NFSv4
Internet-Draft
Expires: April 10, 2006

G. Goodson
NetApp
B. Welch
B. Halevy
Panasas
D. Black
EMC
A. Adamson
CITI

October 7, 2005

NFSv4 pNFS Extensions
draft-ietf-nfsv4-pnfs-00.txt

Status of this Memo

Copyright Notice

Abstract

This Internet-Draft provides a description of the pNFS extension for NFSv4.

The key feature of the protocol extension is the ability for clients to perform read and write operations that go directly from the client to individual storage system elements without funneling all such accesses through a single file server. Of course, the file server must provide sufficient coordination of the client I/O so that the file system retains its integrity.

The extension adds operations that query and manage layout information that allows parallel I/O between clients and storage system elements. The layouts are managed in a similar way to delegations in that they are associated with leases and can be recalled by the server, but layout information is independent of delegations.

Requirements Language

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 [1].

Table of Contents

1. Introduction

The NFSv4 protocol [2] specifies the interaction between a client that accesses files and a server that provides access to files and is responsible for coordinating access by multiple clients. As described in the pNFS problem statement, this requires that all access to a set of files exported by a single NFSv4 server be performed by that server; at high data rates the server may become a bottleneck.

The parallel NFS (pNFS) extensions to NFSv4 allow data accesses to bypass this bottleneck by permitting direct client access to the storage devices containing the 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. The relationship among multiple clients, a single server, and multiple storage devices for pNFS (server and clients have access to all storage devices) is shown in this diagram:

```
       +-----------+
       |+-----------+                                +-----------+
       ||+-----------+                               |           |
       |||           |        NFSv4 + pNFS           |           |
       +||  Clients  |<------------------------------>|   Server  |
        +|           |                                |           |
         +-----------+                                |           |
              |||                                     +-----------+
              |||                                           |
              |||                                           |
              ||| Storage       +-----------+               |
              ||| Protocol      |+-----------+              |
              ||+----------------||+-----------+  Control|
              |+-----------------|||           |    Protocol|
              +------------------+||  Storage  |------------+
                                  +|  Devices  |
                                   +-----------+
```

Figure 1

In this structure, the responsibility for coordination of file access by multiple clients is shared among the server, clients, and storage devices. This is in contrast to NFSv4 without pNFS extensions, in which this is primarily the server's responsibility, some of which can be delegated to clients under strictly specified conditions.

The pNFS extension to NFSv4 takes the form of new operations that manage data location information called a "layout". The layout is

managed in a similar fashion as NFSv4 data delegations (e.g., they are recallable and revocable). However, they are distinct abstractions and are manipulated with new operations that are described in Section 9. When a client holds a layout, it has rights to access the data directly using the location information in the layout.

There are new attributes that describe general layout characteristics. However, much of the required information cannot be managed solely within the attribute framework, because it will need to have a strictly limited term of validity, subject to invalidation by the server. This requires the use of new operations to obtain, return, recall, and modify layouts, in addition to new attributes.

This document specifies both the NFSv4 extensions required to distribute file access coordination between the server and its clients and a NFSv4 file storage protocol that may be used to access data stored on NFSv4 storage devices.

Storage protocols used to access a variety of other storage devices are deliberately not specified here. These might include:

- Block/volume protocols such as iSCSI ([4]), and FCP ([5]). 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.

- Object protocols such as OSD over iSCSI or Fibre Channel [6].

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

pNFS is designed to accommodate these protocols and be extensible to new classes of storage protocols that may be of interest.

The distribution of file access coordination between the server and its clients increases the level of responsibility placed on clients. Clients are already responsible for ensuring that suitable access checks are made to cached data and that attributes are suitably propagated to the server. Generally, a misbehaving client that hosts only a single-user can only impact files accessible to that single user. Misbehavior by a client hosting multiple users may impact files accessible to all of its users. NFSv4 delegations increase the level of client responsibility as a client that carries out actions requiring a delegation without obtaining that delegation will cause its user(s) to see unexpected and/or incorrect behavior.

Some uses of pNFS extend the responsibility of clients beyond delegations. In some configurations, the storage devices cannot perform fine-grained access checks to ensure that clients are only performing accesses within the bounds permitted to them by the pNFS operations with the server (e.g., the checks may only be possible at file system granularity rather than file granularity). In situations where this added responsibility placed on clients creates unacceptable security risks, pNFS configurations in which storage devices cannot perform fine-grained access checks SHOULD NOT be used. All pNFS server implementations MUST support NFSv4 access to any file accessible via pNFS in order to provide an interoperable means of file access in such situations. See Section 4 on Security for further discussion.

Finally, there are issues about how layouts interact with the existing NFSv4 abstractions of data delegations and byte range locking. These issues, and others, are also discussed here.

2. General Definitions

This protocol extension partitions the NFSv4 file system protocol into two parts, the control path and the data path. The control path is implemented by the extended (p)NFSv4 server. When the file system being exported by (p)NFSv4 uses storage devices that are visible to clients over the network, the data path may be implemented by direct communication between the extended (p)NFSv4 file system client and the storage devices. This leads to a few new terms used to describe the protocol extension and some clarifications of existing terms.

2.1 Metadata Server

A pNFS "server" or "metadata server" is a server as defined by RFC3530 [2], which additionally provides support of the pNFS minor extension. When using the pNFS NFSv4 minor extension, the metadata server may hold only the metadata associated with a file, while the data can be stored on the storage devices. However, similar to NFSv4, data may also be written through the metadata server. Note: directory data is always accessed through the metadata server.

2.2 Client

A pNFS "client" is a client as defined by RFC3530 [2], with the addition of supporting the pNFS minor extension server protocol and with the addition of supporting at least one storage protocol for performing I/O directly to storage devices.

2.3 Storage Device

This is a device, or server, that controls the file's data, but leaves other metadata management up to the metadata server. A storage device could be another NFS server, or an Object Storage Device (OSD) or a block device accessed over a SAN (e.g., either FiberChannel or iSCSI SAN). The goal of this extension is to allow direct communication between clients and storage devices.

2.4 Storage Protocol

This is the protocol between the pNFS client and the storage device used to access the file data. Three following types have been described: file protocols (e.g., NFSv4), object protocols (e.g., OSD), and block/volume protocols (e.g., based on SCSI-block commands). These protocols are in turn realizable over a variety of transport stacks. We anticipate there will be variations on these storage protocols, including new protocols that are unknown at this time or experimental in nature. The details of the storage protocols will be described in other documents so that pNFS clients can be written to use these storage protocols. Use of NFSv4 itself as a file-based storage protocol is described in Section 5.

2.5 Control Protocol

This is a protocol used by the exported file system between the server and storage devices. Specification of such protocols is outside the scope of this draft. Such control protocols would be used to control such activities as the allocation and deallocation of storage and the management of state required by the storage devices to perform client access control. The control protocol should not be confused with protocols used to manage LUNs in a SAN and other sysadmin kinds of tasks.

While the pNFS protocol allows for any control protocol, in practice the control protocol is closely related to the storage protocol. For example, if the storage devices are NFS servers, then the protocol between the pNFS metadata server and the storage devices is likely to involve NFS operations. Similarly, when object storage devices are used, the pNFS metadata server will likely use iSCSI/OSD commands to manipulate storage.

However, this document does not mandate any particular control protocol. Instead, it just describes the requirements on the control protocol for maintaining attributes like modify time, the change attribute, and the end-of-file position.

2.6 Metadata

This is information about a file, like its name, owner, where it stored, and so forth. The information is managed by the exported file system server (metadata server). Metadata also includes lower-level information like block addresses and indirect block pointers. Depending the storage protocol, block-level metadata may or may not be managed by the metadata server, but is instead managed by Object Storage Devices or other servers acting as a storage device.

2.7 Layout

A layout defines how a file's data is organized on one or more storage devices. There are many possible layout types. They vary in the storage protocol used to access the data, and in the aggregation scheme that lays out the file data on the underlying storage devices. Layouts are described in more detail below.

3. pNFS protocol semantics

This section describes the semantics of the pNFS protocol extension to NFSv4; this is the protocol between the client and the metadata server.

3.1 Definitions

This sub-section defines a number of terms necessary for describing layouts and their semantics. In addition, it more precisely defines how layouts are identified and how they can be composed of smaller granularity layout segments.

3.1.1 Layout Types

A layout describes the mapping of a file's data to the storage devices that hold the data. A layout is said to belong to a specific "layout type" (see Section 6.1 for its RPC definition). The layout type allows for variants to handle different storage protocols (e.g., block/volume [7], object [8], and file [Section 5] layout types). A metadata server, along with its control protocol, must support at least one layout type. A private sub-range of the layout type name space is also defined. Values from the private layout type range can be used for internal testing or experimentation.

As an example, a file layout type could be an array of tuples (e.g., deviceID, file_handle), along with a definition of how the data is stored across the devices (e.g., striping). A block/volume layout might be an array of tuples that store <deviceID, block_number, block count> along with information about block size and the file offset of

the first block. An object layout might be an array of tuples <deviceID, objectID> and an additional structure (i.e., the aggregation map) that defines how the logical byte sequence of the file data is serialized into the different objects. Note, the actual layouts are more complex than these simple expository examples.

This document defines a NFSv4 file layout type using a stripe-based aggregation scheme (see Section 5). Adjunct specifications are being drafted that precisely define other layout formats (e.g., block/volume [7], and object [8] layouts) to allow interoperability among clients and metadata servers.

3.1.2 Layout Iomode

The iomode indicates to the metadata server the client's intent to perform either READs (only) or a mixture of I/O possibly containing WRITEs as well as READs (i.e., READ/WRITE). For certain layout types, it is useful for a client to specify this intent at LAYOUTGET time. E.g., for block/volume based protocols, block allocation could occur when a READ/WRITE iomode is specified. A special LAYOUTIOMODE_ANY iomode is defined and can only be used for LAYOUTRETURN and LAYOUTRECALL, not for LAYOUTGET. It specifies that layouts pertaining to both READ and RW iomodes are being returned or recalled, respectively.

A storage device may validate I/O with regards to the iomode; this is dependent upon storage device implementation. Thus, if the client's layout iomode differs from the I/O being performed the storage device may reject the client's I/O with an error indicating a new layout with the correct I/O mode should be fetched. E.g., if a client gets a layout with a READ iomode and performs a WRITE to a storage device, the storage device is allowed to reject that WRITE.

The iomode does not conflict with OPEN share modes or lock requests; open mode checks and lock enforcement are always enforced, and are logically separate from the pNFS layout level. As well, open modes and locks are the preferred method for restricting user access to data files. E.g., an OPEN of read, deny-write does not conflict with a LAYOUTGET containing an iomode of READ/WRITE performed by another client. Applications that depend on writing into the same file concurrently may use byte range locking to serialize their accesses.

3.1.3 Layout Segments

Until this point, layouts have been defined in a fairly vague manner. A layout is more precisely identified by the following tuple: <ClientID, FH, layout type>; the FH refers to the FH of the file on the metadata server. Note, layouts describe a file, not a byte-range

of a file.

Since a layout that describes an entire file may be very large, there is a desire to manage layouts in smaller chunks that correspond to byte-ranges of the file. For example, the entire layout need not be returned, recalled, or committed. These chunks are called "layout segments" and are further identified by the byte-range they represent. Layout operations require the identification of the layout segment (i.e., clientID, FH, layout type, and byte-range), as well as the iomode. This structure allows clients and metadata servers to aggregate the results of layout operations into a singly maintained layout.

It is important to define when layout segments overlap and/or conflict with each other. For a layout segment to overlap another layout segment both segments must be of the same layout type, correspond to the same filehandle, and have the same iomode; in addition, the byte-ranges of the segments must overlap. Layout segments conflict, when they overlap and differ in the content of the layout (i.e., the storage device/file mapping parameters differ). Note, differing iomodes do not lead to conflicting layouts. It is permissible for layout segments with different iomodes, pertaining to the same byte range, to be held by the same client.

