Network File System Version 4 T. Haynes
Internet-Draft Hammerspace
Intended status: Standards Track 22 May 2025
Expires: 23 November 2025

Parallel NFS (pNFS) Flexible File Layout Version 2
draft-haynes-nfsv4-flexfiles-v2-01

Abstract

Parallel NFS (pNFS) allows a separation between the metadata (onto a metadata server) and data (onto a storage device) for a file. The flexible file layout type is defined in this document as an extension to pNFS that allows the use of storage devices that require only a limited degree of interaction with the metadata server and use already-existing protocols. Data protection is also added to provide integrity. Both Client-side mirroring and the Mojette algorithm are used for data protection.

Note to Readers

Discussion of this draft takes place on the NFSv4 working group mailing list (nfsv4@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=nfsv4. Source code and issues list for this draft can be found at https://github.com/ietf-wg-nfsv4/uncacheable.

Working Group information can be found at https://github.com/ietf-wg-nfsv4.

This draft is currently a work in progress. It needs to be determined if we want to copy v1 text to v2 or if we want just a diff of the new content. For right now, we are copying the v1 text and adding the new v1 text. Also, expect sections to move as we push the emphasis from flex files to protection types.

As a WIP, the XDR extraction may not yet work.

Status of This Memo

Table of Contents

1. Introduction

In Parallel NFS (pNFS), the metadata server returns layout type structures that describe where file data is located. There are different layout types for different storage systems and methods of arranging data on storage devices. This document defines the flexible file layout type used with file-based data servers that are accessed using the NFS protocols: NFSv3 [RFC1813], NFSv4.0 [RFC7530], NFSv4.1 [RFC8881], and NFSv4.2 [RFC7862].

To provide a global state model equivalent to that of the files layout type, a back-end control protocol might be implemented between the metadata server and NFSv4.1+ storage devices. An implementation can either define its own proprietary mechanism or it could define a control protocol in a Standards Track document. The requirements for a control protocol are specified in [RFC8881] and clarified in [RFC8434].

The control protocol described in this document is based on NFS. It does not provide for knowledge of stateids to be passed between the metadata server and the storage devices. Instead, the storage devices are configured such that the metadata server has full access rights to the data file system and then the metadata server uses synthetic ids to control client access to individual data files.

In traditional mirroring of data, the server is responsible for replicating, validating, and repairing copies of the data file. With client-side mirroring, the metadata server provides a layout that presents the available mirrors to the client. The client then picks a mirror to read from and ensures that all writes go to all mirrors. The client only considers the write transaction to have succeeded if all mirrors are successfully updated. In case of error, the client can use the LAYOUTERROR operation to inform the metadata server, which is then responsible for the repairing of the mirrored copies of the file.

1.1. Definitions

control communication requirements: the specification for information on layouts, stateids, file metadata, and file data that must be communicated between the metadata server and the storage devices. There is a separate set of requirements for each layout type.

control protocol: the particular mechanism that an implementation of a layout type would use to meet the control communication requirement for that layout type. This need not be a protocol as normally understood. In some cases, the same protocol may be used as a control protocol and storage protocol.

client-side mirroring: a feature in which the client, not the server, is responsible for updating all of the mirrored copies of a layout segment.

erasure encoding: tbd

(file) data: that part of the file system object that contains the data to be read or written. It is the contents of the object rather than the attributes of the object.

data server (DS): a pNFS server that provides the file's data when the file system object is accessed over a file-based protocol.

fencing: the process by which the metadata server prevents the storage devices from processing I/O from a specific client to a specific file.

file layout type: a layout type in which the storage devices are accessed via the NFS protocol (see Section 10 of [RFC8881]).

gid: the group id, a numeric value that identifies to which group a file belongs.

layout: the information a client uses to access file data on a storage device. This information includes specification of the protocol (layout type) and the identity of the storage devices to be used.

layout iomode: a grant of either read-only or read/write I/O to the client.

layout segment: a sub-division of a layout. That sub-division might be by the layout iomode (see Sections 3.3.20 and 12.2.9 of [RFC8881]), a striping pattern (see Section 13.3 of [RFC8881]), or requested byte range.

layout stateid: a 128-bit quantity returned by a server that uniquely defines the layout state provided by the server for a specific layout that describes a layout type and file (see Section 12.5.2 of [RFC8881]). Further, Section 12.5.3 of [RFC8881] describes differences in handling between layout stateids and other stateid types.

layout type: a specification of both the storage protocol used to access the data and the aggregation scheme used to lay out the file data on the underlying storage devices.

loose coupling: when the control protocol is a storage protocol.

(file) metadata: the part of the file system object that contains various descriptive data relevant to the file object, as opposed to the file data itself. This could include the time of last modification, access time, EOF position, etc.

metadata server (MDS): the pNFS server that provides metadata information for a file system object. It is also responsible for generating, recalling, and revoking layouts for file system objects, for performing directory operations, and for performing I/O operations to regular files when the clients direct these to the metadata server itself.

mirror: a copy of a layout segment. Note that if one copy of the mirror is updated, then all copies must be updated.

non-systematic encoding: TBD

recalling a layout: a graceful recall, via a callback, of a specific layout by the metadata server to the client. Graceful here means that the client would have the opportunity to flush any WRITEs, etc., before returning the layout to the metadata server.

revoking a layout: an invalidation of a specific layout by the metadata server. Once revocation occurs, the metadata server will not accept as valid any reference to the revoked layout, and a storage device will not accept any client access based on the layout.

resilvering: the act of rebuilding a mirrored copy of a layout segment from a known good copy of the layout segment. Note that this can also be done to create a new mirrored copy of the layout segment.

rsize: the data transfer buffer size used for READs.

stateid: a 128-bit quantity returned by a server that uniquely defines the set of locking-related state provided by the server. Stateids may designate state related to open files, byte-range locks, delegations, or layouts.

storage device: the target to which clients may direct I/O requests when they hold an appropriate layout. See Section 2.1 of [RFC8434] for further discussion of the difference between a data server and a storage device.

storage protocol: the protocol used by clients to do I/O operations to the storage device. Each layout type specifies the set of storage protocols.

systematic encoding: TBD

tight coupling: an arrangement in which the control protocol is one designed specifically for control communication. It may be either a proprietary protocol adapted specifically to a particular metadata server or a protocol based on a Standards Track document.

uid: the user id, a numeric value that identifies which user owns a file.

write hole: TBD

wsize: the data transfer buffer size used for WRITEs.

