driver wants to use the NFSv4 writeback cache. The third and final interface is currently undefined, but will provide a set of operations layout drivers can use to call into the pNFS client, e.g., I/O callbacks. There should be no need for layout drivers to use sunrpc directly to contact the NFSv4 server.

This document presents a minimal design for each interface that meets the (functional) needs of the PVFS2 and NFSv4 file layout drivers. These interfaces may not currently meet the needs of other layout drivers, e.g., objects, blocks, so let the design discussion begin!

3.2.1. Layout driver registration

A layout driver is a kernel module that registers itself with the pNFS client when it is loaded.

To register, a module calls:

```
struct pnfs_layoutdriver_type {
 const      int id;
 const      char *name;
 struct layoutdriver_io_operations *io_ops;
 struct layoutdriver_policy_operations
*p_ops;
};

struct pnfs_client_operations*
            pnfs_register_layoutdriver(
            struct pnfs_layoutdriver_type *);
```

To unregister, a module calls:

```
void
pnfs_unregister_layoutdriver(
            struct pnfs_layoutdriver_type *);
```

The variable id is the unique id of the layout driver, and maps directly to the new file system attribute, LAYOUT_CLASSES, which is the list of layout driver id's that a file system supports. The layoutdriver_operations struct is the I/O operations supported by the layout driver (Section 3.5). Currently, if the pNFS server supports multiple layout driver types, the pNFS client will use the first one listed in the LAYOUT_CLASSES attribute.

The returned pnfs_client_operations struct gives the list of callback operations supported by the pNFS client.

3.2.2. Setting the layout driver for a file system

A pNFS client retrieves the LAYOUT_CLASSES attribute when encountering an unknown file system identifier (at mount time). If a layout driver with a

matching id exists, the layout driver's operations are set in the superblock. If a matching layout driver is not registered, the client uses standard NFSv4. To prevent namespace collisions, a global registry maintainer such as IANA should store the layout driver identifiers.

3.2.3. Layout driver operations interface

There is much debate on the definition of layout driver's interface. Determining a division of labor between the pNFS client and its layout drivers that satisfies the needs of all underlying file systems will surely be an ongoing process.

A single layout driver instance manages each mount point. At mount time, the pNFS client calls the layout driver's `initialize_mountpoint` operation to inform the layout driver of a new super block. The `initialize_mountpoint` returns a `pnfs_mount_type` structure, which contains a pointer to layout driver specific mount information. The pNFS client stores the `pnfs_mount_type` structure in the nfs super block and uses it for all layout driver file system operations.

For each inode, the pNFS client initially calls the layout driver's alloc_layout operation to retrieve a reference to a `pnfs_layout_type` structure. The `pnfs_layout_type` structure contains a pointer to layout driver specifc layout information and is stored in the nfs inode. The pNFS client injects layout information returned by LAYOUTGET into the layout driver through the set_layout operation.

The `pnfs_mount_type` and `pnfs_layout_type` structures enable storing mount and layout information in an layout independent manner while eliminating hashtable lookups within the layout driver.