3.1.4 Device IDs

The "deviceID" is a short name for a storage device. In practice, a significant amount of information may be required to fully identify a storage device. Instead of embedding all that information in a layout, a level of indirection is used. Layouts embed device IDs, and a new operation (GETDEVICEINFO) is used to retrieve the complete identity information about the storage device according to its layout type. For example, the identity of a file server or object server could be an IP address and port. The identity of a block device could be a volume label. Due to multipath connectivity in a SAN environment, agreement on a volume label is considered the reliable way to locate a particular storage device.

The device ID is qualified by the layout type and unique per file system (FSID). This allows different layout drivers to generate device IDs without the need for co-ordination. In addition to GETDEVICEINFO, another operation, GETDEVICELIST, has been added to allow clients to fetch the mappings of multiple storage devices attached to a metadata server.

Clients cannot expect the mapping between device ID and storage device address to persist across server reboots, hence a client MUST fetch new mappings on startup or upon detection of a metadata server

reboot unless it can revalidate its existing mappings. Not all layout types support such revalidation, and the means of doing so is layout specific. If data are reorganized from a storage device with a given device ID to a different storage device (i.e., if the mapping between storage device and data changes), the layout describing the data MUST be recalled rather than assigning the new storage device to the old device ID.

3.1.5 Aggregation Schemes

Aggregation schemes can describe layouts like simple one-to-one mapping, concatenation, and striping. A general aggregation scheme allows nested maps so that more complex layouts can be compactly described. The canonical aggregation type for this extension is striping, which allows a client to access storage devices in parallel. Even a one-to-one mapping is useful for a file server that wishes to distribute its load among a set of other file servers.

3.2 Guarantees Provided by Layouts

Layouts delegate to the client the ability to access data out of band. The layout guarantees the holder that the layout will be recalled when the state encapsulated by the layout becomes invalid (e.g., through some operation that directly or indirectly modifies the layout) or, possibly, when a conflicting layout is requested, as determined by the layout's iomode. When a layout is recalled, and then returned by the client, the client retains the ability to access file data with normal NFSv4 I/O operations through the metadata server. Only the right to do I/O out-of-band is affected.

Holding a layout does not guarantee that a user of the layout has the rights to access the data represented by the layout. All user access rights MUST be obtained through the appropriate open, lock, and access operations (i.e., those that would be used in the absence of pNFS). However, if a valid layout for a file is not held by the client, the storage device should reject all I/Os to that file's byte range that originate from that client. In summary, layouts and ordinary file access controls are independent. The act of modifying a file for which a layout is held, does not necessarily conflict with the holding of the layout that describes the file being modified. However, with certain layout types (e.g., block/volume layouts), the layout's iomode must agree with the type of I/O being performed.

Depending upon the layout type and storage protocol in use, storage device access permissions may be granted by LAYOUTGET and may be encoded within the type specific layout. If access permissions are encoded within the layout, the metadata server must recall the layout when those permissions become invalid for any reason; for example

when a file becomes unwritable or inaccessible to a client. Note, clients are still required to perform the appropriate access operations as described above (e.g., open and lock ops). The degree to which it is possible for the client to circumvent these access operations must be clearly addressed by the individual layout type documents, as well as the consequences of doing so. In addition, these documents must be clear about the requirements and non-requirements for the checking performed by the server.

If the pNFS metadata server supports mandatory byte range locks then byte range locks must behave as specified by the NFSv4 protocol, as observed by users of files. If a storage device is unable to restrict access by a pNFS client who does not hold a required mandatory byte range lock then the metadata server must not grant layouts to a client, for that storage device, that permits any access that conflicts with a mandatory byte range lock held by another client. In this scenario, it is also necessary for the metadata server to ensure that byte range locks are not granted to a client if any other client holds a conflicting layout; in this case all conflicting layouts must be recalled and returned before the lock request can be granted. This requires the pNFS server to understand the capabilities of its storage devices.

3.3 Getting a Layout

A client obtains a layout through a new operation, LAYOUTGET. The metadata server will give out layouts of a particular type (e.g., block/volume, object, or file) and aggregation as requested by the client. The client selects an appropriate layout type which the server supports and the client is prepared to use. The layout returned to the client may not line up exactly with the requested byte range. A field within the LAYOUTGET request, "minlength", specifies the minimum overlap that MUST exist between the requested layout and the layout returned by the metadata server. The "minlength" field should specify a size of at least one. A metadata server may give-out multiple overlapping, non-conflicting layout segments to the same client in response to a LAYOUTGET.

There is no implied ordering between getting a layout and performing a file OPEN. For example, a layout may first be retrieved by placing a LAYOUTGET operation in the same compound as the initial file OPEN. Once the layout has been retrieved, it can be held across multiple OPEN and CLOSE sequences.

The storage protocol used by the client to access the data on the storage device is determined by the layout's type. The client needs to select a "layout driver" that understands how to interpret and use that layout. The API used by the client to talk to its drivers is

outside the scope of the pNFS extension. The storage protocol between the client's layout driver and the actual storage is covered by other protocols specifications such as iSCSI (block storage), OSD (object storage) or NFS (file storage).

Although, the metadata server is in control of the layout for a file, the pNFS client can provide hints to the server when a file is opened or created about preferred layout type and aggregation scheme. The pNFS extension introduces a LAYOUT_HINT attribute that the client can set at creation time to provide a hint to the server for new files. It is suggested that this attribute be set as one of the initial attributes to OPEN when creating a new file. Setting this attribute separately, after the file has been created could make it difficult, or impossible, for the server implementation to comply.

3.4 Committing a Layout

Due to the nature of the protocol, the file attributes, and data location mapping (e.g., which offsets store data vs. store holes) that exist on the metadata storage device may become inconsistent in relation to the data stored on the storage devices; e.g., when WRITEs occur before a layout has been committed (e.g., between a LAYOUTGET and a LAYOUTCOMMIT). Thus, it is necessary to occasionally re-sync this state and make it visible to other clients through the metadata server.

The LAYOUTCOMMIT operation is responsible for committing a modified layout segment to the metadata server. Note: the data should be written and committed to the appropriate storage devices before the LAYOUTCOMMIT occurs. Note, if the data is being written asynchronously through the metadata server a COMMIT to the metadata server is required to sync the data and make it visible on the storage devices (see Section 3.6 for more details). The scope of this operation depends on the storage protocol in use. For block/volume-based layouts, it may require updating the block list that comprises the file and committing this layout to stable storage. While, for file-layouts it requires some synchronization of attributes between the metadata and storage devices (i.e., mainly the size attribute; EOF). It is important to note that the level of synchronization is from the point of view of the client who issued the LAYOUTCOMMIT. The updated state on the metadata server need only reflect the state as of the client's last operation previous to the LAYOUTCOMMIT, it need not reflect a globally synchronized state (e.g., other clients may be performing, or may have performed I/O since the client's last operation and the LAYOUTCOMMIT).

The control protocol is free to synchronize the attributes before it receives a LAYOUTCOMMIT, however upon successful completion of a

LAYOUTCOMMIT, state that exists on the metadata server that describes the file MUST be in sync with the state existing on the storage devices that comprise that file as of the issuing client's last operation. Thus, a client that queries the size of a file between a WRITE to a storage device and the LAYOUTCOMMIT may observe a size that does not reflects the actual data written.

3.4.1 LAYOUTCOMMIT and mtime/atime/change

The change attribute and the modify/access times may be updated, by the server, at LAYOUTCOMMIT time; since for some layout types, the change attribute and atime/mtime can not be updated by the appropriate I/O operation performed at a storage device. The arguments to LAYOUTCOMMIT allow the client to provide suggested access and modify time values to the server. Again, depending upon the layout type, these client provided values may or may not be used. The server should sanity check the client provided values before they are used. For example, the server should ensure that time does not flow backwards. According to the NFSv4 specification, The client always has the option to set these attributes through an explicit SETATTR operation.

As mentioned, for some layout protocols the change attribute and mtime/atime may be updated at or after the time the I/O occurred (e.g., if the storage device is able to communicate these attributes to the metadata server). If, upon receiving a LAYOUTCOMMIT, the server implementation is able to determine that the file did not change since the last time the change attribute was updated (e.g., no WRITEs or over-writes occurred), the implementation need not update the change attribute; file-based protocols may have enough state to make this determination or may update the change attribute upon each file modification. This also applies for mtime and atime; if the server implementation is able to determine that the file has not been modified since the last mtime update, the server need not update mtime at LAYOUTCOMMIT time. Once LAYOUTCOMMIT completes, the new change attribute and mtime/atime should be visible if that file was modified since the latest previous LAYOUTCOMMIT or LAYOUTGET.

3.4.2 LAYOUTCOMMIT and size

The file's size may be updated at LAYOUTCOMMIT time as well. The LAYOUTCOMMIT operation contains an argument that indicates the last byte offset to which the client wrote ("last_write_offset"). Note: for this offset to be viewed as a file size it must be incremented by one byte (e.g., a write to offset 0 would map into a file size of 1, but the last write offset is 0). The metadata server may do one of the following:

1. It may update the file's size based on the last write offset. However, to the extent possible, the metadata server should sanity check any value to which the file's size is going to be set. E.g., it must not truncate the file based on the client presenting a smaller last write offset than the file's current size.

2. If it has sufficient other knowledge of file size (e.g., by querying the storage devices through the control protocol), it may ignore the client provided argument and use the query-derived value.

3. It may use the last write offset as a hint, subject to correction when other information is available as above.

The method chosen to update the file's size will depend on the storage device's and/or the control protocol's implementation. For example, if the storage devices are block devices with no knowledge of file size, the metadata server must rely on the client to set the size appropriately. A new size flag and length are also returned in the results of a LAYOUTCOMMIT. This union indicates whether a new size was set, and to what length it was set. If a new size is set as a result of LAYOUTCOMMIT, then the metadata server must reply with the new size. As well, if the size is updated, the metadata server in conjunction with the control protocol SHOULD ensure that the new size is reflected by the storage devices immediately upon return of the LAYOUTCOMMIT operation; e.g., a READ up to the new file size should succeed on the storage devices (assuming no intervening truncations). Again, if the client wants to explicitly zero-extend or truncate a file, SETATTR must be used; it need not be used when simply writing past EOF.

Since client layout holders may be unaware of changes made to the file's size, through LAYOUTCOMMIT or SETATTR, by other clients, an additional callback/notification has been added for pNFS. CB_SIZECHANGED is a notification that the metadata server sends to layout holders to notify them of a change in file size. This is preferred over issuing CB_LAYOUTRECALL to each of the layout holders.

3.4.3 LAYOUTCOMMIT and layoutupdate

The LAYOUTCOMMIT operation contains a "layoutupdate" argument. This argument is a layout type specific structure. The structure can be used to pass arbitrary layout type specific information from the client to the metadata server at LAYOUTCOMMIT time. For example, if using a block/volume layout, the client can indicate to the metadata server which reserved or allocated blocks it used and which it did not. The "layoutupdate" structure need not be the same structure as

the layout returned by LAYOUTGET. The structure is defined by the layout type and is opaque to LAYOUTCOMMIT.

3.5 Recalling a Layout

3.5.1 Basic Operation

Since a layout protects a client's access to a file via a direct client-storage-device path, a layout need only be recalled when it is semantically unable to serve this function. Typically, this occurs when the layout no longer encapsulates the true location of the file over the byte range it represents. Any operation or action (e.g., server driven restriping or load balancing) that changes the layout will result in a recall of the layout. A layout is recalled by the CB_LAYOUTRECALL callback operation (see Section 10.1). This callback can either recall a layout segment identified by a byte range, or all the layouts associated with a file system (FSID). However, there is no single operation to return all layouts associated with an FSID; multiple layout segments may be returned in a single compound operation. Section 3.5.3 discusses sequencing issues surrounding the getting, returning, and recalling of layouts.

