NFSv4 Working Group
Internet Draft
Expires: April 2006

David L. Black
Stephen Fridella
EMC Corporation
October 21, 2005

pNFS Block/Volume Layout
draft-black-pnfs-block-02.txt

Status of this Memo

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:

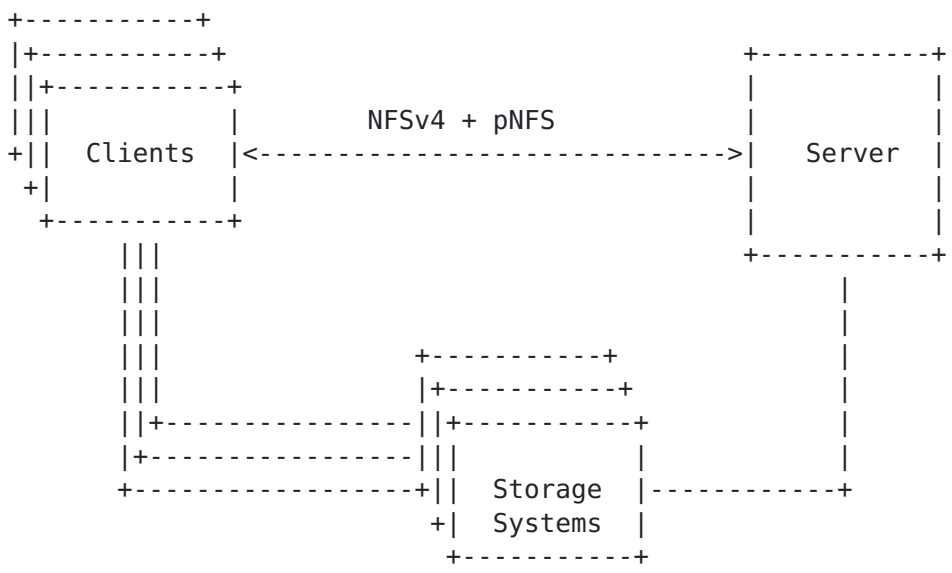

Figure 1 pNFS Architecture

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 [PNFS] 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). This block size is available as an NFSv4 attribute - see Section 7.4 of [PNFS].

This draft relies on the pNFS client indicating whether a requested layout is for read 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 uninitialized storage in those holes (read as zero, can be written by client). This draft also supports client participation in copy on write by providing both read-only and uninitialized storage for the same range in a layout. Reads are initially performed on the read-only storage, with writes going to the uninitialized storage. After the first write that initializes the uninitialized storage, all reads are performed to that now-initialized writeable storage, and the corresponding read-only storage is no longer used.

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. The support for client participation in copy-on-write is based on contributions from those with experience in that area, as HighRoad does not currently support client participation in copy-on-write.

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 {

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

     READ_DATA = 1, /* the data located by this extent is valid for
                       reading only; it may not be written. */

     INVALID_DATA = 2,  /* the location is valid; the data is invalid.
                           It is a newly (pre-) allocated extent.
                           There is physical space. */

     NONE_DATA = 3,  /* 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:

- READ_WRITE_DATA means that storage_offset is valid, and points to valid/initialized data that can be read and written.

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

- 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. A read request for an INVALID_DATA extent must fill the user buffer with zeros. Write requests must write whole blocks to the disk with bytes not initialized by the user must be set to zero. Any write to storage in an INVALID_DATA extent changes the written portion of the extent to READ_WRITE_DATA; the pNFS client is responsible for reporting this change via LAYOUTCOMMIT.

- 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, see Section 2.2.

The extent list lists all relevant extents in increasing order of the file_offset of each extent; any ties are broken by increasing order of the extent state (es).

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

TODO: Define the actual layout and layoutupdate data structures as extent lists.

TODO: Striping support. Layout independent striping will not be added to [PNFS], but it can help compact layout representations when the filesystem is striped across block/volume storage.

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 READ_WRITE_DATA, READ_DATA or NONE_DATA extents (but not INVALID_DATA extents). A READ_WRITE_DATA extent MAY be returned by a pNFS server in a readable layout in order to avoid a subsequent client request for writing (ISSUE: Is that a good idea? It involves server second-guessing client, and the downside is the possible need for a recall callback).