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Coupling of Storage Devices

A server implementation may choose either a loosely coupled model or a tightly coupled model between the metadata server and the storage devices. [RFC8434] describes the general problems facing pNFS implementations. This document details how the new flexible file layout type addresses these issues. To implement the tightly coupled model, a control protocol has to be defined. As the flexible file layout imposes no special requirements on the client, the control protocol will need to provide:

1. management of both security and LAYOUTCOMMITs and

2. a global stateid model and management of these stateids.

When implementing the loosely coupled model, the only control protocol will be a version of NFS, with no ability to provide a global stateid model or to prevent clients from using layouts inappropriately. To enable client use in that environment, this document will specify how security, state, and locking are to be managed.

2.1. LAYOUTCOMMIT

Regardless of the coupling model, the metadata server has the responsibility, upon receiving a LAYOUTCOMMIT (see Section 18.42 of [RFC8881]) to ensure that the semantics of pNFS are respected (see Section 3.1 of [RFC8434]). These do include a requirement that data written to a data storage device be stable before the occurrence of the LAYOUTCOMMIT.

It is the responsibility of the client to make sure the data file is stable before the metadata server begins to query the storage devices about the changes to the file. If any WRITE to a storage device did not result with stable_how equal to FILE_SYNC, a LAYOUTCOMMIT to the metadata server MUST be preceded by a COMMIT to the storage devices written to. Note that if the client has not done a COMMIT to the storage device, then the LAYOUTCOMMIT might not be synchronized to the last WRITE operation to the storage device.

2.2. Fencing Clients from the Storage Device

With loosely coupled storage devices, the metadata server uses synthetic uids (user ids) and gids (group ids) for the data file, where the uid owner of the data file is allowed read/write access and the gid owner is allowed read-only access. As part of the layout (see ffds_user and ffds_group in Section 6.1), the client is provided with the user and group to be used in the Remote Procedure Call (RPC) [RFC5531] credentials needed to access the data file. Fencing off of clients is achieved by the metadata server changing the synthetic uid and/or gid owners of the data file on the storage device to implicitly revoke the outstanding RPC credentials. A client presenting the wrong credential for the desired access will get an NFS4ERR_ACCESS error.

With this loosely coupled model, the metadata server is not able to fence off a single client; it is forced to fence off all clients. However, as the other clients react to the fencing, returning their layouts and trying to get new ones, the metadata server can hand out a new uid and gid to allow access.

It is RECOMMENDED to implement common access control methods at the storage device file system to allow only the metadata server root (super user) access to the storage device and to set the owner of all directories holding data files to the root user. This approach provides a practical model to enforce access control and fence off cooperative clients, but it cannot protect against malicious clients; hence, it provides a level of security equivalent to AUTH_SYS. It is RECOMMENDED that the communication between the metadata server and storage device be secure from eavesdroppers and man-in-the-middle protocol tampering. The security measure could be physical security (e.g., the servers are co-located in a physically secure area), encrypted communications, or some other technique.

With tightly coupled storage devices, the metadata server sets the user and group owners, mode bits, and Access Control List (ACL) of the data file to be the same as the metadata file. And the client must authenticate with the storage device and go through the same authorization process it would go through via the metadata server. In the case of tight coupling, fencing is the responsibility of the control protocol and is not described in detail in this document. However, implementations of the tightly coupled locking model (see Section 2.3) will need a way to prevent access by certain clients to specific files by invalidating the corresponding stateids on the storage device. In such a scenario, the client will be given an error of NFS4ERR_BAD_STATEID.

The client need not know the model used between the metadata server and the storage device. It need only react consistently to any errors in interacting with the storage device. It should both return the layout and error to the metadata server and ask for a new layout. At that point, the metadata server can either hand out a new layout, hand out no layout (forcing the I/O through it), or deny the client further access to the file.

2.2.1. Implementation Notes for Synthetic uids/gids

The selection method for the synthetic uids and gids to be used for fencing in loosely coupled storage devices is strictly an implementation issue. That is, an administrator might restrict a range of such ids available to the Lightweight Directory Access Protocol (LDAP) ’uid’ field [RFC4519]. The administrator might also be able to choose an id that would never be used to grant access. Then, when the metadata server had a request to access a file, a SETATTR would be sent to the storage device to set the owner and group of the data file. The user and group might be selected in a round-robin fashion from the range of available ids.

Those ids would be sent back as ffds_user and ffds_group to the client, who would present them as the RPC credentials to the storage device. When the client is done accessing the file and the metadata server knows that no other client is accessing the file, it can reset the owner and group to restrict access to the data file.

When the metadata server wants to fence off a client, it changes the synthetic uid and/or gid to the restricted ids. Note that using a restricted id ensures that there is a change of owner and at least one id available that never gets allowed access.

Under an AUTH_SYS security model, synthetic uids and gids of 0 SHOULD be avoided. These typically either grant super access to files on a storage device or are mapped to an anonymous id. In the first case, even if the data file is fenced, the client might still be able to access the file. In the second case, multiple ids might be mapped to the anonymous ids.

2.2.2. Example of using Synthetic uids/gids

The user loghyr creates a file "ompha.c" on the metadata server, which then creates a corresponding data file on the storage device.