The iomode is also specified when recalling a layout or layout segment. Generally, the iomode in the recall request must match the layout, or segment, being returned; e.g., a recall with an iomode of RW should cause the client to only return RW layout segments (not R segments). However, a special LAYOUTIOMODE_ANY enumeration is defined to enable recalling a layout of any type (i.e., the client must return both read-only and read/write layouts).

A REMOVE operation may cause the metadata server to recall the layout to prevent the client from accessing a non-existent file and to reclaim state stored on the client. Since a REMOVE may be delayed until the last close of the file has occurred, the recall may also be delayed until this time. As well, once the file has been removed, after the last reference, the client SHOULD no longer be able to perform I/O using the layout (e.g., with file-based layouts an error such as ESTALE could be returned).

Although, the pNFS extension does not alter the caching capabilities of clients, or their semantics, it recognizes that some clients may perform more aggressive write-behind caching to optimize the benefits provided by pNFS. However, write-behind caching may impact the latency in returning a layout in response to a CB_LAYOUTRECALL; just as caching impacts DELEGRETURN with regards to data delegations. Client implementations should limit the amount of dirty data they have outstanding at any one time. Server implementations may fence clients from performing direct I/O to the storage devices if they

perceive that the client is taking too long to return a layout once recalled. A server may be able to monitor client progress by watching client I/Os or by observing LAYOUTRETURNs of sub-portions of the recalled layout. The server can also limit the amount of dirty data to be flushed to storage devices by limiting the byte ranges covered in the layouts it gives out.

Once a layout has been returned, the client MUST NOT issue I/Os to the storage devices for the file, byte range, and iomode represented by the returned layout. If a client does issue an I/O to a storage device for which it does not hold a layout, the storage device SHOULD reject the I/O.

3.5.2 Recall Callback Robustness

For simplicity, the discussion thus far has assumed that pNFS client state for a file exactly matches the pNFS server state for that file and client regarding layout ranges and permissions. This assumption leads to the implicit assumption that any callback results in a LAYOUTRETURN or set of LAYOUTRETURNs that exactly match the range in the callback, since both client and server agree about the state being maintained. However, it can be useful if this assumption does not always hold. For example:

o It may be useful for clients to be able to discard layout information without calling LAYOUTRETURN. If conflicts that require callbacks are very rare, and a server can use a multi-file callback to recover per-client resources (e.g., via a FSID recall, or a multi-file recall within a single compound), the result may be significantly less client-server pNFS traffic.

o 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.

o As well, 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. On the other hand, if a server believes that a client holds a layout segment that the client does not 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.

Thus, 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. A pNFS client must always return layout segments that comprise the full range specified by the recall. Note, the full recalled layout range need not be returned as part of a single operation, but may be returned in segments. This allows the client to stage the flushing of dirty data, layout commits, and returns. Also, it indicates to the metadata server that the client is making progress.

In order to ensure client/server convergence on the layout state, the final LAYOUTRETURN operation in a sequence of returns for a particular recall, SHOULD specify the entire range being recalled, even if layout segments pertaining to partial ranges were previously returned. In addition, if the client holds no layout segment that overlaps the range being recalled, the client should return the NFS4ERR_NOMATCHING_LAYOUT error code. This allows the server to update its view of the client's layout state.

3.5.3 Recall/Return Sequencing

As with other stateful operations, pNFS requires the correct sequencing of layout operations. This proposal assumes that sessions will precede or accompany pNFS into NFSv4.x and thus, pNFS will require the use of sessions. If the sessions proposal does not precede pNFS, then this proposal needs to be modified to provide for the correct sequencing of pNFS layout operations. Also, this specification is reliant on the sessions protocol to provide the correct sequencing between regular operations and callbacks. It is the server's responsibility to avoid inconsistencies regarding the layouts it hands out and the client's responsibility to properly serialize its layout requests.

One critical issue with operation sequencing concerns callbacks. The protocol must defend against races between the reply to a LAYOUTGET operation and a subsequent CB_LAYOUTRECALL. It MUST NOT be possible for a client to process the CB_LAYOUTRECALL for a layout that it has not received in a reply message to a LAYOUTGET.

3.5.3.1 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. This deadlock can be avoided by adhering to the following requirements:

- A LAYOUTGET MUST be rejected with an error (i.e., NFS4ERR_RECALLCONFLICT) if there's an overlapping outstanding recall callback to the same client

- When processing a recall, the client MUST wait for a response to all conflicting outstanding LAYOUTGETs before performing any RETURN that could be affected by any such response.

- The client SHOULD wait for responses to all operations required to complete a recall before sending any LAYOUTGETs that would conflict with the recall because the server is likely to return errors for them.

Now the client can wait for the LAYOUTGET response, as it will be received in both cases.

3.5.3.2 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.

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 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. Again, a client must take the appropriate precautions in serializing its actions.

[ASIDE: 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).]

3.6 Metadata Server Write Propagation

Asynchronous writes written through the metadata server may be propagated lazily to the storage devices. For data written asynchronously through the metadata server, a client performing a read at the appropriate storage device is not guaranteed to see the newly written data until a COMMIT occurs at the metadata server. While the write is pending, reads to the storage device can give out either the old data, the new data, or a mixture thereof. After either a synchronous write completes, or a COMMIT is received (for asynchronously written data), the metadata server must ensure that storage devices give out the new data and that the data has been written to stable storage. If the server implements its storage in any way such that it cannot obey these constraints, then it must recall the layouts to prevent reads being done that cannot be handled correctly.

3.7 Crash Recovery

Crash recovery is complicated due to the distributed nature of the pNFS protocol. In general, crash recovery for layouts is similar to crash recovery for delegations in the base NFSv4 protocol. However, the client's ability to perform I/O without contacting the metadata server introduces subtleties that must be handled correctly if file system corruption is to be avoided.

3.7.1 Leases

The layout lease period plays a critical role in crash recovery. Depending on the capabilities of the storage protocol, it is crucial that the client is able to maintain an accurate layout lease timer to

ensure that I/Os are not issued to storage devices after expiration of the layout lease period. In order for the client to do so, it must know which operations renew a lease.

3.7.1.1 Lease Renewal

The current NFSv4 specification allows for implicit lease renewals to occur upon receiving an I/O. However, due to the distributed pNFS architecture, implicit lease renewals are limited to operations performed at the metadata server; this includes I/O performed through the metadata server. So, a client must not assume that READ and WRITE I/O to storage devices implicitly renew lease state.

If sessions are required for pNFS, as has been suggested, then the SEQUENCE operation is to be used to explicitly renew leases. It is proposed that the SEQUENCE operation be extended to return all the specific information that RENEW does, but not as an error as RENEW returns it. Since, when using session, beginning each compound with the SEQUENCE op allows renews to be performed without an additional operation and without an additional request. Again, the client must not rely on any operation to the storage devices to renew a lease. Using the SEQUENCE operation for renewals, simplifies the client's perception of lease renewal.

3.7.1.2 Client Lease Timer

Depending on the storage protocol and layout type in use, it may be crucial that the client not issue I/Os to storage devices if the corresponding layout's lease has expired. Doing so may lead to file system corruption if the layout has been given out and used by another client. In order to prevent this, the client must maintain an accurate lease timer for all layouts held. RFC3530 has the following to say regarding the maintenance of a client lease timer:

> ...the client must track operations which will renew the lease period. Using the time that each such request was sent and the time that the corresponding reply was received, the client should bound the time that the corresponding renewal could have occurred on the server and thus determine if it is possible that a lease period expiration could have occurred.

To be conservative, the client should start its lease timer based on the time that the it issued the operation to the metadata server, rather than based on the time of the response.

It is also necessary to take propagation delay into account when requesting a renewal of the lease:

...the client should subtract it from lease times (e.g., if the client estimates the one-way propagation delay as 200 msec, then it can assume that the lease is already 200 msec old when it gets it). In addition, it will take another 200 msec to get a response back to the server. So the client must send a lock renewal or write data back to the server 400 msec before the lease would expire.

Thus, the client must be aware of the one-way propagation delay and should issue renewals well in advance of lease expiration. Clients, to the extent possible, should try not to issue I/Os that may extend past the lease expiration time period. However, since this is not always possible, the storage protocol must be able to protect against the effects of inflight I/Os, as is discussed later.

3.7.2 Client Recovery

Client recovery for layouts works in much the same way as NFSv4 client recovery works for other lock/delegation state. When an NFSv4 client reboots, it will lose all information about the layouts that it previously owned. There are two methods by which the server can reclaim these resources and allow otherwise conflicting layouts to be provided to other clients.

The first is through the expiry of the client's lease. If the client recovery time is longer than the lease period, the client's lease will expire and the server will know that state may be released. for layouts the server may release the state immediately upon lease expiry or it may allow the layout to persist awaiting possible lease revival, as long as there are no conflicting requests.

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 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 release all the state associated with the client's previous invocation.

3.7.3 Metadata Server Recovery

The server recovery case is slightly more complex. In general, the recovery process again follows the standard NFSv4 recovery model: the client will discover that the metadata 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, layouts are not reclaimable in the same sense as data delegations;

there is no reclaim bit, thus no guarantee of continuity between the previous and new layout. This is not necessarily required since a layout is not required to perform I/O; I/O can always be performed through the metadata server.

[NOTE: there is no reclaim bit for getting a layout. Thus, in the case of reclaiming an old layout obtained through LAYOUTGET, there is no guarantee of continuity. If a reclaim bit existed a block/volume layout type might be happier knowing it got the layout back with the assurance of continuity. However, this would require the metadata server trusting the client in telling it the exact layout it had (i.e., the full block-list); however, divergence is avoided by having the server tell the client what is contained within the layout.]

If the client has dirty data that it needs to write out, or an outstanding LAYOUTCOMMIT, the client should try to obtain a new layout segment covering the byte range covered by the previous layout segment. However, the client might not not get the same layout segment it had. The range might be different or it might get the same range but the content of the layout might be different. For example, if using a block/volume-based layout, the blocks provisionally assigned by the layout might be different, in which case the client will have to write the corresponding blocks again; in the interest of simplicity, the client might decide to always write them again. Alternatively, the client might be unable to obtain a new layout and thus, must write the data using normal NFSv4 through the metadata server.

There is an important safety concern associated with layouts that does not come into play in the standard NFSv4 case. If a standard NFSv4 client makes use of a stale delegation, while reading, the consequence could be to deliver stale data to an application. If writing, using a stale delegation or a stale state stateid for an open or lock would result in the rejection of the client's write with the appropriate stale stateid error.

However, the pNFS layout 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 metadata. Thus, it is vitally important that the client discover that the metadata server has rebooted, and that the client stops using stale layouts before the metadata server gives them away to other clients. To ensure this, the client must be implemented so that layouts are never used to access the storage after the client's lease timer has expired. It is crucial that clients have precise knowledge of the lease periods of their layouts. For specific details on lease renewal and client lease timers, see Section 3.7.1.

The prohibition on using stale layouts applies to all layout related accesses, especially the flushing of dirty data to the storage devices. 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 until the server can be contacted and the layout can be officially recovered or reclaimed. However, this is only part of the solution. It is also necessary to deal with the consequences of I/Os already in flight.