- A request for a writeable layout returns READ_WRITE_DATA or INVALID_DATA extents (but not NONE_DATA extents). It may also return READ_DATA extents only when the offset ranges in those extents are also covered by INVALID_DATA extents to permit writes.

- 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 for a read-only layout. For a read-write layout, the set of writable extents (i.e., excluding READ_DATA extents) MUST be logically contiguous. Every READ_DATA extent in a read-write layout MUST be covered by an INVALID_DATA extent. This overlap of READ_DATA and INVALID_DATA extents is the only permitted extent overlap.

- Extents MUST be ordered in the list by starting offset, with READ_DATA extents preceding INVALID_DATA extents in the case of equal file_offsets.

2.1.2. Client Copy-on-Write Processing

Distinguishing the READ_WRITE_DATA and READ_DATA extent types combined with the allowed overlap of READ_DATA extents with 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 requires clients to participate in copy-on-write operations.

When a client wishes to write data covered by a READ_DATA extent, it MUST have requested a writable layout from the server; that layout will contain INVALID_DATA extents to cover all the data ranges of that layout's READ_DATA extents. More precisely, for any file_offset range covered by one or more READ_DATA extents in a writable layout, the server MUST include one or more INVALID_DATA extents in the layout that cover the same file_offset range. The client MUST

logically copy the data from the READ_DATA extent for any partial blocks of file_offset and range, merge in the changes to be written, and write the result to the INVALID_DATA extent for the blocks for that file_offset and range. That is, if entire blocks of data are to be overwritten by an operation, the corresponding READ_DATA blocks need not be fetched, but any partial-block writes must be merged with data fetched via READ_DATA extents before storing the result via INVALID_DATA extents. Storing of data in an INVALID_DATA extent converts the written portion of the INVALID_DATA extent to a READ_WRITE_DATA extent; all subsequent reads MUST be performed from this extent; the corresponding portion of the READ_DATA extent MUST NOT be used after storing data in an INVALID_DATA extent.

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

ISSUE: This assumes that all block/volume pNFS clients will support copy-on-write. Negotiating this would require additional server code to cope with clients that don't support this, which doesn't seem like a good idea.

2.1.3. Extents are Permissions

Layout extents returned to pNFS clients grant permission to read or write; READ_DATA and NONE_DATA are read-only (NONE_DATA reads as zeroes), READ_WRITE_DATA and INVALID_DATA are read/write, (INVALID_DATA reads as zeros, any write converts it to READ_WRITE_DATA). This is the only client means 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; this can be particularly nasty if the underlying storage is striped and the operations complete in different orders on different stripes. A pNFS server can avoid these conflicts by implementing a single writer XOR multiple readers concurrency control policy when there are multiple clients who wish to access the same data. This policy SHOULD be implemented when storage devices do not provide atomicity for concurrent read/write and write/write operations to the same data.