The metadata server entry may look like:

```
-rw-r--r--    1 loghyr  staff    1697 Dec  4 11:31 ompha.c
```

Figure 1: Metadata’s view of ompha.c

On the storage device, the file may be assigned some unpredictable synthetic uid/gid to deny access:

```
-rw-r-----    1 19452   28418    1697 Dec  4 11:31 data_ompha.c
```

Figure 2: Data’s view of ompha.c

When the file is opened on a client and accessed, the user will try to get a layout for the data file. Since the layout knows nothing about the user (and does not care), it does not matter whether the user loghyr or garbo opens the file. The client has to present an uid of 19452 to get write permission. If it presents any other value for the uid, then it must give a gid of 28418 to get read access.

Further, if the metadata server decides to fence the file, it should change the uid and/or gid such that these values neither match earlier values for that file nor match a predictable change based on an earlier fencing.

```
-rw-r-----    1 19453   28419    1697 Dec  4 11:31 data_ompha.c
```

Figure 3: Fenced Data’s view of ompha.c

The set of synthetic gids on the storage device should be selected such that there is no mapping in any of the name services used by the storage device, i.e., each group should have no members.

If the layout segment has an iomode of LAYOUTIOMODE4_READ, then the metadata server should return a synthetic uid that is not set on the storage device. Only the synthetic gid would be valid.

The client is thus solely responsible for enforcing file permissions in a loosely coupled model. To allow loghyr write access, it will send an RPC to the storage device with a credential of 1066:1067. To allow garbo read access, it will send an RPC to the storage device with a credential of 1067:1067. The value of the uid does not matter as long as it is not the synthetic uid granted when getting the layout.

While pushing the enforcement of permission checking onto the client may seem to weaken security, the client may already be responsible for enforcing permissions before modifications are sent to a server. With cached writes, the client is always responsible for tracking who is modifying a file and making sure to not coalesce requests from multiple users into one request.

2.3. State and Locking Models

An implementation can always be deployed as a loosely coupled model. There is, however, no way for a storage device to indicate over an NFS protocol that it can definitively participate in a tightly coupled model:

* Storage devices implementing the NFSv3 and NFSv4.0 protocols are always treated as loosely coupled.

* NFSv4.1+ storage devices that do not return the EXCHGID4_FLAG_USE_PNFS_DS flag set to EXCHANGE_ID are indicating that they are to be treated as loosely coupled. From the locking viewpoint, they are treated in the same way as NFSv4.0 storage devices.

* NFSv4.1+ storage devices that do identify themselves with the EXCHGID4_FLAG_USE_PNFS_DS flag set to EXCHANGE_ID can potentially be tightly coupled. They would use a back-end control protocol to implement the global stateid model as described in [RFC8881].

A storage device would have to be either discovered or advertised over the control protocol to enable a tightly coupled model.

2.3.1. Loosely Coupled Locking Model

When locking-related operations are requested, they are primarily dealt with by the metadata server, which generates the appropriate stateids. When an NFSv4 version is used as the data access protocol, the metadata server may make stateid-related requests of the storage devices. However, it is not required to do so, and the resulting stateids are known only to the metadata server and the storage device.

Given this basic structure, locking-related operations are handled as follows:

* OPENs are dealt with by the metadata server. Stateids are selected by the metadata server and associated with the client ID describing the client's connection to the metadata server. The metadata server may need to interact with the storage device to locate the file to be opened, but no locking-related functionality need be used on the storage device.

* OPEN_DOWNGRADE and CLOSE only require local execution on the metadata server.

* Advisory byte-range locks can be implemented locally on the metadata server. As in the case of OPENs, the stateids associated with byte-range locks are assigned by the metadata server and only used on the metadata server.

* Delegations are assigned by the metadata server that initiates recalls when conflicting OPENs are processed. No storage device involvement is required.

* TEST_STATEID and FREE_STATEID are processed locally on the metadata server, without storage device involvement.

All I/O operations to the storage device are done using the anonymous stateid. Thus, the storage device has no information about the openowner and lockowner responsible for issuing a particular I/O operation. As a result:

* Mandatory byte-range locking cannot be supported because the storage device has no way of distinguishing I/O done on behalf of the lock owner from those done by others.

* Enforcement of share reservations is the responsibility of the client. Even though I/O is done using the anonymous stateid, the client must ensure that it has a valid stateid associated with the openowner.

In the event that a stateid is revoked, the metadata server is responsible for preventing client access, since it has no way of being sure that the client is aware that the stateid in question has been revoked.

As the client never receives a stateid generated by a storage device, there is no client lease on the storage device and no prospect of lease expiration, even when access is via NFSv4 protocols. Clients will have leases on the metadata server. In dealing with lease expiration, the metadata server may need to use fencing to prevent revoked stateids from being relied upon by a client unaware of the fact that they have been revoked.

2.3.2. Tightly Coupled Locking Model

When locking-related operations are requested, they are primarily dealt with by the metadata server, which generates the appropriate stateids. These stateids must be made known to the storage device using control protocol facilities, the details of which are not discussed in this document.

Given this basic structure, locking-related operations are handled as follows:

* OPENs are dealt with primarily on the metadata server. Stateids are selected by the metadata server and associated with the client ID describing the client's connection to the metadata server. The metadata server needs to interact with the storage device to locate the file to be opened and to make the storage device aware of the association between the metadata-server-chosen stateid and the client and openowner that it represents. OPEN_DOWNGRADE and CLOSE are executed initially on the metadata server, but the state change made must be propagated to the storage device.

* Advisory byte-range locks can be implemented locally on the metadata server. As in the case of OPENs, the stateids associated with byte-range locks are assigned by the metadata server and are available for use on the metadata server. Because I/O operations are allowed to present lock stateids, the metadata server needs the ability to make the storage device aware of the association between the metadata-server-chosen stateid and the corresponding open stateid it is associated with.

* Mandatory byte-range locks can be supported when both the metadata server and the storage devices have the appropriate support. As in the case of advisory byte-range locks, these are assigned by the metadata server and are available for use on the metadata server. To enable mandatory lock enforcement on the storage device, the metadata server needs the ability to make the storage device aware of the association between the metadata-server-chosen stateid and the client, openowner, and lock (i.e., lockowner, byte-range, and lock-type) that it represents. Because I/O operations are allowed to present lock stateids, this information needs to be propagated to all storage devices to which I/O might be directed rather than only to storage device that contain the locked region.