The issue of the effects of I/Os started before lease expiration and possibly continuing through lease expiration is the responsibility of the data storage protocol and as such is layout type specific. There are two approaches the data storage protocol can take. The protocol may adopt a global solution which prevents all I/Os from being executed after the lease expiration and thus is safe against a client who issues I/Os after lease expiration. This is the preferred solution and the solution used by NFSv4 file based layouts (see Section 5.6); as well, the object storage device protocol allows storage to fence clients after lease expiration. Alternatively, the storage protocol may rely on proper client operation and only deal with the effects of lingering I/Os. These solutions may impact the client layout-driver, the metadata server layout-driver, and the control protocol.

3.7.4 Storage Device Recovery

Storage device crash recovery is mostly dependent upon the layout type in use. However, there are a few general techniques a client can use if it discovers a storage device has crashed while holding asynchronously written, non-committed, data. First and foremost, it is important to realize that the client is the only one who has the information necessary to recover asynchronously written data; since, it holds the dirty data and most probably nobody else does. Second, the best solution is for the client to err on the side or caution and attempt to re-write the dirty data through another path.

The client, rather than hold the asynchronously written data indefinitely, is encouraged to, and can make sure that the data is written by using other paths to that data. The client may write the data to the metadata server, either synchronously or asynchronously with a subsequent COMMIT. Once it does this, there is no need to wait for the original storage device. In the event that the data range to be committed is transferred to a different storage device, as indicated in a new layout, the client may write to that storage device. Once the data has been committed at that storage device, either through a synchronous write or through a commit to that storage device (e.g., through the NFSv4 COMMIT operation for the NFSv4 file layout), the client should consider the transfer of

responsibility for the data to the new server as strong evidence that this is the intended and most effective method for the client to get the data written. In either case, once the write is on stable storage (through either the storage device or metadata server), there is no need to continue either attempting to commit or attempting to synchronously write the data to the original storage device or wait for that storage device to become available. That storage device may never be visible to the client again.

This approach does have a "lingering write" problem, similar to regular NFSv4. Suppose a WRITE is issued to a storage device for which no response is received. The client breaks the connection, trying to re-establish a new one, and gets a recall of the layout. The client issues the I/O for the dirty data through an alternative path, for example, through the metadata server and it succeeds. The client then goes on to perform additional writes that all succeed. If at some time later, the original write to the storage device succeeds, data inconsistency could result. The same problem can occur in regular NFSv4. For example, a WRITE is held in a switch for some period of time while other writes are issued and replied to, if the original WRITE finally succeeds, the same issues can occur. However, this is solved by sessions in NFSv4.x.

4. Security Considerations

The pNFS extension partitions the NFSv4 file system protocol into two parts, the control path and the data path (i.e., storage protocol). The control path contains all the new operations described by this extension; all existing NFSv4 security mechanisms and features apply to the control path. The combination of components in a pNFS system (see Figure 1) is required to preserve the security properties of NFSv4 with respect to an entity accessing data via a client, including security countermeasures to defend against threats that NFSv4 provides defenses for in environments where these threats are considered significant.

In some cases, the security countermeasures for connections to storage devices may take the form of physical isolation or a recommendation not to use pNFS in an environment. For example, it is currently infeasible to provide confidentiality protection for some storage device access protocols to protect against eavesdropping; in environments where eavesdropping on such protocols is of sufficient concern to require countermeasures, physical isolation of the communication channel (e.g., via direct connection from client(s) to storage device(s)) and/or a decision to forego use of pNFS (e.g., and fall back to NFSv4) may be appropriate courses of action.

In full generality where communication with storage devices is

subject to the same threats as client-server communication, the protocols used for that communication need to provide security mechanisms comparable to those available via RPSEC_GSS for NFSv4. Many situations in which pNFS is likely to be used will not be subject to the overall threat profile for which NFSv4 is required to provide countermeasures.

pNFS implementations MUST NOT remove NFSv4's access controls. The combination of clients, storage devices, and the server are responsible for ensuring that all client to storage device file data access respects NFSv4 ACLs and file open modes. This entails performing both of these checks on every access in the client, the storage device, or both. If a pNFS configuration performs these checks only in the client, the risk of a misbehaving client obtaining unauthorized access is an important consideration in determining when it is appropriate to use such a pNFS configuration. Such configurations SHOULD NOT be used when client- only access checks do not provide sufficient assurance that NFSv4 access control is being applied correctly.

The following subsections describe security considerations specifically applicable to each of the three major storage device protocol types supported for pNFS.

[Requiring strict equivalence to NFSv4 security mechanisms is the wrong approach. Will need to lay down a set of statements that each protocol has to make starting with access check location/properties.]

4.1 File Layout Security

A NFSv4 file layout type is defined in Section 5; see Section 5.7 for additional security considerations and details. In summary, the NFSv4 file layout type requires that all I/O access checks MUST be performed by the storage devices, as defined by the NFSv4 specification. If another file layout type is being used, additional access checks may be required. But in all cases, the access control performed by the storage devices must be at least as strict as that specified by the NFSv4 protocol.

4.2 Object Layout Security

The object storage protocol MUST implement the security aspects described in version 1 of the T10 OSD protocol definition [6]. The remainder of this section gives an overview of the security mechanism described in that standard. The goal is to give the reader a basic understanding of the object security model. Any discrepancies between this text and the actual standard are obviously to be resolved in favor of the OSD standard.

The object storage protocol relies on a cryptographically secure capability to control accesses at the object storage devices. Capabilities are generated by the metadata server, returned to the client, and used by the client as described below to authenticate their requests to the Object Storage Device (OSD). Capabilities therefore achieve the required access and open mode checking. They allow the file server to define and check a policy (e.g., open mode) and the OSD to check and enforce that policy without knowing the details (e.g., user IDs and ACLs). Since capabilities are tied to layouts, and since they are used to enforce access control, the server should recall layouts and revoke capabilities when the file ACL or mode changes in order to signal the clients.

Each capability is specific to a particular object, an operation on that object, a byte range w/in the object, and has an explicit expiration time. The capabilities are signed with a secret key that is shared by the object storage devices (OSD) and the metadata managers. clients do not have device keys so they are unable to forge capabilities. The the following sketch of the algorithm should help the reader understand the basic model.

LAYOUTGET returns

```
  {CapKey = MAC<SecretKey>(CapArgs), CapArgs}
```

The client uses CapKey to sign all the requests it issues for that object using the respective CapArgs. In other words, the CapArgs appears in the request to the storage device, and that request is signed with the CapKey as follows:

```
  ReqMAC = MAC<CapKey>(Req, Nonceln)
```

The following is sent to the OSD: {CapArgs, Req, Nonceln, ReqMAC}. The OSD uses the SecretKey it shares with the metadata server to compare the ReqMAC the client sent with a locally computed

```
  MAC<MAC<SecretKey>(CapArgs)>(Req, Nonceln)
```

and if they match the OSD assumes that the capabilities came from an authentic metadata server and allows access to the object, as allowed by the CapArgs. Therefore, if the server LAYOUTGET reply, holding CapKey and CapArgs, is snooped by another client, it can be used to generate valid OSD requests (within the CapArgs access restriction).

To provide the required privacy requirements for the capabilities returned by LAYOUTGET, the GSS-API can be used, e.g. by using a session key known to the file server and to the client to encrypt the whole layout or parts of it. Two general ways to provide privacy in

the absence of GSS-API that are independent of NFSv4 are either an isolated network such as a VLAN or a secure channel provided by IPsec.

4.3 Block/Volume Layout Security

As typically used, block/volume protocols rely on clients to enforce file access checks since the storage devices are generally unaware of the files they are storing and in particular are unaware of which blocks belongs to which file. In such environments, the physical addresses of blocks are exported to pNFS clients via layouts. 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 access control is important and client-only access checks provide insufficient assurance of access control enforcement (e.g., there is concern about a malicious of malfunctioning client skipping the access checks) and where physical block addresses are exported to clients, the storage devices will generally be unable to compensate for these client deficiencies.

In such threat environments, block/volume protocols SHOULD NOT be used with pNFS, unless the storage device is able to implement the appropriate access checks, via use of virtualized block addresses, or other means. NFSv4 without pNFS or pNFS with a different type of storage protocol would be a more suitable means to access files in such environments. Storage-device/protocol-specific methods (e.g. LUN masking/mapping) may be available to prevent malicious or high-risk clients from directly accessing storage devices.

5. The NFSv4 File Layout Type

This section describes the semantics and format of NFSv4 file-based layouts.

5.1 File Striping and Data Access

The file layout type describes a method for striping data across multiple devices. The data for each stripe unit is stored within an NFSv4 file located on a particular storage device. The structures used to describe the stripe layout are as follows:

```
    enum stripetype4 {
           STRIPE_SPARSE = 1,
           STRIPE_DENSE = 2
    };

    struct nfsv4_file_layouthint {
            stripetype4            stripe_type;
            length4                stripe_unit;
            uint32_t               stripe_width;
    };

    struct nfsv4_file_layout {                   /* Per data stripe */
           pnfs_deviceid4          dev_id<>;
           nfs_fh4                 fh;
    };

    struct nfsv4_file_layouttype4 {              /* Per file */
           stripetype4             stripe_type;
           length4                 stripe_unit;
           length4                 file_size;
           nfsv4_file_layout       dev_list<>;
    };
```

The file layout specifies an ordered array of <deviceID, filehandle> tuples, as well as the stripe size, type of stripe layout (discussed a little later), and the file's current size as of LAYOUTGET time. The filehandle, "fh", identifies the file on a storage device identified by "dev_id", that holds a particular stripe of the file. The "dev_id" array can be used for multipathing and is discussed further in Section 5.1.3. The stripe width is determined by the stripe unit size multiplied by the number of devices in the dev_list. The stripe held by <dev_id, fh> is determined by that tuples position within the device list, "dev_list". For example, consider a dev_list consisting of the following <dev_id, fh> pairs:

<(1,0x12), (2,0x13), (1,0x15)> and stripe_unit = 32KB

The stripe width is 32KB * 3 devices = 96KB. The first entry specifies that on device 1 in the data file with filehandle 0x12 holds the first 32KB of data (and every 32KB stripe beginning where the file's offset % 96KB == 0).

Devices may be repeated multiple times within the device list array; this is shown where storage device 1 holds both the first and third stripe of data. Filehandles can only be repeated if a sparse stripe type is used. Data is striped across the devices in the order listed in the device list array in increments of the stripe size. A data file stored on a storage device MUST map to a single file as defined

by the metadata server; i.e., data from two files as viewed by the metadata server MUST NOT be stored within the same data file on any storage device.

The "stripe_type" field specifies how the data is laid out within the data file on a storage device. It allows for two different data layouts: sparse and dense or packed. The stripe type determines the calculation that must be made to map the client visible file offset to the offset within the data file located on the storage device.

The layout hint structure is described in more detail in Section 6.7. It is used, by the client, as by the FILE_LAYOUT_HINT attribute to specify the type of layout to be used for a newly created file.

5.1.1 Sparse and Dense Storage Device Data Layouts

The stripe_type field allows for two storage device data file representations. Example sparse and dense storage device data layouts are illustrated below:

```
 Sparse file-layout (stripe_unit = 4KB)
 ------------------

 Is represented by the following file layout on the storage devices:

     Offset  ID:0    ID:1   ID:2
     0       +--+    +--+   +--+                 +--+  indicates a
             |//|    |  |   |  |                 |//|  stripe that
     4KB     +--+    +--+   +--+                 +--+  contains data
             |  |    |//|   |  |
     8KB     +--+    +--+   +--+
             |  |    |  |   |//|
     12KB    +--+    +--+   +--+
             |//|    |  |   |  |
     16KB    +--+    +--+   +--+
             |  |    |//|   |  |
             +--+    +--+   +--+
```

The sparse file-layout has holes for the byte ranges not exported by that storage device. This allows clients to access data using the real offset into the file, regardless of the storage device's position within the stripe. However, if a client writes to one of the holes (e.g., offset 4-12KB on device 1), then an error MUST be returned by the storage device. This requires that the storage device have knowledge of the layout for each file.