The syntax for these functions is:

```
/* Layout driver specific identifier for a mount
point.  For each mountpoint a reference is
stored in the nfs_server structure. */
struct pnfs_mount_type {
         void* mountid;
};

/* Layout driver specific identifier for layout
information for a file.  Each inode has a
specific layout type structure. A reference is
stored in the nfs_inode structure. */
struct pnfs_layout_type {
         struct pnfs_mount_type* mountid;
         void* layoutid;
};

/* Layout driver I/O operations.  Either the
pagecache or non-pagecache read/write operations
must be implemented */
struct layoutdriver_io_operations {

/* Functions that use the pagecache.  If
use_pagecache == 1, then these functions must be
implemented. */
ssize_t (*read_pagelist) (
         struct pnfs_layout_type * layoutid,
         struct inode *,
         unsigned int pgbase,
         struct page** pagevec,
         loff_t offset,
         size_t count,
         void* private);

ssize_t (*write_pagelist) (
         struct pnfs_layout_type * layoutid,
         struct inode *,
         unsigned int pgbase,
         struct page** pagevec,
         loff_t offset,
         size_t count,
         int sync,
         void* private);

/* Functions that do not use the pagecache.  If
use_pagecache == 0, then these functions must be
implemented. */
ssize_t (*read) (
         struct pnfs_layout_type * layoutid,
         struct file*,
         char __user *,
         size_t, loff_t *);

ssize_t (*write) (
         struct pnfs_layout_type * layoutid,
         struct file*,
         const char __user *,
         size_t,
         loff_t *);

ssize_t (*readv) (
         struct pnfs_layout_type * layoutid,
         struct file*,
         const struct iovec *,
         unsigned long, loff_t *);

ssize_t (*writev) (
         struct pnfs_layout_type * layoutid,
         struct file*,
         const struct iovec *,
         unsigned long,
         loff_t *);

/* Consistency ops */
int (*fsync) (
         struct pnfs_layout_type * layoutid,
         struct file *,
         struct dentry *,
         int);

int (*commit) (
         struct pnfs_layout_type * layoutid,
         struct inode * inode,
         struct list_head * pagelist,
         int sync,
         void* private);

/* Layout information. For each inode,
alloc_layout is executed once to retrieve an
inode specific layout structure.  Each
subsequent layoutget operation results in a
set_layout call to set the opaque layout in the
layout driver.  free_layout deallocs layout
resources */
struct pnfs_layout_type* (*alloc_layout) (
         struct pnfs_mount_type * mountid,
         struct inode * inode);

void (*free_layout) (
         struct pnfs_layout_type * layoutid,
         struct inode * inode);

struct pnfs_layout_type* (*set_layout) (
```

```
        struct pnfs_layout_type * layoutid,
        struct inode * inode,
        void* layout);

/* Registration information for a new mounted
file system */
struct pnfs_mount_type*(*initialize_mountpoint)(
        struct super_block *);
int (*uninitialize_mountpoint) (
        struct pnfs_mount_type* mountid);

/* Allow custom/extra behavior through ioctl,
just like a file system */
int (*ioctl) (
        struct pnfs_layout_type *,
        struct inode *,
        struct file *,
        unsigned int,
        unsigned long);
};
```

To inject the file layout map, the pNFS client passes the opaque array as an argument to the `set_layout` function. Once the layout driver has finished processing the layout, the pNFS client is free to call the driver's read and write functions.

3.2.4. Layout driver policy interface

The following is a first draft of layout driver policies:

```
struct layoutdriver_policy_operations {
/* The stripe size of the file system */
ssize_t (*get_stripesize) (
       struct pnfs_layout_type * layoutid,
       struct inode *);

/* Should the NFS req. gather algorithm cross
stripe boundaries? */
int (*gather_across_stripes) (
       struct pnfs_mount_type * mountid);

/* Retreive the block size of the file system.
If gather_across_stripes == 1, then the file
system will gather requests into the block size
*/
ssize_t (*get_blocksize) (
       struct pnfs_mount_type *);

/* I/O requests under this value are sent to the
NFSv4 server */
int (*get_io_threshold) (
       struct pnfs_layout_type *,
       struct inode*);

/* Use the linux page cache prior to calling
layout driver read/write functions */
int (*use_pagecache) (struct pnfs_layout_type *,
struct inode *);

/* Should the pNFS client issue a layoutget call
in the same compound as the OPEN operation? */
int (*layoutget_on_open) (
       struct pnfs_mount_type *);
};
```

I would like to see additional policies that allow more flexible I/O semantics such as:

- Layout driver support for write or read gathering and use of the writeback cache without use of the data cache. This is the default I/O policy in Lustre and critical to improve the efficiency of small I/Os.

3.2.5. I/O Driver

Currently the I/O driver is more of a design concept than a reality. Both the NFSv4 file and PVFS2 layout drivers integrate the I/O driver into the layout driver. The NFSv4 file layout driver only supports sunrpc (although this is changing with new technologies such as RDMA). The PVFS2 layout driver only supports its custom built BMI protocol.

3.2.6. pNFS client callback interface

```
struct pnfs_device
{
     unsigned int  layoutclass;
     int           dev_id;
     unsigned int  dev_addr_len;
     char          dev_addr_buf[MAXSIZE];
};

struct pnfs_devicelist {
     unsigned int        num_devs;
     unsigned int        devs_len;
     struct pnfs_device  devs[MAXCOUNT];
};

struct pnfs_client_operations {
int (*nfs_fsync) (
     struct file * file,
     struct dentry * dentry,
     int datasync);

int (*nfs_getdevicelist) (
     struct super_block * sb,
     struct pnfs_devicelist* devlist);

int (*nfs_getdeviceinfo) (
     struct super_block * sb,
     u32 dev_id,
     struct pnfs_device * dev);
};
```

Depending on where the asynchrony starts, we may need I/O callbacks. The NFSv4 layout driver pushes asynchrony to the sunrpc driver, and therefore does not require I/O callbacks either.