* Delegations are assigned by the metadata server that initiates recalls when conflicting OPENs are processed. Because I/O operations are allowed to present delegation stateids, the metadata server requires the ability:

  1. to make the storage device aware of the association between the metadata-server-chosen stateid and the filehandle and delegation type it represents

  2. to break such an association.

* TEST_STATEID is processed locally on the metadata server, without storage device involvement.

* FREE_STATEID is processed on the metadata server, but the metadata server requires the ability to propagate the request to the corresponding storage devices.

Because the client will possess and use stateids valid on the storage device, there will be a client lease on the storage device, and the possibility of lease expiration does exist. The best approach for the storage device is to retain these locks as a courtesy. However, if it does not do so, control protocol facilities need to provide the means to synchronize lock state between the metadata server and storage device.

Clients will also have leases on the metadata server that are subject to expiration. In dealing with lease expiration, the metadata server would be expected to use control protocol facilities enabling it to invalidate revoked stateids on the storage device. In the event the client is not responsive, the metadata server may need to use fencing to prevent revoked stateids from being acted upon by the storage device.

3. Client-Side Protection Modes

3.1. Client-Side Mirroring

Do I want this?

4. XDR Description of the Flexible File Layout Type

This document contains the External Data Representation (XDR) [RFC4506] description of the flexible file layout type. The XDR description is embedded in this document in a way that makes it simple for the reader to extract into a ready-to-compile form. The reader can feed this document into the shell script in Figure 4 to produce the machine-readable XDR description of the flexible file layout type.

```
#!/bin/sh
grep '^ *///' $* | sed 's?^ */// ??' | sed 's?^ *///$??'
```

Figure 4: extract.sh

That is, if the above script is stored in a file called "extract.sh" and this document is in a file called "spec.txt", then the reader can run the script as in Figure 5.

```
sh extract.sh < spec.txt > flex_files2_prot.x
```

Figure 5: Example use of extract.sh

The effect of the script is to remove leading blank space from each line, plus a sentinel sequence of "///".

The embedded XDR file header follows. Subsequent XDR descriptions with the sentinel sequence are embedded throughout the document.

Note that the XDR code contained in this document depends on types from the NFSv4.1 nfs4_prot.x file [RFC5662]. This includes both nfs types that end with a 4, such as offset4, length4, etc., as well as more generic types such as uint32_t and uint64_t.

5. Device Addressing and Discovery

Data operations to a storage device require the client to know the network address of the storage device. The NFSv4.1+ GETDEVICEINFO operation (Section 18.40 of [RFC8881]) is used by the client to retrieve that information.

5.1. ff_device_addr4

The ff_device_addr4 data structure (see Figure 7) is returned by the server as the layout-type-specific opaque field da_addr_body in the device_addr4 structure by a successful GETDEVICEINFO operation.

```
/// struct ff_device_versions4 {
///         uint32_t        ffdv_version;
///         uint32_t        ffdv_minorversion;
///         uint32_t        ffdv_rsize;
///         uint32_t        ffdv_wsize;
///         bool            ffdv_tightly_coupled;
/// };
///
```

Figure 6: ff_device_versions4

```
/// struct ff_device_addr4 {
///         multipath_list4     ffda_netaddrs;
///         ff_device_versions4 ffda_versions<>;
/// };
///
```

Figure 7: ff_device_addr4

The ffda_netaddrs field is used to locate the storage device. It MUST be set by the server to a list holding one or more of the device network addresses.

The ffda_versions array allows the metadata server to present choices as to NFS version, minor version, and coupling strength to the client. The ffdv_version and ffdv_minorversion represent the NFS protocol to be used to access the storage device. This layout specification defines the semantics for ffdv_versions 3 and 4. If ffdv_version equals 3, then the server MUST set ffdv_minorversion to 0 and ffdv_tightly_coupled to false. The client MUST then access the storage device using the NFSv3 protocol [RFC1813]. If ffdv_version equals 4, then the server MUST set ffdv_minorversion to one of the NFSv4 minor version numbers, and the client MUST access the storage device using NFSv4 with the specified minor version.

Note that while the client might determine that it cannot use any of the configured combinations of ffdv_version, ffdv_minorversion, and ffdv_tightly_coupled, when it gets the device list from the metadata server, there is no way to indicate to the metadata server as to which device it is version incompatible. However, if the client waits until it retrieves the layout from the metadata server, it can at that time clearly identify the storage device in question (see Section 6.4).

The ffdv_rsize and ffdv_wsize are used to communicate the maximum rsize and wsize supported by the storage device. As the storage device can have a different rsize or wsize than the metadata server, the ffdv_rsize and ffdv_wsize allow the metadata server to communicate that information on behalf of the storage device.

ffdv_tightly_coupled informs the client as to whether or not the metadata server is tightly coupled with the storage devices. Note that even if the data protocol is at least NFSv4.1, it may still be the case that there is loose coupling in effect. If ffdv_tightly_coupled is not set, then the client MUST commit writes to the storage devices for the file before sending a LAYOUTCOMMIT to the metadata server. That is, the writes MUST be committed by the client to stable storage via issuing WRITEs with stable_how == FILE_SYNC or by issuing a COMMIT after WRITEs with stable_how != FILE_SYNC (see Section 3.3.7 of [RFC1813]).

5.2. Storage Device Multipathing

The flexible file layout type supports multipathing to multiple storage device addresses. Storage-device-level multipathing is used for bandwidth scaling via trunking and for higher availability of use in the event of a storage device failure. Multipathing allows the client to switch to another storage device address that may be that of another storage device that is exporting the same data stripe unit, without having to contact the metadata server for a new layout.

To support storage device multipathing, ffda_netaddrs contains an array of one or more storage device network addresses. This array (data type multipath_list4) represents a list of storage devices (each identified by a network address), with the possibility that some storage device will appear in the list multiple times.