When using a sparse layout, the offset into the storage device data file is the same as the offset into the main file.

```
    Dense/packed file-layout (stripe_unit = 4KB)
    ------------------------

    Is represented by the following file layout on the storage devices:

        Offset  ID:0    ID:1   ID:2
        0       +--+    +--+   +--+
                |//|    |//|   |//|
        4KB     +--+    +--+   +--+
                |//|    |//|   |//|
        8KB     +--+    +--+   +--+
                |//|    |//|   |//|
        12KB    +--+    +--+   +--+
                |//|    |//|   |//|
        16KB    +--+    +--+   +--+
                |//|    |//|   |//|
                +--+    +--+   +--+
```

The dense or packed file-layout does not leave holes on the storage devices. Each stripe unit is spread across the storage devices. As such, the storage devices need not know the file's layout since the client is allowed to write to any offset.

The calculation to determine the byte offset within the data file for dense storage device layouts is:

```
  stripe_width = stripe_unit * N; where N = |dev_list|
  dev_offset = floor(file_offset / stripe_width) * stripe_unit +
               file_offset % stripe_unit
```

Regardless of the storage device data file layout, the calculation to determine the index into the device array is the same:

```
  dev_idx = floor(file_offset / stripe_unit) mod N
```

Section 5.5 describe the semantics for dealing with reads to holes within the striped file. This is of particular concern, since each individual component stripe file (i.e., the component of the striped file that lives on a particular storage device) may be of different length. Thus, clients may experience 'short' reads when reading off the end of one of these component files.

5.1.2 Metadata and Storage Device Roles

In many cases, the metadata server and the storage device will be separate pieces of physical hardware. The specification text is written as if that were always case. However, it can be the case that the same physical hardware is used to implement both a metadata

and storage device and in this case, the specification text's references to these two entities are to be understood as referring to the same physical hardware implementing two distinct roles and it is important that it be clearly understood on behalf of which role the hardware is executing at any given time.

Two sub-cases can be distinguished. In the first sub-case, the same physical hardware is used to implement both a metadata and data server in which each role is addressed through a distinct network interface (e.g., IP addresses for the metadata server and storage device are distinct). As long as the storage device address is obtained from the layout and is distinct from the metadata server's address, using the device ID therein to obtain the appropriate storage device address, it is always clear, for any given request, to what role it is directed, based on the destination IP address.

However, it may also be the case that even though the metadata server and storage device are distinct from one client's point of view, the roles may be reversed according to another client's point of view. For example, in the cluster file system model a metadata server to one client, may be a storage device to another client. Thus, it is safer to always mark the filehandle so that operations addressed to storage devices can be distinguished.

The second sub-case is where both the metadata and storage device have the same network address. This requires us to make the distinction as to which role each request is directed, on a another basis. Since the network address is the same, the request is understood as being directed at one or the other, based on the filehandle of the first current filehandle value for the request. If the first current file handle is one derived from a layout (i.e., it is specified within the layout) (and it is recommended that these be distinguishable), then the request is to be considered as executed by a storage device. Otherwise, the operation is to be understood as executed by the metadata server.

If a current filehandle is set that is inconsistent with the role to which it is directed, then the error NFS4ERR_BADHANDLE should result. For example, if a request is directed at the storage device, because the first current handle is from a layout, any attempt to set the current filehandle to be a value not from a layout should be rejected. Similarly, if the first current file handle was for a value not from a layout, a subsequent attempt to set the current file handle to a value obtained from a layout should be rejected.

5.1.3 Device Multipathing

The NFSv4 file layout supports multipathing to 'equivalent' devices.

Device-level multipathing is primarily of use in the case of a data server failure --- it allows the client to switch to another storage device that is exporting the same data stripe, without having to contact the metadata server for a new layout.

To support device multipathing, an array of device IDs is encoded within the data stripe portion of the file's layout. This array represents an ordered list of devices where the first element has the highest priority. Each device in the list MUST be 'equivalent' to every other device in the list and each device must be attempted in the order specified.

Equivalent devices MUST export the same system image (e.g., the stateids and filehandles that they use are the same) and must provide the same consistency guarantees. Two equivalent storage devices must also have sufficient connections to the storage, such that writing to one storage device is equivalent to writing to another, this also applies to reading. Also, if multiple copies of the same data exist, reading from one must provide access to all existing copies. As such, it is unlikely that multipathing will provide additional benefit in the case of an I/O error.

[NOTE: the error cases in which a client is expected to attempt an equivalent storage device should be specified.]

5.1.4 Operations Issued to Storage Devices

Clients MUST use the filehandle described within the layout when accessing data on the storage devices. When using the layout's filehandle, the client MUST only issue READ, WRITE, PUTFH, COMMIT, and NULL operations to the storage device associated with that filehandle. If a client issues an operation other than those specified above, using the filehandle and storage device listed in the client's layout, that storage device SHOULD return an error to the client. The client MUST follow the instruction implied by the layout (i.e., which filehandles to use on which devices). As described in Section 3.2, a client MUST NOT issue I/Os to storage devices for which it does not hold a valid layout. The storage devices may reject such requests.

GETATTR and SETATTR MUST be directed to the metadata server. In the case of a SETATTR of the size attribute, the control protocol is responsible for propagating size updates/truncations to the storage devices. In the case of extending WRITEs to the storage devices, the new size must be visible on the metadata server once a LAYOUTCOMMIT has completed (see Section 3.4.2). Section 5.5, describes the mechanism by which the client is to handle storage device file's that do not reflect the metadata server's size.

5.2 Global Stateid Requirements

Note, there are no stateids returned embedded within the layout. The client MUST use the stateid representing open or lock state as returned by an earlier metadata operation (e.g., OPEN, LOCK), or a special stateid to perform I/O on the storage devices, as in regular NFSv4. Special stateid usage for I/O is subject to the NFSv4 protocol specification. The stateid used for I/O MUST have the same effect and be subject to the same validation on storage device as it would if the I/O was being performed on the metadata server itself in the absence of pNFS. This has the implication that stateids are globally valid on both the metadata and storage devices. This requires the metadata server to propagate changes in lock and open state to the storage devices, so that the storage devices can validate I/O accesses. This is discussed further in Section 5.4. Depending on when stateids are propagated, the existence of a valid stateid on the storage device may act as proof of a valid layout.

[NOTE: a number of proposals have been made that have the possibility of limiting the amount of validation performed by the storage device, if any of these proposals are accepted or obtain consensus, the global stateid requirement can be revisited.]

5.3 The Layout Iomode

The layout iomode need not used by the metadata server when servicing NFSv4 file-based layouts, although in some circumstances it may be useful to use. For example, if the server implementation supports reading from read-only replicas or mirrors, it would be useful for the server to return a layout enabling the client to do so. As such, the client should set the iomode based on its intent to read or write the data. The client may default to an iomode of READ/WRITE (LAYOUTIOMODE_RW). The iomode need not be checked by the storage devices when clients perform I/O. However, the storage devices SHOULD still validate that the client holds a valid layout and return an error if the client does not.

5.4 Storage Device State Propagation

Since the metadata server, which handles lock and open-mode state changes, as well as ACLs, may not be collocated with the storage devices where I/O access are validated, as such, the server implementation MUST take care of propagating changes of this state to the storage devices. Once the propagation to the storage devices is complete, the full effect of those changes must be in effect at the storage devices. However, some state changes need not be propagated immediately, although all changes SHOULD be propagated promptly. These state propagations have an impact on the design of the control

protocol, even though the control protocol is outside of the scope of this specification. Immediate propagation refers to the synchronous propagation of state from the metadata server to the storage device(s); the propagation must be complete before returning to the client.

5.4.1 Lock State Propagation

Mandatory locks MUST be made effective at the storage devices before the request that establishes them returns to the caller. Thus, mandatory lock state MUST be synchronously propagated to the storage devices. On the other hand, since advisory lock state is not used for checking I/O accesses at the storage devices, there is no semantic reason for propagating advisory lock state to the storage devices. However, since all lock, unlock, open downgrades and upgrades affect the sequence ID stored within the stateid, the stateid changes which may cause difficulty if this state is not propagated. Thus, when a client uses a stateid on a storage device for I/O with a newer sequence number than the one the storage device has, the storage device should query the metadata server and get any pending updates to that stateid. This allows stateid sequence number changes to be propagated lazily, on-demand.

[NOTE: With the reliance on the sessions protocol, there is no real need for sequence ID portion of the stateid to be validated on I/O accesses. It is proposed that the seq. ID checking is obsoleted.]

Since updates to advisory locks neither confer nor remove privileges, these changes need not be propagated immediately, and may not need to be propagated promptly. The updates to advisory locks need only be propagated when the storage device needs to resolve a question about a stateid. In fact, if byte-range locking is not mandatory (i.e., is advisory) the clients are advised not to use the lock-based stateids for I/O at all. The stateids returned by open are sufficient and eliminate overhead for this kind of state propagation.

5.4.2 Open-mode Validation

Open-mode validation MUST be performed against the open mode(s) held by the storage devices. However, the server implementation may not always require the immediate propagation of changes. Reduction in access because of CLOSEs or DOWNGRADEs do not have to be propagated immediately, but SHOULD be propagated promptly; whereas changes due to revocation MUST be propagated immediately. On the other hand, changes that expand access (e.g., new OPEN's and upgrades) don't have to be propagated immediately but the storage device SHOULD NOT reject a request because of mode issues without making sure that the upgrade is not in flight.

5.4.3 File Attributes

Since the SETATTR operation has the ability to modify state that is visible on both the metadata and storage devices (e.g., the size), care must be taken to ensure that the resultant state across the set of storage devices is consistent; especially when truncating or growing the file.

As described earlier, the LAYOUTCOMMIT operation is used to ensure that the metadata is synced with changes made to the storage devices. For the file-based protocol, it is necessary to re-sync state such as the size attribute, and the setting of mtime/atime. See Section 3.4 for a full description of the semantics regarding LAYOUTCOMMIT and attribute synchronization. It should be noted, that by using a file-based layout type, it is possible to synchronize this state before LAYOUTCOMMIT occurs. For example, the control protocol can be used to query the attributes present on the storage devices.

Any changes to file attributes that control authorization or access as reflected by ACCESS calls or READs and WRITEs on the metadata server, MUST be propagated to the storage devices for enforcement on READ and WRITE I/O calls. If the changes made on the metadata server result in more restrictive access permissions for any user, those changes MUST be propagated to the storage devices synchronously.

Recall that the NFSv4 protocol [2] specifies that:

> ...since the NFS version 4 protocol does not impose any requirement that READs and WRITEs issued for an open file have the same credentials as the OPEN itself, the server still must do appropriate access checking on the READs and WRITEs themselves.

This also includes changes to ACLs. The propagation of access right changes due to changes in ACLs may be asynchronous only if the server implementation is able to determine that the updated ACL is not more restrictive for any user specified in the old ACL. Due to the relative infrequency of ACL updates, it is suggested that all changes be propagated synchronously.

[NOTE: it has been suggested that the NFSv4 specification is in error with regard to allowing principles other than those used for OPEN to be used for file I/O. If changes within a minor version alter the behavior of NFSv4 with regard to OPEN principals and stateids some access control checking at the storage device can be made less expensive. pNFS should be altered to take full advantage of these changes.]

5.5 Storage Device Component File Size

A potential problem exists when a component data file on a particular storage device is grown past EOF; the problem exists for both dense and sparse layouts. Imagine the following scenario: a client creates a new file (size == 0) and writes to byte 128KB; the client then seeks to the beginning of the file and reads byte 100. The client should receive 0s back as a result of the read. However, if the read falls on a different storage device to the client's original write, the storage device servicing the READ may still believe that the file's size is at 0 and return no data with the EOF flag set. The storage device can only return 0s if it knows that the file's size has been extended. This would require the immediate propagation of the file's size to all storage devices, which is potentially very costly, instead, another approach as outlined below.

