CITI Technical Report 08-1

Parallel NFS Block Layout Module for Linux

William A. Adamson, University of Michigan
andros@citi.umich.edu
Frederic Isaman, University of Michigan
iisaman@citi.umich.edu
Jason Glasgow, EMC
Glasgow_Jason@emc.com

ABSTRACT

This position statement presents CITI's Linux prototype of NFSv4.1 pNFS client block layout module and reviews our implementation approach. CITI's prototype implements the IETF draft specification draft-ietf-nfsv4-pnfs-block and is one of three layout modules being developed along with the Linux pNFS generic client, which implements the draft-ietf-nfsv4-minorversion1 specification. The block layout module provides for an I/O data path over iSCSI directly to client SCSI devices identified by the pNFS block server.

February 12, 2008

Center for Information Technology Integration
University of Michigan
535 W. William St., Suite 3100
Ann Arbor, MI 48103-4978

Parallel NFS Block Layout Module for Linux

William A. Adamson, University of Michigan
andros@citi.umich.edu

Frederic Isaman, University of Michigan
iisaman@citi.umich.edu

Jason Glasgow, EMC
Glasgow_Jason@emc.com

Introduction

This position statement presents CITI's Linux prototype of NFSv4.1 pNFS client block layout module and reviews our implementation approach. CITI's prototype implements the IETF draft specification draft-ietf-nfsv4-pnfs-block and is one of three layout modules being developed along with the Linux pNFS generic client, which implements the draft-ietf-nfsv4-minorversion1 specification. The block layout module provides for an I/O data path over iSCSI directly to client SCSI devices identified by the pNFS block server.

CITI has also developed a Python-based NFSv4.1 test environment -- an LVM-based pNFS block layout server that supports SCSI disks emulated in RAM and the iSCSI protocol -- to test direct block I/O and complex volume topologies along with the pNFS and other NFSv4.1 operations.

We refer to draft-ietf-nfsv4-minorversion1-17.txt Section 12 for a detailed description of NFSv4.1 Parallel NFS.

http://www1.ietf.org/internet-drafts/draft-ietf-nfsv4-minorversion1-17.txt

Here are snippets of the introduction to set the stage.

```
12.1.  Introduction
   pNFS is a set of optional features within NFSv4.1; the pNFS feature
   set allows direct client access to the storage devices containing
   file data.  When file data for a single NFSv4 server is stored on
   multiple and/or higher throughput storage devices (by comparison to
   the server's throughput capability), the result can be significantly
   better file access performance.

   pNFS takes the form of OPTIONAL operations that manage protocol
   objects called 'layouts' which contain data location information.

   The NFSv4.1 pNFS feature has been structured to allow for a variety
   of storage protocols to be defined and used.

   The NFSv4.1 protocol directly defines one storage protocol, the NFSv4.1
   storage type, and its use.

   Examples of other storage protocols that could be used with NFSv4.1's
   pNFS are:

   o  Block/volume protocols such as iSCSI ([35]), and FCP ([36]).  The
      block/volume protocol support can be independent of the addressing
      structure of the block/volume protocol used, allowing more than
      one protocol to access the same file data and enabling
      extensibility to other block/volume protocols.

   o  Object protocols such as OSD over iSCSI or Fibre Channel [37].

   o  Other storage protocols, including PVFS and other file systems
      that are in use in HPC environments.
```

The pNFS block layout module is specified in draft-ietf-nfsv4-pnfs-block-05.txt

http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-pnfs-block-05.txt

The pNFS operations carry an opaque payload, which conforms to a storage protocol layout type, between the pNFS client and the pNFS server. The Linux pNFS client and server prototype implementations reflect this design.

The Linux pNFS generic client performs tasks common to all layout types. Per layout type payloads are opaque to this generic code. The generic code passes an opaque payload to the appropriate registered layout module via a layout module API, which runs code specific to the layout type.

The Linux NFSv4.1 pNFS server exports pNFS capable file systems. The pNFS server design is similar to the pNFS client. The pNFS server performs tasks common to all layout types. Opaque payloads are passed to and from the exported pNFS capable file system via a new set of operations in struct export_operations.

A team of engineers from CITI, Network Appliance, Panasas, and IBM are implementing the Linux pNFS client and server prototypes. The Linux block layout module is based on the EMC MPFS file system client, which shares many design points with pNFS.

The Linux pNFS client and server prototypes have been tested at four interoperability events where the Linux pNFS client demonstrated support for simultaneous multiple layout modules, and the Linux pNFS server was used to export object (Panasas) and NFSv4.1 (IBM and Network Appliance) based file systems.

Protocol Requirements

Here is a list of the requirements placed on the pNFS block layout client by the protocol. We are considering only the iSCSI storage protocol transport at this time.

1) Identify storage volumes by content

2) Support arbitrarily complex volume topologies per file system id

3) Break down and reset logical disk/volume topology

4) I/O requires mapping file offset extent volume logical offset physical disk + offset

5) Block I/O to SCSI disk

6) 0 ≤ write size ≤ server file system block size

7) Fail over to NFSv4.1 server

8) Specify the maximum I/O time of the I/O stack

9) Copy-on-write support

Here are the kernel interfaces that our block layout module prototype uses to implement the requirements.

Prototype Kernel Interfaces

The first job is to identify which of the SCSI disks the client can see belong to the pNFS server.

1) Identify storage volumes by content.

Our prototype does the following.

- EXPORT drivers/scsi/hosts.c: shost_class symbol lists all SCSI devices. We walk the list, and identify all SCSI disks that can be open_by_dev() and bd_claim().
- We retrieve a list of device IDs from the pNFS block server and corresponding content at an offset. For each device ID, the prototype reads each SCSI disk comparing content at the offset. When we find a match, the pNFS server device ID is associated with the disk. We bd_release() and close all un-associated disks.

INTERFACE:
a) shost_class
b) open_by_dev
c) bd_claim
d) bios read.

The next three requirements inspired our use of the LVM interface.

2) Support arbitrarily complex volume topologies per file system id
3) Break down and reset logical disk/volume topology
4) I/O requires mapping file offset extent volume logical offset physical disk + offset

These requirements seemed to be a good match for the user land LVM2 software. When our prototype gets a volume topology with disks identified via content, we call the LVM2 services in the kernel.

- Create a dm device (a la user land LVM ioctl interface) to represent volume topology.
- For I/O, the dm device handles the logical offset to physical disk + offset mapping for us.

INTERFACE: create a dm device

This does most of what we want. We could write our own dm target. We use the ioctl interface in the kernel. Splitting dm functionality to service both the existing ioctl interface and the desired kernel interface remains an issue.

The dm device gives us a device for performing I/O. Next, we are faced with the need to choose the best kernel interface for the following requirements.

5) Block I/O to SCSI disk

6) 0 ≤ write size ≤ server file system block size

7) Fail over to NFSv4.1 server

INTERFACE: We have implemented two paths:

- Use the block_read/write_full_page() interface.
  - We need our own callback routine for cleanup and error recovery.
  - Calls bios, and does it right!
  - Large server file system block size is an issue because we might have to deal with fs block sizes larger than a page.
- Code our own bios routines.
  - Write case is complex
  - It is easy to code our own callback routine

Issues

We have yet to address these remaining requirements:

8) Specify the maximum I/O time of the I/O stack

9) Copy-on-write support

Knowing when to give up on I/O to a disk is an issue. We have not yet addressed the copy-on-write requirement.

We hope to discuss this and other pNFS position statements at the Linux File Systems Workshop, which will help us move pNFS into the Linux kernel over the next 1–2 years.