The client is free to use any of the network addresses as a destination to send storage device requests. If some network addresses are less desirable paths to the data than others, then the metadata server SHOULD NOT include those network addresses in ffda_netaddrs. If less desirable network addresses exist to provide failover, the RECOMMENDED method to offer the addresses is to provide them in a replacement device-ID-to-device-address mapping or a replacement device ID. When a client finds no response from the storage device using all addresses available in ffda_netaddrs, it SHOULD send a GETDEVICEINFO to attempt to replace the existing device-ID-to-device-address mappings. If the metadata server detects that all network paths represented by ffda_netaddrs are unavailable, the metadata server SHOULD send a CB_NOTIFY_DEVICEID (if the client has indicated it wants device ID notifications for changed device IDs) to change the device-ID-to-device-address mappings to the available addresses. If the device ID itself will be replaced, the metadata server SHOULD recall all layouts with the device ID and thus force the client to get new layouts and device ID mappings via LAYOUTGET and GETDEVICEINFO.

Generally, if two network addresses appear in ffda_netaddrs, they will designate the same storage device. When the storage device is accessed over NFSv4.1 or a higher minor version, the two storage device addresses will support the implementation of client ID or session trunking (the latter is RECOMMENDED) as defined in [RFC8881]. The two storage device addresses will share the same server owner or major ID of the server owner. It is not always necessary for the two storage device addresses to designate the same storage device with trunking being used. For example, the data could be read-only, and the data consist of exact replicas.

6. Flexible File Layout Type

The original layouttype4 introduced in [RFC5662] is modified to as in Figure 8.

```
        enum layouttype4 {
            LAYOUT4_NFSV4_1_FILES   = 1,
            LAYOUT4_OSD2_OBJECTS    = 2,
            LAYOUT4_BLOCK_VOLUME    = 3,
            LAYOUT4_FLEX_FILES      = 4
        };

        struct layout_content4 {
            layouttype4             loc_type;
            opaque                  loc_body<>;
        };

        struct layout4 {
            offset4                 lo_offset;
            length4                 lo_length;
            layoutiomode4           lo_iomode;
            layout_content4         lo_content;
        };
```

Figure 8: The original layout type

This document defines structures associated with the layouttype4 value LAYOUT4_FLEX_FILES. [RFC8881] specifies the loc_body structure as an XDR type "opaque". The opaque layout is uninterpreted by the generic pNFS client layers but is interpreted by the flexible file layout type implementation. This section defines the structure of this otherwise opaque value, ff_layout4.

6.1. ff_layout4

```
   /// const FF_FLAGS_NO_LAYOUTCOMMIT   = 0x00000001;
   /// const FF_FLAGS_NO_IO_THRU_MDS    = 0x00000002;
   /// const FF_FLAGS_NO_READ_IO        = 0x00000004;
   /// const FF_FLAGS_WRITE_ONE_MIRROR  = 0x00000008;
   ///
   /// typedef uint32_t            ff_flags4;
   ///
   /// /*
   ///  * NFsv4.0, NFSv4.1, and NFSv4.2 can all
   ///  * have unique stateids for the file.
   ///  */
   /// struct ff2_file_info4 {
   ///     stateid4                fffi_stateid;
   ///     nfs_fh4                 fffi_fh_vers;
   /// };
   ///
   /// /*
   ///  * For now, allow all protection types to
   ///  * be in the same flags space.
   ///  */
   /// const FF2_DS_FLAGS_ACTIVE        = 0x00000001;
   /// const FF2_DS_FLAGS_SPARE         = 0x00000002;
   /// const FF2_DS_FLAGS_REPAIR        = 0x00000004;
   /// typedef uint32_t            ff2_ds_flags4;
   ///
   /// struct ff2_data_server4 {
   ///     deviceid4               ffds_deviceid;
   ///     uint32_t                ffds_efficiency;
   ///     ff2_file_info4          ffds_file_info<>;
   ///     fattr4_owner            ffds_user;
   ///     fattr4_owner_group      ffds_group;
   ///     ff2_ds_flags4           ffds_flags;
   /// };
   ///
   /// struct ff2_mirror4 {
   ///     ff2_data_server4        ffm_data_servers<>;
   ///     ff2_protection_type     ffds_protection;
   /// };
   ///
   /// // X_Y: Need X+Y to write
   /// // Can lose Y files
   /// // So Y spares
   /// enum ff2_mojette_faulty_devices {
   ///         FF2_MOJETTE_FAULTY_DEVICES_2_1       = 0x1;
   ///         FF2_MOJETTE_FAULTY_DEVICES_4_1       = 0x2;
   ///         FF2_MOJETTE_FAULTY_DEVICES_4_2       = 0x3;
   ///         FF2_MOJETTE_FAULTY_DEVICES_8_1       = 0x4;
   ///         FF2_MOJETTE_FAULTY_DEVICES_8_2       = 0x5;
```

```
   ///         FF2_MOJETTE_FAULTY_DEVICES_8_3       = 0x6;
   ///         FF2_MOJETTE_FAULTY_DEVICES_8_4       = 0x7;
   /// };
   ///
   /// //
   /// // Need to define projection header for READ/WRITE
   /// //
   ///
   /// union ff2_protection_data switch (ff2_protection_type fpd_type) {
   ///     case FF2_PROTECTION_TYPE_MOJETTE:
   ///         uint32_t                        fpd_mojette_rsize;
   ///         uint32_t                        fpd_mojette_wsize;
   ///         ff2_mojette_faulty_devices      fpd_mojette_potection_configuration;
   ///     case FF2_PROTECTION_TYPE_MIRRORED:
   ///         void;
   /// };
   ///
   /// struct ff2_layout4 {
   ///     length4                 ffl_stripe_unit;
   ///     ff2_mirror4             ffl_mirrors<>;
   ///     ff_flags4               ffl_flags;
   ///     uint32_t                ffl_stats_collect_hint;
   ///     ff2_protection_data     ffl_protection_data;
   /// };
   ///
   ///
   ///
```