First, the file's size is returned within the layout by LAYOUTGET. This size must reflect the latest size at the metadata server as set by the most recent of either the last LAYOUTCOMMIT or SETATTR; however, it may be more recent. Second, if a client performs a read that is returned short (i.e., is fully within the file's size, but the storage device indicates EOF and returns partial or no data), the client must assume that it is a hole and substitute 0s for the data not read up until its known local file size. If a client extends the file, it must update its local file size. Third, if the metadata server receives a SETATTR of the size or a LAYOUTCOMMIT that alters the file's size, the metadata server must send out CB_SIZECHANGED messages with the new size to clients holding layouts; it need not send a notification to the client that performed the operation that resulted in the size changing). Upon reception of the CB_SIZECHANGED notification, clients must update their local size for that file. As well, if a new file size is returned as a result to LAYOUTCOMMIT, the client must update their local file size.

5.6 Crash Recovery Considerations

As described in Section 3.7, the layout type specific storage protocol is responsible for handling the effects of I/Os started before lease expiration, extending through lease expiration. The NFSv4 file layout type prevents all I/Os from being executed after lease expiration, without relying on a precise client lease timer and without requiring storage devices to maintain lease timers.

It works as follows. In the presence of sessions, each compound begins with a SEQUENCE operation that contains the "clientID". On the storage device, the clientID can be used to validate that the client has a valid layout for the I/O being performed, if it does not, the I/O is rejected. Before the metadata server takes any

action to invalidate a layout given out by a previous instance, it must make sure that all layouts from that previous instance are invalidated at the storage devices. Note: it is sufficient to invalidate the stateids associated with the layout only if special stateids are not being used for I/O at the storage devices, otherwise the layout itself must be invalidated.

This means that a metadata server may not restripe a file until it has contacted all of the storage devices to invalidate the layouts from the previous instance nor may it give out locks that conflict with locks embodied by the stateids associated with any layout from the previous instance without either doing a specific invalidation (as it would have to do anyway) or doing a global storage device invalidation.

5.7 Security Considerations

The NFSv4 file layout type MUST adhere to the security considerations outlined in Section 4. More specifically, storage devices must make all of the required access checks on each READ or WRITE I/O as determined by the NFSv4 protocol [2]. This impacts the control protocol and the propagation of state from the metadata server to the storage devices; see Section 5.4 for more details.

5.8 Alternate Approaches

Two alternate approaches exist for file-based layouts and the method used by clients to obtain stateids used for I/O. Both approaches embed stateids within the layout.

However, before examining these approaches it is important to understand the distinction between clients and owners. Delegations belong to clients, while locks (e.g., record and share reservations) are held by owners which in turn belong to a specific client. As such, delegations can only protect against inter-client conflicts, not intra-client conflicts. Layouts are held by clients and SHOULD NOT be associated with state held by owners. Therefore, if stateids used for data access are embedded within a layout, these stateids can only act as delegation stateids, protecting against inter-client conflicts; stateids pertaining to an owner can not be embedded within the layout. This has the implication that the client MUST arbitrate among all intra-client conflicts (e.g., arbitrating among lock requests by different processes) before issuing pNFS operations. Using the stateids stored within the layout, storage devices can only arbitrate between clients (not owners).

The first alternate approach is to do away with global stateids, stateids returned by OPEN/LOCK that are valid on the metadata server

and storage devices, and use only stateids embedded within the layout. This approach has the drawback that the stateids used for I/O access can not be validated against per owner state, since they are only associated with the client holding the layout. It breaks the semantics of tieing a stateid used for I/O to an open instance. This has the implication that clients must delegate per owner lock and open requests internally, rather than push the work onto the storage devices. The storage devices can still arbitrate and enforce inter-client lock and open state.

The second approach is a hybrid approach. This approach allows for stateids to be embedded with the layout, but also allows for the possibility of global stateids. If the stateid embedded within the layout is a special stateid of all zeros, then the stateid referring to the last successful OPEN/LOCK should be used. This approach is recommended if it is decided that using NFSv4 as a control protocol is required.

This proposal suggests the global stateid approach due to the cleaner semantics it provides regarding the relationship between stateids used for I/O and their corresponding open instance or lock state. However, it does have a profound impact on the control protocol's implementation and the state propagation that is required (as described in Section 5.4).

6. pNFS Typed Data Structures

6.1 pnfs_layouttype4

```
enum pnfs_layouttype4 {
        LAYOUT_NFSV4_FILES = 1,
        LAYOUT_OSD2_OBJECTS = 2,
        LAYOUT_BLOCK_VOLUME = 3
};
```

A layout type specifies the layout being used. The implication is that clients have "layout drivers" that support one or more layout types. The file server advertises the layout types it supports through the LAYOUT_TYPES file system attribute. A client asks for layouts of a particular type in LAYOUTGET, and passes those layouts to its layout driver. The set of well known layout types must be defined. As well, a private range of layout types is to be defined by this document. This would allow custom installations to introduce new layout types.

[OPEN ISSUE: Determine private range of layout types]

New layout types must be specified in RFCs approved by the IESG

before becoming part of the pNFS specification.

The LAYOUT_NFSV4_FILES enumeration specifies that the NFSv4 file layout type is to be used. The LAYOUT_OSD2_OBJECTS enumeration specifies that the object layout, as defined in [8], is to be used. Similarly, the LAYOUT_BLOCK_VOLUME enumeration that the block/volume layout, as defined in [7], is to be used.

6.2 pnfs_deviceid4

```
  typedef uint64_t pnfs_deviceid4;       /* 64-bit device ID */
```

Layout information includes device IDs that specify a storage device through a compact handle. Addressing and type information is obtained with the GETDEVICEINFO operation. A client must not assume that device IDs are valid across metadata server reboots. The device ID is qualified by the layout type and are unique per file system (FSID). This allows different layout drivers to generate device IDs without the need for co-ordination. See Section 3.1.4 for more details.

6.3 pnfs_deviceaddr4

```
  struct pnfs_netaddr4 {
           string           r_netid<>;   /* network ID */
           string           r_addr<>;    /* universal address */
  };

  union pnfs_deviceaddr4 switch (pnfs_layouttype4 layout_type) {
         case LAYOUT_NFSV4_FILES:
                pnfs_netaddr4    netaddr;
         default:
                opaque           device_addr<>; /* Other layouts */
  };
```

The device address is used to set up a communication channel with the storage device. Different layout types will require different types of structures to define how they communicate with storage devices. The union is switched on the layout type.

Currently, the only device address defined is that for the NFSv4 file layout, which identifies a storage device by network IP address and port number. This is sufficient for the clients to communicate with the NFSv4 storage devices, and may also be sufficient for object-based storage drivers to communicate with OSDs. The other device address we expect to support is a SCSI volume identifier. The final protocol specification will detail the allowed values for device_type and the format of their associated location information.

[NOTE: other device addresses will be added as the respective specifications mature. It has been suggested that a separate device_type enumeration is used as a switch to the pnfs_deviceaddr4 structure (e.g., if multiple types of addresses exist for the same layout type). Until such a time as a real case is made and the respective layout types have matured, the device address structure will be left as is.]

6.4 pnfs_devlist_item4

```
struct pnfs_devlist_item4 {
        pnfs_deviceid4          id;
        pnfs_deviceaddr4        addr;
};
```

An array of these values is returned by the GETDEVICELIST operation. They define the set of devices associated with a file system.

6.5 pnfs_layout4

```
union pnfs_layoutdata4 switch (pnfs_layouttype4 layout_type) {
        case LAYOUT_NFSV4_FILES:
                nfsv4_file_layouttype4 file_layout;
        default:
                opaque          layout_data<>;
};

struct pnfs_layout4 {
        offset4                 offset;
        length4                 length;
        pnfs_layoutiomode4      iomode;
        pnfs_layoutdata4        layout;
};
```

The pnfs_layout4 structure defines a layout for a file. The pnfs_layoutdata4 union contains the portion of the layout specific to the layout type. Currently, only the NFSv4 file layout type is defined; see Section 5.1 for its definition. Since layouts are sub-dividable, the offset and length together with the file's filehandle, the clientid, iomode, and layout type, identifies the layout.

[OPEN ISSUE: there is a discussion of moving the striping information, or more generally the "aggregation scheme", up to the generic layout level. This creates a two-layer system where the top level is a switch on different data placement layouts, and the next level down is a switch on different data storage types. This lets different layouts (e.g., striping or mirroring or redundant servers) to be layered over different storage devices. This would move

geometry information out of nfsv4_file_layouttype4 and up into a generic pnfs_striped_layout type that would specify a set of pnfs_deviceid4 and pnfs_devicetype4 to use for storage. Instead of nfsv4_file_layouttype4, there would be pnfs_nfsv4_devicetype4.]

6.6 pnfs_layoutupdate4

```
   union pnfs_layoutupdate4 switch (pnfs_layouttype4 layout_type) {
          case LAYOUT_NFSV4_FILES:
                 void;
          default:
                 opaque           layout_data<>;
   };
```

The pnfs_layoutupdate4 structure is used by the client to return 'updated' layout information to the metadata server at LAYOUTCOMMIT time. This provides a channel to pass layout type specific information back to the metadata server. E.g., for block/volume layout types this could include the list of reserved blocks that were written. The contents of the structure are determined by the layout type and are defined in their context.

6.7 pnfs_layouthint4

```
   union pnfs_layouthint4 switch (pnfs_layouttype4 layout_type) {
          case LAYOUT_NFSV4_FILES:
                 nfsv4_file_layouthint layout_hint;
          default:
                 opaque                layout_hint_data<>;
   };
```

The pnfs_layouthint4 structure is used by the client to pass in a hint about the type of layout it would like created for a particular file. It is the structure specified by the FILE_LAYOUT_HINT attribute described below. The metadata server may ignore the hint, or may selectively ignore fields within the hint. This hint should be provided at create time as part of the initial attributes within OPEN. The "nfsv4_file_layouthint" structure is defined in Section 5.1.

6.8 pnfs_layoutiomode4

```
   enum pnfs_layoutiomode4 {
           LAYOUTIOMODE_READ          = 1,
           LAYOUTIOMODE_RW            = 2,
           LAYOUTIOMODE_ANY           = 3
   };
```

The iomode specifies whether the client intends to read or write (with the possibility of reading) the data represented by the layout. The ANY iomode MUST NOT be used for LAYOUTGET, however, it can be used for LAYOUTRETURN and LAYOUTRECALL. The ANY iomode specifies that layouts pertaining to both READ and RW iomodes are being returned or recalled, respectively. The metadata server's use of the iomode may depend on the layout type being used. The storage devices may validate I/O accesses against the iomode and reject invalid accesses.

7. pNFS File Attributes

7.1 pnfs_layouttype4<> FS_LAYOUT_TYPES

This attribute applies to a file system and indicates what layout types are supported by the file system. We expect this attribute to be queried when a client encounters a new fsid. This attribute is used by the client to determine if it has applicable layout drivers.

7.2 pnfs_layouttype4<> FILE_LAYOUT_TYPES

This attribute indicates the particular layout type(s) used for a file. This is for informational purposes only. The client needs to use the LAYOUTGET operation in order to get enough information (e.g., specific device information) in order to perform I/O.

7.3 pnfs_layouthint4 FILE_LAYOUT_HINT

This attribute may be set on newly created files to influence the metadata server's choice for the file's layout. It is suggested that this attribute is set as one of the initial attributes within the OPEN call. The metadata server may ignore this attribute. This attribute is a sub-set of the layout structure returned by LAYOUTGET. For example, instead of specifying particular devices, this would be used to suggest the stripe width of a file. It is up to the server implementation to determine which fields within the layout it uses.

