NFSv4 Working Group
Internet Draft
Expires: December 2005

David L. Black
Stephen Fridella
EMC Corporation
June 3, 2005

pNFS Block/Volume Layout
draft-black-pnfs-block-00.txt

Status of this Memo

Copyright Notice

Abstract

Parallel NFS (pNFS) extends 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 draft specifies storage-class-independent extensions to NFS; this draft specifies the additional extensions (primarily data structures) for use of pNFS with block and volume based storage.

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].

Table of Contents

NOTE: This is an early stage draft. It's still rough in places, with significant work to be done.

1. Introduction

Figure 1 shows the overall architecture of a pNFS system:

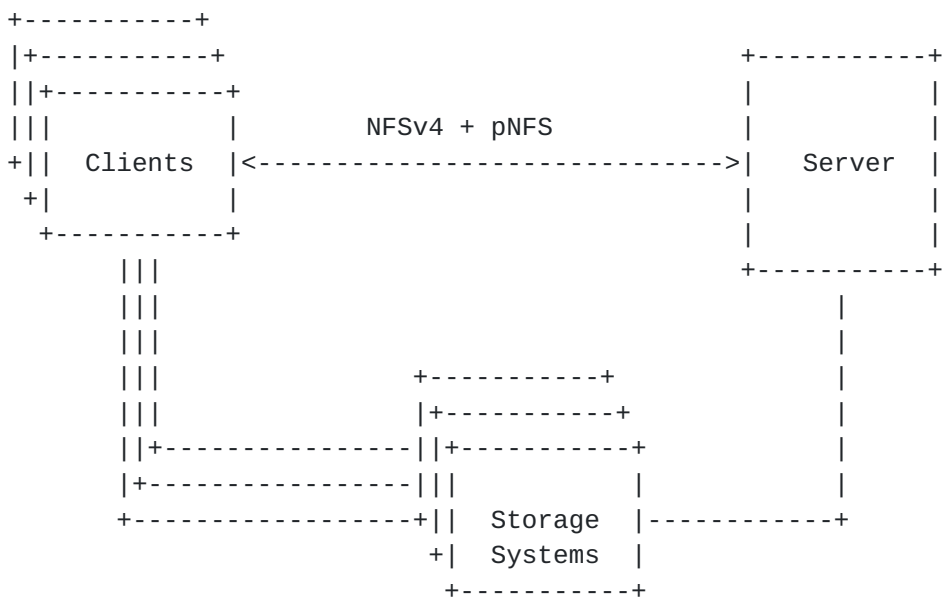

Figure 1 pNFS Architecture

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 [WELCH-OPS] for more details. This draft is concerned with access from pNFS clients to Storage Systems over storage protocols based on blocks and volumes, such as the SCSI protocol family (e.g., parallel SCSI, FCP for Fibre Channel, iSCSI, SAS). This class of storage is referred to as block/volume storage. While the Server to Storage System protocol is not of concern for interoperability here, it will typically also be a block/volume protocol when clients use block/volume protocols.

2. 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 SHOULD be aligned to the block size used by the NFSv4 server in managing the actual filesystem (4 kilobytes and 8 kilobytes are common block sizes).

OPEN ISSUE: Client ability to ask server for block size - if block size is constant per filesystem (fsid), it can enable internal client optimizations. Constant filesystem block size is probably the common case - an additional (required) FS attribute would suffice.

This draft draws extensively on the authors' familiarity with the the mapping functionality and protocol in EMC's HighRoad system. The protocol used by HighRoad is called FMP (File Mapping Protocol); it is an add-on protocol that runs in parallel with filesystem protocols such as NFSv3 to provide pNFS-like functionality for block/volume storage. While drawing on HighRoad FMP, the data structures and functional considerations in this draft differ in significant ways, based on lessons learned and the opportunity to take advantage of NFSv4 features such as COMPOUND operations.

2.1. Data Structures: Extents and Extent Lists

A pNFS layout is a list of extents with associated properties. EAch extent MUST be at least 512-byte aligned.

```
struct extent {

  offset4      file_offset;/* the logical location in the file */

  length4      extent_length; /* the size of this extent in file and
                                 and on storage */

  pnfs_deviceid4  volume_ID;  /* the logical volume/physical device
                                 that this extent is on */

  offset4      storage_offset;/* the logical location of
                              this extent in the volume */

  extentState4    es;   /* the state of this extent */

};
```

```
enum extentState4 {

  VALID_DATA = 0, /* the data located by this extent is valid for
                     reading and writing. */

  INVALID_DATA = 1,  /* the location is valid; the data is invalid.
                        It could be overwritten by the valid data.
                        It is a newly (pre-) allocated extent. There
                        is physical space. */

  NONE_DATA = 2,  /* the location is invalid. It is a hole in the
                     file. There is no physical space. */

  };
```

The file_offset, extent_length, and es fields for an extent returned from the server are always valid. The interpretation of the storage_offset field depends on the value of es as follows:

- VALID_DATA means that storage_offset is valid, and points to valid/initialized data which can and should be fetched from the disk to satisfy read requests (and partial-block write requests).