Figure 9: The flex files layout type v2

```
   struct ff_data_server4 {
       deviceid4              ffds_deviceid;
       uint32_t               ffds_efficiency;
       stateid4               ffds_stateid;
       nfs_fh4                ffds_fh_vers<>;
       fattr4_owner           ffds_user;
       fattr4_owner_group     ffds_group;
   };


   struct ff_mirror4 {
       ff_data_server4        ffm_data_servers<>;
   };


   struct ff_layout4 {
       length4                ffl_stripe_unit;
       ff_mirror4             ffl_mirrors<>;
       ff_flags4              ffl_flags;
       uint32_t               ffl_stats_collect_hint;
   };
```

Figure 10: The flex files layout type v1

The ff_layout4 structure (see Figure 10) specifies a layout in that portion of the data file described in the current layout segment. It is either a single instance or a set of mirrored copies of that portion of the data file. When mirroring is in effect, it protects against loss of data in layout segments.

While not explicitly shown in Figure 10, each layout4 element returned in the logr_layout array of LAYOUTGET4res (see Section 18.43.2 of [RFC8881]) describes a layout segment. Hence, each ff_layout4 also describes a layout segment. It is possible that the file is concatenated from more than one layout segment. Each layout segment MAY represent different striping parameters.

The ffl_stripe_unit field is the stripe unit size in use for the current layout segment. The number of stripes is given inside each mirror by the number of elements in ffm_data_servers. If the number of stripes is one, then the value for ffl_stripe_unit MUST default to zero. The only supported mapping scheme is sparse and is detailed in Section 7. Note that there is an assumption here that both the stripe unit size and the number of stripes are the same across all mirrors.

The ffl_mirrors field is the array of mirrored storage devices that provide the storage for the current stripe; see Figure 11.

The ffl_stats_collect_hint field provides a hint to the client on how often the server wants it to report LAYOUTSTATS for a file. The time is in seconds.

```
            +-----------+
            |           |
            |           |
            |   File    |
            |           |
            |           |
            +-----+-----+
                  |
     +------------+------------+
     |                         |
+----+-----+             +-----+----+
| Mirror 1 |             | Mirror 2 |
+----+-----+             +-----+----+
     |                         |
+-----------+            +-----------+
|+-----------+           |+-----------+
||+-----------+          ||+-----------+
+||  Storage  |          +||  Storage  |
 +|  Devices  |           +|  Devices  |
  +-----------+            +-----------+
```

Figure 11: The Relationship between MDS and DSes

The ffs_mirrors field represents an array of state information for each mirrored copy of the current layout segment. Each element is described by a ff_mirror4 type.

ffds_deviceid provides the deviceid of the storage device holding the data file.

ffds_fh_vers is an array of filehandles of the data file matching the available NFS versions on the given storage device. There MUST be exactly as many elements in ffds_fh_vers as there are in ffda_versions. Each element of the array corresponds to a particular combination of ffdv_version, ffdv_minorversion, and ffdv_tightly_coupled provided for the device. The array allows for server implementations that have different filehandles for different combinations of version, minor version, and coupling strength. See Section 6.4 for how to handle versioning issues between the client and storage devices.

For tight coupling, ffds_stateid provides the stateid to be used by the client to access the file. For loose coupling and an NFSv4 storage device, the client will have to use an anonymous stateid to perform I/O on the storage device. With no control protocol, the metadata server stateid cannot be used to provide a global stateid model. Thus, the server MUST set the ffds_stateid to be the anonymous stateid.

This specification of the ffds_stateid restricts both models for NFSv4.x storage protocols:

loosely couple the stateid has to be an anonymous stateid

tightly couple the stateid has to be a global stateid

A number of issues stem from a mismatch between the fact that ffds_stateid is defined as a single item while ffds_fh_vers is defined as an array. It is possible for each open file on the storage device to require its own open stateid. Because there are established loosely coupled implementations of the version of the protocol described in this document, such potential issues have not been addressed here. It is possible for future layout types to be defined that address these issues, should it become important to provide multiple stateids for the same underlying file.

For loosely coupled storage devices, ffds_user and ffds_group provide the synthetic user and group to be used in the RPC credentials that the client presents to the storage device to access the data files. For tightly coupled storage devices, the user and group on the storage device will be the same as on the metadata server; that is, if ffdv_tightly_coupled (see Section 5.1) is set, then the client MUST ignore both ffds_user and ffds_group.

The allowed values for both ffds_user and ffds_group are specified as owner and owner_group, respectively, in Section 5.9 of [RFC8881]. For NFSv3 compatibility, user and group strings that consist of decimal numeric values with no leading zeros can be given a special interpretation by clients and servers that choose to provide such support. The receiver may treat such a user or group string as representing the same user as would be represented by an NFSv3 uid or gid having the corresponding numeric value. Note that if using Kerberos for security, the expectation is that these values will be a name@domain string.

ffds_efficiency describes the metadata server’s evaluation as to the effectiveness of each mirror. Note that this is per layout and not per device as the metric may change due to perceived load, availability to the metadata server, etc. Higher values denote higher perceived utility. The way the client can select the best mirror to access is discussed in Section 9.1.

ffl_flags is a bitmap that allows the metadata server to inform the client of particular conditions that may result from more or less tight coupling of the storage devices.

FF_FLAGS_NO_LAYOUTCOMMIT can be set to indicate that the client is not required to send LAYOUTCOMMIT to the metadata server.

F_FLAGS_NO_IO_THRU_MDS can be set to indicate that the client should not send I/O operations to the metadata server. That is, even if the client could determine that there was a network disconnect to a storage device, the client should not try to proxy the I/O through the metadata server.

FF_FLAGS_NO_READ_IO can be set to indicate that the client should not send READ requests with the layouts of iomode LAYOUTIOMODE4_RW. Instead, it should request a layout of iomode LAYOUTIOMODE4_READ from the metadata server.