A client that makes a layout request that conflicts with an existing layout delegation 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 needs to 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 layout delegations, I/Os 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. 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 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 [PNFS]

```
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 are required to be multiples of 4).

3. Operations Issues

NOTE: This section and its subsections are preserved for historical/review purposes only, as the [PNFS] draft has addressed all of these issues. The section and all subsections will be deleted in the next version of this draft.

This section collects issues in the operations draft encountered in writing this block/volume layout draft. Most of these issues are expected to be resolved in draft-welch-pnfs-ops-03.txt .

1. RESOLVED: LAYOUTGET provides minimum and desired (max) lengths to server.

2. RESOLVED: Layouts are managed by offset and range; they are no longer treated as indivisible objects.

3. RESOLVED: There is a callback for the server to convey a new EOF to the client.

4. RESOLVED: HighRoad supports three types of layout recalls beyond range recalls: "everything in a file", "everything in a list of files", "everything in a filesystem". The first and third are supported in [PNFS] (set offset to zero and length to all 1's for everything in a file - [PNFS] implies this, but isn't explicit about it), and the second one can probably be done as a COMPOUND with reasonable effectiveness. LAYOUTRETURN supports return of everything in a file in a similar fashion (offset of zero, length of all 1's).

5. RESOLVED: Access and Modify time behavior. LAYOUTCOMMIT operation sets both Access and Modify times. LAYOUTRETURN cannot set either time - use a SETATTR in a COMPOUND to do this (Q: Can this inadvertently make time run backwards?).

6. RESOLVED: The disk signature approach to volume identification appears to be supportable via the opaque pnfs_devaddr4 union element.

7. RESOLVED: The LAYOUTCOMMIT operation has no LAYOUTRETURN side effects in -03. If it ever did, they were not intended.

3.1. Layout Operation Ordering Considerations

This deserves its own subsection because there is some serious subtlety here.

In contrast to NFSv4 callbacks that expect immediate responses, HighRoad layout callback responses are delayed to allow the 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 and tell the server that all callback related processing has been completed by the client. For consistency, pNFS should use the NFSv4 callback approach in which immediate responses are expected. As a result all returns of layout ranges MUST be explicit.

3.1.1. Client Side Considerations

Consider a pNFS client that has issued a LAYOUTGET and then receives an overlapping recall callback for the same file. There are two possibilities, which the client cannot distinguish when the callback arrives:

1. The server processed the LAYOUTGET before issuing the recall, so the LAYOUTGET response is in flight, and must be waited for because it may be carrying layout info that will need to be returned to deal with the recall callback.

2. The server issued the callback before receiving the LAYOUTGET. The server will not respond to the LAYOUTGET until the recall callback is processed.

This can cause deadlock, as the client must wait for the LAYOUTGET response before processing the recall in the first case, but that response will not arrive until after the recall is processed in the second case. The deadlock is avoided via a simple rule:

> RULE: A LAYOUTGET MUST be rejected with an error if there's an overlapping outstanding recall callback to the same client. The client MUST process the outstanding recall callback before retrying the LAYOUTGET.

Now the client can wait for the LAYOUTGET response because it will come in both cases. This RULE also applies to the callback to send an updated EOF to the client.

The resulting situation is still less than desired, because issuance of a recall callback indicates a conflict and potential contention at the server, so recall callbacks should be processed as fast as possible by clients. In the second case, if the client knows that the LAYOUTGET will be rejected, it is beneficial for the client to process the recall immediately without waiting for the LAYOUTGET rejection. To do so without added client complexity, the server needs to reject the LAYOUTGET even if it arrives at the server after the client operations that process the recall callback; if the client still wants that layout, it can reissue the LAYOUTGET.

HighRoad uses the equivalent of a per-file layout stateid to enable this optimization. The layout stateid increments on each layout operation completion and callback issuance, and the current value of the layout stateid is sent in every operation response and every callback. If the initial layout stateid value is N, then in the first case above, the recall callback carries stateid N+2 indicating that the LAYOUTGET response is carrying N+1 and hence has to be waited for. In the second case above, the recall callback carries layout stateid N+1 indicating that the LAYOUTGET will be rejected with a stale layout stateid (N where N+1 or greater is current) whenever it arrives, and hence the callback can be processed immediately. This per-file layout stateid approach entails prohibiting concurrent callbacks to the for the same file to the same client, as server issuance of a new callback could cause stale layout stateid errors for operations that the client is performing to deal with an earlier recall callback.

ISSUE: Does restricting all pNFS client operations on the same file to a single session help?

3.1.2. Server Side Considerations

Consider a related situation from the pNFS server's point of view. The server has issued a recall callback and receives an overlapping LAYOUTGET for the same file before the LAYOUTRETURN(s) that respond to the recall callback. Again, there are two cases:

1. The client issued the LAYOUTGET before processing the recall callback. The LAYOUTGET MUST be rejected according to the RULE in the previous subsection.

2. The client issued the LAYOUTGET after processing the recall callback, but it arrived before the LAYOUTRETURN that completed that processing.

The simplest approach is to apply the RULE and always reject the overlapping LAYOUTGET. The client has two ways to avoid this result - it can issue the LAYOUTGET as a subsequent element of a COMPOUND containing the LAYOUTRETURN that completes the recall callback, or it can wait for the response to that LAYOUTRETURN.

This leads to a more general problem; in the absence of a callback if a client issues concurrent overlapping LAYOUTGET and LAYOUTRETURN operations, it is possible for the server to process them in either order. HighRoad forbids a client from doing this, as the per-file layout stateid will cause one of the two operations to be rejected with a stale layout stateid. This approach is simpler and produces better results by comparison to allowing concurrent operations, at least for this sort of conflict case, because server execution of operations in an order not anticipated by the client may produce results that are not useful to the client (e.g., if a LAYOUTRETURN is followed by a concurrent overlapping LAYOUTGET, but executed in the other order, the client will not retain layout extents for the overlapping range).

3.2. Recall Callback Completion and Robustness Concerns

The discussion of layout operation ordering implicitly assumed that any callback results in a LAYOUTRETURN or set of LAYOUTRETURNs that match the range in the callback. This envisions that the pNFS client state for a file match the pNFS server state for that file and client regarding layout ranges and permissions. That may not be the best design assumption because:

1. It may be useful for clients to be able to discard layout information without calling LAYOUTRETURN. If conflicts that require callbacks are rare, and a server can use a multi-file callback to recover per-client resources (e.g., via a multi-file recall operation based on some sort of LRU), the result may be significantly less client-server pNFS traffic.

2. It may be similarly useful for servers to enhance information about what layout ranges are held by a client beyond what a client actually holds. In the extreme, a server could manage conflicts on a per-file basis, only issuing whole-file callbacks even though clients may request and be granted sub-file ranges.

3. The synchronized state assumption is not robust to minor errors. A more robust design would allow for divergence between client and server and the ability to recover. It is vital that a client not assign itself layout permissions beyond what the server has granted and that the server not forget layout permissions that have been granted in order to avoid errors. OTOH, if a server believes that a client holds an extent that the client doesn't know about, it's useful for the client to be able to issue the LAYOUTRETURN that the server is expecting in response to a recall.

At a minimum, in light of the above, it is useful for a server to be able to issue callbacks for layout ranges it has not granted to a client, and for a client to return ranges it does not hold. This leads to a couple of requirements:

A pNFS client's final operation in processing a recall callback SHOULD be a LAYOUTRETURN whose range matches that in the callback. If the pNFS client holds no layout permissions in the range that has been recalled, it MUST respond with a LAYOUTRETURN whose range matches that in the callback.

This avoids any need for callback cookies (server to client) that would have to be returned to indicate recall callback completion.

For a callback to set EOF, the client MUST logically apply the new EOF before issuing the response to the callback, and MUST NOT issue any other pNFS operations before responding to the callback.

ISSUE: HighRoad FMP also requires that LAYOUTCOMMIT operations be stalled at the server while an EOF callback is outstanding.

3.3. 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.

3.4. 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 may not be 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 result in operation ordering subtleties, e.g., 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.

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. IANA Considerations

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

7. Revision History

-00: Initial Version

-01: Rework discussion of extents as locks to talk about extents granting access permissions. Rewrite operation ordering section to discuss deadlocks and races that can cause problems. Add new section on recall completion. Add client copy-on-write based on text from Craig Everhart.

-02: Fix glitches in extent state descriptions. Describe most issues as RESOLVED. Most of Section 3 has been incorporated into the [PNFS] draft, add NOTE to that effect and say that it will be deleted in the next version of this draft (which should be a draft-ietf-nfsv4 draft). Cleaning up a number of things have been left to that draft revision, including the interlocks with the types in [PNFS], layout striping support, and finishing the Security Considerations section.

8. 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. The design to support pNFS client participation in copy-on-write is based on text and ideas contributed by Craig Everhart of IBM.

9. References

9.1. Normative References

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

[PNFS] Goodson, G., et. al. "NFSv4 pNFS Extensions", draft-ietf-nfsv4-pnfs-00.txt, Work in Progress, October 2005.

TODO: Need to reference RFC 3530.

9.2. 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 (508) 293-7953
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.