[OPEN ISSUE: it has been suggested that the HINT is a well defined type other than pnfs_layoutdata4, similar to pnfs_layoutupdate4.]

7.4 uint32_t FS_LAYOUT_PREFERRED_BLOCKSIZE

This attribute is a file system wide attribute and indicates the preferred block size for direct storage device access.

7.5 uint32_t FS_LAYOUT_PREFERRED_ALIGNMENT

This attribute is a file system wide attribute and indicates the

preferred alignment for direct storage device access.

8. pNFS Error Definitions

NFS4ERR_BADLAYOUT Layout specified is invalid.

NFS4ERR_BADIOMODE Layout iomode is invalid.

NFS4ERR_LAYOUTUNAVAILABLE Layouts are not available for the file or its containing file system.

NFS4ERR_LAYOUTTRYLATER Layouts are temporarily unavailable for the file, client should retry later.

NFS4ERR_NOMATCHING_LAYOUT Client has no matching layout (segment) to return.

NFS4ERR_RECALLCONFLICT Layout is unavailable due to a conflicting LAYOUTRECALL that is in progress.

NFS4ERR_UNKNOWN_LAYOUTTYPE Layout type is unknown.

9. pNFS Operations

9.1 LAYOUTGET - Get Layout Information

SYNOPSIS

```
  (cfh), clientid, layout_type, iomode, offset, length,
  minlength, maxcount -> layout
```

ARGUMENT

```
  struct LAYOUTGET4args {
          /* CURRENT_FH: file */
          clientid4               clientid;
          pnfs_layouttype4        layout_type;
          pnfs_layoutiomode4      iomode;
          offset4                 offset;
          length4                 length;
          length4                 minlength;
          count4                  maxcount;
  };
```

RESULT

```
  struct LAYOUTGET4resok {
          pnfs_layout4            layout;
  };

  union LAYOUTGET4res switch (nfsstat4 status) {
          case NFS4_OK:
                  LAYOUTGET4resok resok4;
          default:
                  void;
  };
```

DESCRIPTION

Requests a layout for reading or writing (and reading) the file given by the filehandle at the byte range specified by offset and length. Layouts are identified by the clientid, filehandle, and layout type. The use of the iomode depends upon the layout type, but should reflect the client's data access intent.

The LAYOUTGET operation returns layout information for the specified byte range, a layout segment. To get a layout segment from a specific offset through the end-of-file, regardless of the file's length, a length field with all bits set to 1 (one) should be used. If the length is zero, or if a length which is not all bits set to one is specified, and length when added to the offset exceeds the maximum 64-bit unsigned integer value, the error NFS4ERR_INVAL will

result.

The "minlength" field specifies the minimum size overlap with the requested offset and length that is to be returned. If this requirement cannot be met, no layout must be returned; the error NFS4ERR_LAYOUTTRYLATER can be returned.

The "maxcount" field specifies the maximum layout size (in bytes) that the client can handle. If the size of the layout structure exceeds the size specified by maxcount, the metadata server will return the NFS4ERR_TOOSMALL error.

As well, the metadata server may adjust the range of the returned layout segment based on striping patterns and usage implied by the iomode. The client must be prepared to get a layout that does not line up exactly with their request; there MUST be at least an overlap of "minlength" between the layout returned by the server and the client's request, or the server SHOULD reject the request. See Section 3.3 for more details.

The metadata server may also return a layout segment with an iomode other than that requested by the client. If it does so, it must ensure that the iomode is more permissive than the iomode requested. E.g., this allows an implementation to upgrade read-only requests to read/write requests at its discretion, within the limits of the layout type specific protocol. An iomode of either LAYOUTIOMODE_READ or LAYOUTIOMODE_RW must be returned.

The format of the returned layout is specific to the underlying file system. Layout types other than the NFSv4 file layout type should be specified outside of this document.

If layouts are not supported for the requested file or its containing file system the server SHOULD return NFS4ERR_LAYOUTUNAVAILABLE. If the layout type is not supported, the metadata server should return NFS4ERR_UNKNOWN_LAYOUTTYPE. If layouts are supported but no layout matches the client provided layout identification, the server should return NFS4ERR_BADLAYOUT. If an invalid iomode is specified, or an iomode of LAYOUTIOMODE_ANY is specified, the server should return NFS4ERR_BADIOMODE.

If the layout for the file is unavailable due to transient conditions, e.g. file sharing prohibits layouts, the server must return NFS4ERR_LAYOUTTRYLATER.

If the layout request is rejected due to an overlapping layout recall, the server must return NFS4ERR_RECALLCONFLICT. See Section 3.5.3 for details.

If the layout conflicts with a mandatory byte range lock held on the file, and if the storage devices have no method of enforcing mandatory locks, other than through the restriction of layouts, the metadata server should return NFS4ERR_LOCKED.

On success, the current filehandle retains its value.

IMPLEMENTATION

Typically, LAYOUTGET will be called as part of a compound RPC after an OPEN operation and results in the client having location information for the file; a client may also hold a layout across multiple OPENs. The client specifies a layout type that limits what kind of layout the server will return. This prevents servers from issuing layouts that are unusable by the client.

ERRORS

NFS4ERR_BADLAYOUT
NFS4ERR_BADIOMODE
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_LAYOUTUNAVAILABLE
NFS4ERR_LAYOUTTRYLATER
NFS4ERR_LOCKED
NFS4ERR_NOFILEHANDLE
NFS4ERR_NOTSUPP
NFS4ERR_RECALLCONFLICT
NFS4ERR_STALE
NFS4ERR_STALE_CLIENTID
NFS4ERR_TOOSMALL
NFS4ERR_UNKNOWN_LAYOUTTYPE

9.2 LAYOUTCOMMIT - Commit writes made using a layout

SYNOPSIS

```
     (cfh), clientid, offset, length, last_write_offset,
     time_modify, time_access, layoutupdate -> newsize
```

ARGUMENT

```
     union newtime4 switch (bool timechanged) {
             case TRUE:
                     nfstime4           time;
             case FALSE:
                     void;
     };

     union newsize4 switch (bool sizechanged) {
             case TRUE:
                     length4            size;
             case FALSE:
                     void;
     };

     struct LAYOUTCOMMIT4args {
             /* CURRENT_FH: file */
             clientid4               clientid;
             offset4                 offset;
             length4                 length;
             length4                 last_write_offset;
             newtime4                time_modify;
             newtime4                time_access;
             pnfs_layoutupdate4      layoutupdate;
     };
```

RESULT

```
     struct LAYOUTCOMMIT4resok {
            newsize4                 newsize;
     };

     union LAYOUTCOMMIT4res switch (nfsstat4 status) {
             case NFS4_OK:
                     LAYOUTCOMMIT4resok  resok4;
             default:
                     void;
     };
```

DESCRIPTION

Commits changes in the layout segment represented by the current filehandle, clientid, and byte range. Since layouts are sub-dividable, a smaller portion of a layout, retrieved via LAYOUTGET, may be committed. The region being committed is specified through the byte range (length and offset). Note: the "layoutupdate" structure does not include the length and offset, as they are already specified in the arguments.

The LAYOUTCOMMIT operation indicates that the client has completed writes using a layout obtained by a previous LAYOUTGET. The client may have only written a subset of the data range it previously requested. LAYOUTCOMMIT allows it to commit or discard provisionally allocated space and to update the server with a new end of file. The layout referenced by LAYOUTCOMMIT is still valid after the operation completes and can be continued to be referenced by the clientid, filehandle, byte range, and layout type.

The "last_write_offset" field specifies the offset of the last byte written by the client previous to the LAYOUTCOMMIT. Note: this value is never equal to the file's size (at most it is one byte less than the file's size). The metadata server may use this information to determine whether the file's size needs to be updated. If the metadata server updates the file's size as the result of the LAYOUTCOMMIT operation, it must return the new size as part of the results.

The "time_modify" and "time_access" fields allow the client to suggest times it would like the metadata server to set. The metadata server may use these time values or it may use the time of the LAYOUTCOMMIT operation to set these time values. If the metadata server uses the client provided times, it should sanity check the values (e.g., to ensure time does not flow backwards). If the client wants to force the metadata server to set an exact time, the client should use a SETATTR operation in a compound right after LAYOUTCOMMIT. See Section 3.4 for more details. If the new client desires the resultant mtime or atime, it should issue a GETATTR following the LAYOUTCOMMIT; e.g., later in the same compound.

The "layoutupdate" argument to LAYOUTCOMMIT provides a mechanism for a client to provide layout specific updates to the metadata server. For example, the layout update can describe what regions of the original layout have been used and what regions can be deallocated. There is no NFSv4 file layout specific layoutupdate structure.

The layout information is more verbose for block devices than for objects and files because the latter hide the details of block allocation behind their storage protocols. At the minimum, the client needs to communicate changes to the end of file location back

to the server, and, if desired, its view of the file modify and access time. For block/volume layouts, it needs to specify precisely which blocks have been used.

If the layout identified in the arguments does not exist, the error NFS4ERR_BADLAYOUT is returned. The layout being committed may also be rejected if it does not correspond to an existing layout with an iomode of RW.

On success, the current filehandle retains its value.

ERRORS

```
   NFS4ERR_BADLAYOUT
   NFS4ERR_BADIOMODE
   NFS4ERR_FHEXPIRED
   NFS4ERR_INVAL
   NFS4ERR_NOFILEHANDLE
   NFS4ERR_STALE
   NFS4ERR_STALE_CLIENTID
   NFS4ERR_UNKNOWN_LAYOUTTYPE
```

9.3 LAYOUTRETURN - Release Layout Information

SYNOPSIS

```
  (cfh), clientid, offset, length, iomode, layout_type -> -
```

ARGUMENT

```
  struct LAYOUTRETURN4args {
          /* CURRENT_FH: file */
          clientid4               clientid;
          offset4                 offset;
          length4                 length;
          pnfs_layoutiomode4      iomode;
          pnfs_layouttype4        layout_type;
  };
```

RESULT

```
  struct LAYOUTRETURN4res {
          nfsstat4        status;
  };
```

DESCRIPTION

Returns the layout segment represented by the current filehandle, clientid, byte range, iomode, and layout type. After this call, the client MUST NOT use the layout and the associated storage protocol to access the file data. The layout being returned may be a subdivision of a layout previously fetched through LAYOUTGET. As well, it may be a subset or superset of a layout specified by CB_LAYOUTRECALL. However, if it is a subset, the recall is not complete until the full byte range has been returned. It is also permissible, and no error should result, for a client to return a byte range covering a layout it does not hold. If the length is all 1s, the layout covers the range from offset to EOF. An iomode of ANY specifies that all layouts that match the other arguments to LAYOUTRETURN (i.e., clientid, byte range, and type) are being returned.

Layouts may be returned when recalled or voluntarily (i.e., before the server has recalled them). In either case the client must properly propagate state changed under the context of the layout to storage or to the server before returning the layout.

If a client fails to return a layout in a timely manner, then the metadata server should use its control protocol with the storage devices to fence the client from accessing the data referenced by the layout. See Section 3.5 for more details.

If the layout identified in the arguments does not exist, the error NFS4ERR_BADLAYOUT is returned. If a layout exists, but the iomode does not match, NFS4ERR_BADIOMODE is returned.

On success, the current filehandle retains its value.

[OPEN ISSUE: Should LAYOUTRETURN be modified to handle FSID callbacks?]