FF_FLAGS_WRITE_ONE_MIRROR can be set to indicate that the client only needs to update one of the mirrors (see Section 9.2).

6.1.1. Error Codes from LAYOUTGET

[RFC8881] provides little guidance as to how the client is to proceed with a LAYOUTGET that returns an error of either NFS4ERR_LAYOUTTRYLATER, NFS4ERR_LAYOUTUNAVAILABLE, and NFS4ERR_DELAY. Within the context of this document:

NFS4ERR_LAYOUTUNAVAILABLE there is no layout available and the I/O is to go to the metadata server. Note that it is possible to have had a layout before a recall and not after.

NFS4ERR_LAYOUTTRYLATER there is some issue preventing the layout from being granted. If the client already has an appropriate layout, it should continue with I/O to the storage devices.

NFS4ERR_DELAY there is some issue preventing the layout from being granted. If the client already has an appropriate layout, it should not continue with I/O to the storage devices.

6.1.2. Client Interactions with FF_FLAGS_NO_IO_THRU_MDS

Even if the metadata server provides the FF_FLAGS_NO_IO_THRU_MDS flag, the client can still perform I/O to the metadata server. The flag functions as a hint. The flag indicates to the client that the metadata server prefers to separate the metadata I/O from the data I/O, most likely for performance reasons.

6.2. LAYOUTCOMMIT

The flexible file layout does not use lou_body inside the loca_layoutupdate argument to LAYOUTCOMMIT. If lou_type is LAYOUT4_FLEX_FILES, the lou_body field MUST have a zero length (see Section 18.42.1 of [RFC8881]).

6.3. Interactions between Devices and Layouts

The file layout type is defined such that the relationship between multipathing and filehandles can result in either 0, 1, or N filehandles (see Section 13.3 of [RFC8881]). Some rationales for this are clustered servers that share the same filehandle or allow for multiple read-only copies of the file on the same storage device. In the flexible file layout type, while there is an array of filehandles, they are independent of the multipathing being used. If the metadata server wants to provide multiple read-only copies of the same file on the same storage device, then it should provide multiple mirrored instances, each with a different ff_device_addr4. The client can then determine that, since the each of the ffds_fh_vers are different, there are multiple copies of the file for the current layout segment available.

6.4. Handling Version Errors

When the metadata server provides the ffda_versions array in the ff_device_addr4 (see Section 5.1), the client is able to determine whether or not it can access a storage device with any of the supplied combinations of ffdv_version, ffdv_minorversion, and ffdv_tightly_coupled. However, due to the limitations of reporting errors in GETDEVICEINFO (see Section 18.40 in [RFC8881]), the client is not able to specify which specific device it cannot communicate with over one of the provided ffdv_version and ffdv_minorversion combinations. Using ff_ioerr4 (Section 10.1.1) inside either the LAYOUTRETURN (see Section 18.44 of [RFC8881]) or the LAYOUTERROR (see Section 15.6 of [RFC7862] and Section 11 of this document), the client can isolate the problematic storage device.

The error code to return for LAYOUTRETURN and/or LAYOUTERROR is NFS4ERR_MINOR_VERS_MISMATCH. It does not matter whether the mismatch is a major version (e.g., client can use NFSv3 but not NFSv4) or minor version (e.g., client can use NFSv4.1 but not NFSv4.2), the error indicates that for all the supplied combinations for ffdv_version and ffdv_minorversion, the client cannot communicate with the storage device. The client can retry the GETDEVICEINFO to see if the metadata server can provide a different combination, or it can fall back to doing the I/O through the metadata server.

7. Striping via Sparse Mapping

While other layout types support both dense and sparse mapping of logical offsets to physical offsets within a file (see, for example, Section 13.4 of [RFC8881]), the flexible file layout type only supports a sparse mapping.

With sparse mappings, the logical offset within a file (L) is also the physical offset on the storage device. As detailed in Section 13.4.4 of [RFC8881], this results in holes across each storage device that does not contain the current stripe index.

```
L: logical offset within the file

W: stripe width
    W = number of elements in ffm_data_servers

S: number of bytes in a stripe
    S = W * ffl_stripe_unit

N: stripe number
    N = L / S
```

Figure 12: Stripe Mapping Math

8. Recovering from Client I/O Errors

The pNFS client may encounter errors when directly accessing the storage devices. However, it is the responsibility of the metadata server to recover from the I/O errors. When the LAYOUT4_FLEX_FILES layout type is used, the client MUST report the I/O errors to the server at LAYOUTRETURN time using the ff_ioerr4 structure (see Section 10.1.1).

The metadata server analyzes the error and determines the required recovery operations such as recovering media failures or reconstructing missing data files.

The metadata server MUST recall any outstanding layouts to allow it exclusive write access to the stripes being recovered and to prevent other clients from hitting the same error condition. In these cases, the server MUST complete recovery before handing out any new layouts to the affected byte ranges.

Although the client implementation has the option to propagate a corresponding error to the application that initiated the I/O operation and drop any unwritten data, the client should attempt to retry the original I/O operation by either requesting a new layout or sending the I/O via regular NFSv4.1+ READ or WRITE operations to the metadata server. The client SHOULD attempt to retrieve a new layout and retry the I/O operation using the storage device first and only retry the I/O operation via the metadata server if the error persists.

9. Mirroring

The flexible file layout type has a simple model in place for the mirroring of the file data constrained by a layout segment. There is no assumption that each copy of the mirror is stored identically on the storage devices. For example, one device might employ compression or deduplication on the data. However, the over-the-wire transfer of the file contents MUST appear identical. Note, this is a constraint of the selected XDR representation in which each mirrored copy of the layout segment has the same striping pattern (see Figure 11).

The metadata server is responsible for determining the number of mirrored copies and the location of each mirror. While the client may provide a hint to how many copies it wants (see Section 10), the metadata server can ignore that hint; in any event, the client has no means to dictate either the storage device (which also means the coupling and/or protocol levels to access the layout segments) or the location of said storage device.