- INVALID_DATA means that storage_offset is valid, but points to invalid uninitialized data. This data must not be physically read from the disk until it has been initialized. Read request from an INVALID_DATA extent, must fill the user buffer with zeros. Write requests must write whole blocks to the disk. Bytes not initialized by the user must be set to zero. INVALID_DATA extents are returned by requests for writeable extents; they are never returned if the request was only for reading..

- NONE_DATA means that 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 INVALID_DATA. NONE_DATA extents are returned by requests for readable extents; they are never returned if the request was for a writeable extent.

The volume_ID field for an extent returned by the server is used to identify the logical volume on which this extent resides, and its interpretation depends on the volume-management protocol being used by the client and server.

The extent list lists all relevant extents in increasing order of the file_offset of each extent.

```
typedef extent extentList<MAX_EXTENTS>;  /* MAX_EXTENTS = 256; */
```

2.1.1. Layout Requests and Extent Lists

Each request for a layout specifies at least three parameters: offset, desired size, and minimum size (the desired size is missing from the operations draft - see Section 3). If the status of a request indicates success, the extent list returned must meet the following criteria:

- A request for a readable (but not writeable layout returns only VALID_DATA or NONE_DATA extents (but not INVALID_DATA extents).
- A request for a writeable layout returns only VALID_DATA or INVALID_DATA extents (but not NONE_DATA extents).
- The first extent in the list MUST contain the starting offset.
- The total size of extents in the extent list MUST cover at least the minimum size and no more than the desired size. One exception is allowed: the total size MAY be smaller if only readable extents were requested and EOF is encountered.
- Extents in the extent list MUST be logically contiguous and non-overlapping).

2.1.2. Extents Have Lock-like Behavior

Extents returned to pNFS clients function as locks in that they grant clients permission to read or write. Both read/write and write/write conflicts must be controlled by the pNFS server as a read/write conflict may cause a read to return a mixture of before-write and after-write data from a block-based storage system and a write/write conflict may cause the result on the block-based storage system to be a mixture of data from the two write operations; both of these outcomes are unacceptable, as in the absence of pNFS, the NFSv4 server would have correctly sequenced the conflicting operations to avoid this mixing. This is particularly nasty if the underlying storage is striped and the operations complete in different orders on the different stripes.

A client which makes a layout request that conflicts with an existing layout delegation will be rejected with the error NFS4_Locked (OPEN_ISSUE: New error code needed?). This client is then expected

to retry the request after a short interval. During this interval the server needs to recall the conflicting portion of the layout delegation from the client that currently holds it. It has been noted that this mode of reject/retry operation does not prevent a requesting client from being starved 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 IO operations that are permissible to other clients. Since all IO operations ultimately arrive at the NFSv4 server for processing, the server is in a position to enforce these restrictions. However, with pNFS layout delegations, IOs will be issued from the clients that hold the delegations 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 layout delegations that conflict with mandatory locks or share reservations. Furthermore, if a conflicting mandatory lock request or a conflicting open request arrives at the server, the server must recall the part of the layout delegation in conflict with the request before processing the request.

2.2. 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 address 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. 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.

A volume is content-identified by a disk signature made up of extents within blocks and contents that must match.

block_device_addr_list - A list of the disk signatures for the physical volumes on which the file system resides. This is list of variable number of diskSigInfo structures. This is the device_addr_list<> as returned by GETDEVICELIST in [WELCH-OPS]

```
typedef diskSigInfo block_device_addr_list<MAX_DEVICE>;
         /* disksignature  info */
```

where diskSigInfo is:

```
struct diskSigInfo {       /* used in DISK_SIGNATURE */
   diskSig        ds;      /* disk signature */

   pnfs_deviceid4 volume_ID;  /* volume ID the server will use in
                                 extents. */

};
```

where diskSig is defined as:

```
typedef sigComp diskSig<MAX_SIG_COMPONENTS>;

struct sigComp {        /*  disk signature component */

   offset4  sig_offset; /* byte offset of component */

   length4  sig_length; /* byte length of component */

   sigCompContents contents;  /* contents of this component of the
                                 signature (this is opaque) */

};
```

sigCompContents MUST NOT be interpreted as a zero-terminated string, as it may contain embedded zero-valued octets. It contains sig_length octets. There are no restrictions on alignment (e.g., neither sig_offset nor sig_length need to be multiples of 4).