3.2.7. Retrieving device information

The GETDEVINFO and GETDEVLIST operations retrieve additional information about one or more storage nodes. The layout driver has the option to call these operations at any time through the pNFS client callback interface. We suspect the most common point will occur when a file system is mounted inside the layout driver initialization operation.

One example of when GETDEVINFO may be executed is after the use of GETDEVLIST to get more detailed device information, e.g., SAN volume label information or port numbers.

3.2.8. Write design

The pNFS client uses a writeback cache to gather write requests into `wsize` requests. Write requests larger than the `wsize` are split into `wsize`-sized requests. The maximum wsize is 32 KB. When an inode is flushed—either due to memory pressure, file close, or application fsync—all dirty pages written to the server. The list of pages to be flushed is handed off to the layout driver through the `writepage` operation.

Execution of `writepage` is currently a synchronous operation. The current NFSv4 implementation does not directly support asynchronous I/O, instead relying on the multithreaded sunrpc implementation for asynchronous I/O support.

If the server is mounted with the sync option set, all pages are immediately written to the data servers.

Issuing I/O in 32 KB requests severely reduces performance for some parallel file systems, e.g., PVFS2, which are designed for large block sizes. As a workaround, the current pNFS implementation has added another layout driver I/O interface operation, `write`. This operation is invoked before the pNFS client's use of the writeback and data caches, allowing the original write request to be processed by the layout driver.

The current pNFS implementation does not support O_DIRECT.

3.2.9. Read design

The pNFS read code path is very similar to the write code path. pNFS read requests are gathered or split into `rsize` (32 KB) requests. The Linux readahead algorithm determines the read request offset and extent arguments. A list of pages to be filled are handed to the layout driver `readpage` operation.

Execution of `readpage` is currently a synchronous operation. The current NFSv4 implementation does not directly support asynchronous I/O, instead relying on the multithreaded sunrpc libraries for asynchronous I/O support.

If the server is mounted with the sync option set, all pages are immediately filled through read requests to the data servers.

Issuing I/O in 32 KB requests severely reduces performance for some parallel file systems, e.g., PVFS2, which are designed for large block sizes. As a workaround, the current pNFS implementation has added another layout driver I/O interface operation, `read`. This operation bypasses the data cache, allowing the original read request to be processed by the layout driver.

The current pNFS implementation does not support O_DIRECT.

3.2.10. Layout management

The LAYOUTGET operation obtains file access information for a byte-range of a file, i.e., the file layout, from the underlying storage system. At no time does the pNFS client attempt to interpret this object, it acts simply as a conduit between the storage system and the layout driver. The byte range described by the returned layout may be smaller or larger than the requested size due to block alignments, layout prefetching, space limitations, etc.

The client issues a LAYOUTGET operation after opening a file and before reading or writing file data. The NFSv4 file layout driver issues a LAYOUTGET within the same compound as OPEN, whereas the PVFS2 layout driver issues LAYOUTGET operations as required.

The current pNFS client implementation assumes the layout driver contains a layout cache, allowing repeated reads and writes to the layout driver once the layout is set. The pNFS client tracks the range for which layouts are acquired, issuing LAYOUTGET operations for byte ranges of the file not currently covered in the layout driver's layout cache.

3.2.11. Commit and Layoutcommit

The current pNFS client implementation of LAYOUTCOMMIT only updates the pNFS server with the lastbytewritten field. If the lastbytewritten field is greater than the current file size, the pNFS server updates the size of the file.

Currently, an NFSv4 client sends a COMMIT to the server in response to client memory pressure or just before the file is closed. With pNFS, instead of executing a COMMIT to the server, the pNFS client calls the layout driver `commit` operation. Once the layout driver is finished commit data to the data servers, the pNFS client issues a LAYOUTCOMMIT operation to the server. Our implementation currently tracks the last byte written to an inode between invocations of LAYOUTCOMMIT and uses this value for LAYOUTCOMMIT.

3.2.12. Returning a layout

This operation informs the server that obtained layout information is no longer required. Clients return a layout voluntarily or when they receive a server recall request. The current code implements this operation but does not execute it.

4. Layout Driver Specific Information

4.1. NFSv4 File Layout

- Issues LAYOUTGET with OPEN (Since this is an optimization, we currently retrieve layout info as needed. We will implement this when the code is more stable.)

- Ensures that the pNFS client does not gather read and write requests across stripe boundaries.
- Asynchronous I/O support via sunrpc

4.2. PVFS2

- Issue LAYOUTGET as needed