ERRORS

NFS4ERR_BADLAYOUT
NFS4ERR_BADIOMODE
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_NOFILEHANDLE
NFS4ERR_STALE
NFS4ERR_STALE_CLIENTID
NFS4ERR_UNKNOWN_LAYOUTTYPE

9.4 GETDEVICEINFO - Get Device Information

SYNOPSIS

```
  (cfh), device_id, layout_type, maxcount -> device_addr
```

ARGUMENT

```
  struct GETDEVICEINFO4args {
          /* CURRENT_FH: file */
          pnfs_deviceid4                  device_id;
          pnfs_layouttype4                layout_type;
          count4                          maxcount;
  };
```

RESULT

```
  struct GETDEVICEINFO4resok {
          pnfs_deviceaddr4                device_addr;
  };

  union GETDEVICEINFO4res switch (nfsstat4 status) {
          case NFS4_OK:
                  GETDEVICEINFO4resok     resok4;
          default:
                  void;
  };
```

DESCRIPTION

Returns device type and device address information for a specified device. The returned device_addr includes a type that indicates how to interpret the addressing information for that device. The current filehandle (cfh) is used to identify the file system; device IDs are unique per file system (FSID) and are qualified by the layout type.

See Section 3.1.4 for more details on device ID assignment.

If the size of the device address exceeds maxcount bytes, the metadata server will return the error NFS4ERR_TOOSMALL. If an invalid device ID is given, the metadata server will respond with NFS4ERR_INVAL.

ERRORS

```
   NFS4ERR_FHEXPIRED
   NFS4ERR_INVAL
   NFS4ERR_TOOSMALL
```

```
       NFS4ERR_UNKNOWN_LAYOUTTYPE
```

9.5   GETDEVICELIST - Get List of Devices

SYNOPSIS

```
  (cfh), layout_type, maxcount, cookie, cookieverf ->
  cookie, cookieverf, device_addrs<>
```

ARGUMENT

```
  struct GETDEVICELIST4args {
          /* CURRENT_FH: file */
          pnfs_layouttype4                layout_type;
          count4                          maxcount;
          nfs_cookie4                     cookie;
          verifier4                       cookieverf;
  };
```

RESULT

```
  struct GETDEVICELIST4resok {
          nfs_cookie4                     cookie;
          verifier4                       cookieverf;
          pnfs_devlist_item4              device_addrs<>;
  };

  union GETDEVICELIST4res switch (nfsstat4 status) {
          case NFS4_OK:
                  GETDEVICELIST4resok     resok4;
          default:
                  void;
  };
```

DESCRIPTION

In some applications, especially SAN environments, it is convenient to find out about all the devices associated with a file system. This lets a client determine if it has access to these devices, e.g., at mount time.

This operation returns an array of items (pnfs_devlist_item4) that establish the association between the short pnfs_deviceid4 and the addressing information for that device, for a particular layout type. This operation may not be able to fetch all device information at once, thus it uses a cookie based approach, similar to READDIR, to fetch additional device information (see [2], section 14.2.24). As

in GETDEVICEINFO, the current filehandle (cfh) is used to identify the file system.

As in GETDEVICEINFO, maxcount specifies the maximum number of bytes to return. If the metadata server is unable to return a single device address, it will return the error NFS4ERR_TOOSMALL. If an invalid device ID is given, the metadata server will respond with NFS4ERR_INVAL.

ERRORS

NFS4ERR_BAD_COOKIE
NFS4ERR_FHEXPIRED
NFS4ERR_INVAL
NFS4ERR_TOOSMALL
NFS4ERR_UNKNOWN_LAYOUTTYPE

10. Callback Operations

10.1 CB_LAYOUTRECALL

SYNOPSIS

```
  layout_type, iomode, layoutrecall -> -
```

ARGUMENT

```
  enum layoutrecall_type4 {
          RECALL_FILE = 1,
          RECALL_FSID = 2
  };

  struct layoutrecall_file4 {
          nfs_fh4         fh;
          offset4         offset;
          length4         length;
  };

  union layoutrecall4 switch(layoutrecall_type4 recalltype) {
          case RECALL_FILE:
                  layoutrecall_file4 layout;
          case RECALL_FSID:
                  fsid4             fsid;
  };

  struct CB_LAYOUTRECALLargs {
          pnfs_layouttype4        layout_type;
          pnfs_layoutiomode4      iomode;
          layoutrecall4           layoutrecall;
  };
```

RESULT

```
  struct CB_LAYOUTRECALLres {
          nfsstat4        status;
  };
```

DESCRIPTION

The CB_LAYOUTRECALL operation is used to begin the process of recalling a layout, a portion thereof, or all layouts pertaining to a particular file system (FSID). If RECALL_FILE is specified, the offset and length fields specify the portion of the layout to be returned. The iomode specifies the set of layouts to be returned. An iomode of ANY specifies that all matching layouts, regardless of iomode, must be returned; otherwise, only layouts that exactly match the iomode must be returned.

If RECALL_FSID is specified, the fsid specifies the file system for which any outstanding layouts must be returned. Layouts are returned through the LAYOUTRETURN operation.

If the client does not hold any layout segment either matching or overlapping with the requested layout, it returns NFS4ERR_NOMATCHING_LAYOUT. If a length of all 1s is specified then the layout corresponding to the byte range from "offset" to the end-of-file MUST be returned.

IMPLEMENTATION

The client should reply to the callback immediately. Replying does not complete the recall except when an error is returned. The recall is not complete until the layout(s) are returned using a LAYOUTRETURN.

The client should complete any in-flight I/O operations using the recalled layout(s) before returning it/them via LAYOUTRETURN. If the client has buffered dirty data, it may chose to write it directly to storage before calling LAYOUTRETURN, or to write it later using normal NFSv4 WRITE operations to the metadata server.

If dirty data is flushed while the layout is held, the client must still issue LAYOUTCOMMIT operations at the appropriate time, especially before issuing the LAYOUTRETURN. If a large amount of dirty data is outstanding, the client may issue LAYOUTRETURNs for portions of the layout being recalled; this allows the server to monitor the client's progress and adherence to the callback. However, the last LAYOUTRETURN in a sequence of returns, SHOULD specify the full range being recalled (see Section 3.5.2 for details).

ERRORS

NFS4ERR_NOMATCHING_LAYOUT

10.2 CB_SIZECHANGED

SYNOPSIS

```
  fh, size -> -
```

ARGUMENT

```
  struct CB_SIZECHANGEDargs {
          nfs_fh4         fh;
          length4         size;
  };
```

RESULT

```
  struct CB_SIZECHANGEDres {
          nfsstat4        status;
  };
```

DESCRIPTION

The CB_SIZECHANGED operation is used to notify the client that the size pertaining to the filehandle associated with "fh", has changed. The new size is specified. Upon reception of this notification callback, the client should update its internal size for the file. If the layout being held for the file is of the NFSv4 file layout type, then the size field within that layout should be updated (see Section 5.5). For other layout types see Section 3.4.2 for more details.

If the handle specified is not one for which the client holds a layout, an NFS4ERR_BADHANDLE error is returned.

ERRORS

NFS4ERR_BADHANDLE

11. Layouts and Aggregation

This section describes several aggregation schemes in a semi-formal way to provide context for layout formats. These definitions will be formalized in other protocols. However, the set of understood types is part of this protocol in order to provide for basic interoperability.

The layout descriptions include (deviceID, objectID) tuples that identify some storage object on some storage device. The addressing

formation associated with the deviceID is obtained with GETDEVICEINFO. The interpretation of the objectID depends on the storage protocol. The objectID could be a filehandle for an NFSv4 storage device. It could be a OSD object ID for an object server. The layout for a block device generally includes additional block map information to enumerate blocks or extents that are part of the layout.

11.1 Simple Map

The data is located on a single storage device. In this case the file server can act as the front end for several storage devices and distribute files among them. Each file is limited in its size and performance characteristics by a single storage device. The simple map consists of (deviceID, objectID).

11.2 Block Extent Map

The data is located on a LUN in the SAN. The layout consists of an array of (deviceID, blockID, offset, length) tuples. Each entry describes a block extent.

11.3 Striped Map (RAID 0)

The data is striped across storage devices. The parameters of the stripe include the number of storage devices (N) and the size of each stripe unit (U). A full stripe of data is N * U bytes. The stripe map consists of an ordered list of (deviceID, objectID) tuples and the parameter value for U. The first stripe unit (the first U bytes) are stored on the first (deviceID, objectID), the second stripe unit on the second (deviceID, objectID) and so forth until the first complete stripe. The data layout then wraps around so that byte (N*U) of the file is stored on the first (deviceID, objectID) in the list, but starting at offset U within that object. The striped layout allows a client to read or write to the component objects in parallel to achieve high bandwidth.

The striped map for a block device would be slightly different. The map is an ordered list of (deviceID, blockID, blocksize), where the deviceID is rotated among a set of devices to achieve striping.

11.4 Replicated Map

The file data is replicated on N storage devices. The map consists of N (deviceID, objectID) tuples. When data is written using this map, it should be written to N objects in parallel. When data is read, any component object can be used.

This map type is controversial because it highlights the issues with error recovery. Those issues get interesting with any scheme that employs redundancy. The handling of errors (e.g., only a subset of replicas get updated) is outside the scope of this protocol extension. Instead, it is a function of the storage protocol and the metadata control protocol.

11.5 Concatenated Map

The map consists of an ordered set of N (deviceID, objectID, size) tuples. Each successive tuple describes the next segment of the file.

11.6 Nested Map

The nested map is used to compose more complex maps out of simpler ones. The map format is an ordered set of M sub-maps, each submap applies to a byte range within the file and has its own type such as the ones introduced above. Any level of nesting is allowed in order to build up complex aggregation schemes.

12. References

12.1 Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.

[2] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, April 2003.

[3] Gibson, G., "pNFS Problem Statement", July 2004, <ftp://www.ietf.org/internet-drafts/draft-gibson-pnfs-problem-statement-01.txt>.

12.2 Informative References

[4] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M., and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)", RFC 3720, April 2004.

[5] Snively, R., "Fibre Channel Protocol for SCSI, 2nd Version (FCP-2)", ANSI/INCITS 350-2003, Oct 2003.

[6] Weber, R., "Object-Based Storage Device Commands (OSD)", ANSI/INCITS 400-2004, July 2004, <http://www.t10.org/ftp/t10/drafts/osd/osd-r10.pdf>.

[7] Black, D., "pNFS Block/Volume Layout", July 2005, <ftp://www.ietf.org/internet-drafts/draft-black-pnfs-block-01.txt>.

[8] Zelenka, J., Welch, B., and B. Halevy, "Object-based pNFS Operations", July 2005, <ftp://www.ietf.org/internet-drafts/draft-zelenka-pnfs-obj-01.txt>.

Authors' Addresses

Garth Goodson
Network Appliance
495 E. Java Dr
Sunnyvale, CA 94089
USA

Phone: +1-408-822-6847
Email: goodson@netapp.com

Brent Welch
Panasas, Inc.
6520 Kaiser Drive
Fremont, CA 95444
USA

Phone: +1-650-608-7770
Email: welch@panasas.com
URI: http://www.panasas.com/

Benny Halevy
Panasas, Inc.
1501 Reedsdale St., #400
Pittsburgh, PA 15233
USA

Phone: +1-412-323-3500
Email: bhalevy@panasas.com
URI: http://www.panasas.com/

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

Phone: +1-508-293-7953
Email: black_david@emc.com

Andy Adamson
CITI University of Michigan
519 W. William
Ann Arbor, MI 48103-4943
USA

Phone: +1-734-764-9465
Email: andros@umich.edu

Appendix A. Acknowledgments

Many members of the pNFS informal working group have helped considerably. The authors would like to thank Gary Grider, Peter Corbett, Dave Noveck, and Peter Honeyman. This work is inspired by the NASD and OSD work done by Garth Gibson. Gary Grider of the national labs (LANL) has been a champion of high-performance parallel I/O.

Acknowledgment

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