3. Operations Issues

This section collects issues in the operations draft encountered in writing this block/volume layout draft.

1. Request for a layout (LAYOUTGET) only conveys minimum required size - for the block storage class, a desired size is also useful. This allows the client to ask for a good size for performance but allow the server to reduce the size when other clients are actively writing different areas of the file for conflict management.

2. The operations draft treats a layout returned by an operation as an indivisible object (at least for callback and return - commit seems to only be able to handle one extent). For block storage layouts, it is important to be able to recall, commit, or return a portion of a layout. The server needs to be in control of the conflict granularity to minimize the impact of false sharing, and the client needs to be able to manage its layout state in a flexible fashion.

3. Need a callback to set EOF. The underlying issue here is that block pNFS clients have to handle EOF enforcement because the Storage Systems have no concept of file, let alone EOF. Hence client interactions based on EOF changes (e.g., one client truncates file, another tries to write beyond new EOF) require updates to tell clients that the EOF has moved. Calling back layouts beyond the new EOF to force the client to check for EOF change is both inefficient and overkill.

4. HighRoad supports three additional types of layout recalls - "everything in a file", "everything in a list of files", "everything in a filesystem". HighRoad also supports an "everything in a file" layout return. The "everything in a file" type is very convenient to get rid of all state for a file. The "everything in a filesystem" is crucial to get unmount of a busy filesystem to actually work. The "everything in a list of files" turns out to be useful for quota situations, although it's a bit blunt - when a user is nearing her quota, recall her writeable layouts to force the commits needed to manage the quota. OPEN ISSUE: This may not be the best way to handle approaching a quota limit.

5. Access and Modify time behavior. Any LAYOUTCOMMIT operation should implicitly set both the Access and Modify times. LAYOUTRETURN needs flags saying whether to set Access time or Access and Modify times or neither.

6. The disk signature approach to volume identification is noted in the [WELCH-OPS] draft, but the data structures in the -01 version of that draft do not support it.

3.1. Ordering Issues

This deserves its own subsection because there is some serious subtlety here. High Road uses two mechanisms for ordering:

1. In contrast to NFSv4 callbacks that expect immediate responses, HighRoad layout callback responses may be delayed to allow a client to perform any required commits, etc. prior to responding to the callback. This allows the reply to the callback to serve as an implicit return of the recalled range or ranges. For a simple return case, this saves a round trip (client replies to callback, doesn't have to issue a separate return). Another useful case is that the response to a set EOF callback discards all layout info beyond the block containing the new EOF (need filesystem block size attribute for this to work). If NFSv4 style callbacks that expect immediate responses are used, the client has to perform an explicit LAYOUTRETURN.

2. HighRoad uses a server message number for operation sequencing, which appears to correspond well to the layout stateid in [WELCH-OPS], except that the server message number has per-file rather than per-layout scope. The pNFS layout stateid should probably have per-file scope in order to deal well with Issue 1 in Section 3 above. The server message number serves to ensure that a pNFS client can process pNFS server replies (operation completions) and callbacks *in the same order* as the pNFS server.

The delayed callback response creates an ordering issue in that the client may immediately issue a LAYOUTGET for the range that its callback reply returns - if that request crosses the callback reply on the wire, the server must detect this reordering and tell the client to retry. This does not require a sequence number/stateid mechanism - the server must wait for the callback to finish before processing any conflicting LAYOUTGET from the same client. With an NFSv4-style callback, the client must wait for its LAYOUTRETURN to complete before issuing the LAYOUTGET, so this issue does not arise.

In the reverse direction, the same "cross on the wire" scenario applies, and requires a sequencing mechanism. The server may issue a recall for a range covered by a LAYOUTGET immediately after returning the layout to the client. If the recall arrives first, the client has to queue it until the LAYOUTGET result comes back and process the callback against that new layout. A variant on this that appears

similar to the client but requires a different response occurs when the server issued the recall before processing the LAYOUTGET; in this case the server will reject the LAYOUTGET as having a stale sequence number/stateid (because that number/stateid was incremented by the recall callback) and the client needs to process the callback before retrying the LAYOUTGET.

3.2. Crash Recovery Issues

Client recovery for layout delegations works in much the same way as NFSv4 client recovery for other lock/delegation state. When an NFSv4 client reboots, it will lose all information about the layout delegations that it previously owned. There are two methods by which the server can reclaim these resources and begin providing them to other clients. The first is through the expiry of the client's lock/delegation lease. If the client recovery time is longer than the lease period, the client's lock/delegation lease will expire and the server will know to reclaim any state held by the client. On the other hand, the client may recover in less time than it takes for the lease period to expire. In such a case, the client will be required to contact the server through the standard SETCLIENTID protocol. The server will find that the client's id matches the id of the previous client invocation, but that the verifier is different. The server uses this as a signal to reclaim all the state associated with the client's previous invocation.

The server recovery case is slightly more complex. In general, the recovery process will again follow the standard NFSv4 recovery model: the client will discover that the server has rebooted when it receives an unexpected STALE_STATEID or STALE_CLIENTID reply from the server; it will then proceed to try to reclaim its previous delegations during the server's recovery grace period. However there is an important safety concern associated with layout delegations that does not come into play in the standard NFSv4 case. If a standard NFSv4 client makes use of a stale delegation, the consequence could be to deliver stale data to an application. However, the pNFS layout delegation enables the client to directly access the file system storage---if this access is not properly managed by the NFSv4 server the client can potentially corrupt the file system data or meta-data.

Thus it is vitally important that the client discover that the server has rebooted as soon as possible, and that the client stops using stale layout delegations before the server gives the delegations away to other clients. To ensure this, the client must be implemented so that layout delegations are never used to access the storage after the client's lease timer has expired. This prohibition applies to

all accesses, especially the flushing of dirty data to storage. If the client's lease timer expires because the client could not contact the server for any reason, the client MUST immediately stop using the layout delegation until the server can be contacted and the delegation can be officially recovered or reclaimed.

3.3. Additional Features - Not Needed or Recommended

This subsection is a place to record things that existing SAN or clustered filesystems do that aren't needed or recommended for pNFS:

- Callback for write-to-read downgrade. Writers tend to want to remain writers, so this feature isn't very useful.

- HighRoad FMP implements several frequently used operation combinations as single RPCs for efficiency; these can be effectively handled by NFSv4 COMPOUNDs. One subtle difference is that a single RPC is treated as a single operation, whereas NFSv4 COMPOUNDs are not atomic in any sense. This can cause operation ordering subtleties, such as having to set the new EOF *before* returning the layout extent that contains the new EOF, even within a single COMPOUND.

- Queued request support. The HighRoad FMP protocol specification allows the server to return an "operation blocked" result code with a cookie that is later passed to the client in a "it's done now" callback. This has not proven to be of great use vs. having the client retry with some sort of back-off. Recommendations on how to back off should be added to the ops draft.

- Additional client and server crash detection mechanisms. As a separate protocol, HighRoad FMP had to handle this on its own. As an NFSv4 extension, NFSv4's SETCLIENTID, STALE CLIENTID and STALE STATEID mechanisms combined with implicit lease renewal and (per-file) layout stateids should be sufficient for pNFS.

- The use of separate read and write layouts to enable client participation in copy-on-write (as in IBM's SAN.FS) does not seem to be important to pNFS; this may be an implementation approach that is unique to SAN.FS .

4. Security Considerations

Certain security responsibilities are delegated to pNFS clients. 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 it currently holds

(e.g., and not access storage for files on which a layout extent is not held). 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 read-layout-delegations for the file are granted to pNFS clients. However, it is up to each pNFS client to ensure that the read layout delegation 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, block/volume-based pNFS SHOULD NOT be used.

<TBD: Need discussion about security for block/volume protocol vis-a-vis NFSv4 security. Client may not even use same identity for both (e.g., for Fibre Channel, same identity as NFSv4 is impossible). Need to talk about consistent security protection of data via NFSv4 vs. direct block/volume access. Some of this extends discussion in previous paragraph about client responsibility for security as part of overall system.>

5. Conclusions

<TBD: Add any conclusions>

6. Acknowledgments

This draft draws extensively on the authors' familiarity with the the mapping functionality and protocol in EMC's HighRoad system. The protocol used by HighRoad is called FMP (File Mapping Protocol); it is an add-on protocol that runs in parallel with filesystem protocols such as NFSv3 to provide pNFS-like functionality for block/volume storage. While drawing on HighRoad FMP, the data structures and functional considerations in this draft differ in significant ways, based on lessons learned and the opportunity to take advantage of NFSv4 features such as COMPOUND operations.

7. References

7.1. Normative References

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

[WELCH-OPS] Welch, B., et. al. "pNFS Operations Summary", draft-welch-pnfs-ops-01.txt, Work in Progress, May 2005.

TODO: Need to reference RFC 3530.

7.2. Informative References

OPEN ISSUE: HighRoad and/or SAN.FS references?

Author's Addresses

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

Phone: +1 (978) 263-0937
Email: black_david@emc.com

Stephen Fridella
EMC Corporation
32 Coslin Drive
Southboro, MA 01772

Phone: +1 (508) 305-8512
Email: fridella_stephen@emc.com

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.