The updating of mirrored layout segments is done via client-side mirroring. With this approach, the client is responsible for making sure modifications are made on all copies of the layout segments it is informed of via the layout. If a layout segment is being resilvered to a storage device, that mirrored copy will not be in the layout. Thus, the metadata server MUST update that copy until the client is presented it in a layout. If the FF_FLAGS_WRITE_ONE_MIRROR is set in ffl_flags, the client need only update one of the mirrors (see Section 9.2). If the client is writing to the layout segments via the metadata server, then the metadata server MUST update all copies of the mirror. As seen in Section 9.3, during the resilvering, the layout is recalled, and the client has to make modifications via the metadata server.

9.1. Selecting a Mirror

When the metadata server grants a layout to a client, it MAY let the client know how fast it expects each mirror to be once the request arrives at the storage devices via the ffds_efficiency member. While the algorithms to calculate that value are left to the metadata server implementations, factors that could contribute to that calculation include speed of the storage device, physical memory available to the device, operating system version, current load, etc.

However, what should not be involved in that calculation is a perceived network distance between the client and the storage device. The client is better situated for making that determination based on past interaction with the storage device over the different available network interfaces between the two; that is, the metadata server might not know about a transient outage between the client and storage device because it has no presence on the given subnet.

As such, it is the client that decides which mirror to access for reading the file. The requirements for writing to mirrored layout segments are presented below.

9.2. Writing to Mirrors

9.2.1. Single Storage Device Updates Mirrors

If the FF_FLAGS_WRITE_ONE_MIRROR flag in ffl_flags is set, the client only needs to update one of the copies of the layout segment. For this case, the storage device MUST ensure that all copies of the mirror are updated when any one of the mirrors is updated. If the storage device gets an error when updating one of the mirrors, then it MUST inform the client that the original WRITE had an error. The client then MUST inform the metadata server (see Section 9.2.3). The client’s responsibility with respect to COMMIT is explained in Section 9.2.4. The client may choose any one of the mirrors and may use ffds_efficiency as described in Section 9.1 when making this choice.

9.2.2. Client Updates All Mirrors

If the FF_FLAGS_WRITE_ONE_MIRROR flag in ffl_flags is not set, the client is responsible for updating all mirrored copies of the layout segments that it is given in the layout. A single failed update is sufficient to fail the entire operation. If all but one copy is updated successfully and the last one provides an error, then the client needs to inform the metadata server about the error. The client can use either LAYOUTRETURN or LAYOUTERROR to inform the metadata server that the update failed to that storage device. If the client is updating the mirrors serially, then it SHOULD stop at the first error encountered and report that to the metadata server. If the client is updating the mirrors in parallel, then it SHOULD wait until all storage devices respond so that it can report all errors encountered during the update.

9.2.3. Handling Write Errors

When the client reports a write error to the metadata server, the metadata server is responsible for determining if it wants to remove the errant mirror from the layout, if the mirror has recovered from some transient error, etc. When the client tries to get a new layout, the metadata server informs it of the decision by the contents of the layout. The client MUST NOT assume that the contents of the previous layout will match those of the new one. If it has updates that were not committed to all mirrors, then it MUST resend those updates to all mirrors.

There is no provision in the protocol for the metadata server to directly determine that the client has or has not recovered from an error. For example, if a storage device was network partitioned from the client and the client reported the error to the metadata server, then the network partition would be repaired, and all of the copies would be successfully updated. There is no mechanism for the client to report that fact, and the metadata server is forced to repair the file across the mirror.

If the client supports NFSv4.2, it can use LAYOUTERROR and LAYOUTRETURN to provide hints to the metadata server about the recovery efforts. A LAYOUTERROR on a file is for a non-fatal error. A subsequent LAYOUTRETURN without a ff_ioerr4 indicates that the client successfully replayed the I/O to all mirrors. Any LAYOUTRETURN with a ff_ioerr4 is an error that the metadata server needs to repair. The client MUST be prepared for the LAYOUTERROR to trigger a CB_LAYOUTRECALL if the metadata server determines it needs to start repairing the file.

9.2.4. Handling Write COMMITs

When stable writes are done to the metadata server or to a single replica (if allowed by the use of FF_FLAGS_WRITE_ONE_MIRROR), it is the responsibility of the receiving node to propagate the written data stably, before replying to the client.

In the corresponding cases in which unstable writes are done, the receiving node does not have any such obligation, although it may choose to asynchronously propagate the updates. However, once a COMMIT is replied to, all replicas must reflect the writes that have been done, and this data must have been committed to stable storage on all replicas.

In order to avoid situations in which stale data is read from replicas to which writes have not been propagated:

* A client that has outstanding unstable writes made to single node (metadata server or storage device) MUST do all reads from that same node.

* When writes are flushed to the server (for example, to implement close-to-open semantics), a COMMIT must be done by the client to ensure that up-to-date written data will be available irrespective of the particular replica read.

9.3. Metadata Server Resilvering of the File

The metadata server may elect to create a new mirror of the layout segments at any time. This might be to resilver a copy on a storage device that was down for servicing, to provide a copy of the layout segments on storage with different storage performance characteristics, etc. As the client will not be aware of the new mirror and the metadata server will not be aware of updates that the client is making to the layout segments, the metadata server MUST recall the writable layout segment(s) that it is resilvering. If the client issues a LAYOUTGET for a writable layout segment that is in the process of being resilvered, then the metadata server can deny that request with an NFS4ERR_LAYOUTUNAVAILABLE. The client would then have to perform the I/O through the metadata server.

10. Flexible File Layout Type Return

layoutreturn_file4 is used in the LAYOUTRETURN operation to convey layout-type-specific information to the server. It is defined in Section 18.44.1 of [RFC8881] (also shown in Figure 13).

```
   /* Constants used for LAYOUTRETURN and CB_LAYOUTRECALL */
   const LAYOUT4_RET_REC_FILE      = 1;
   const LAYOUT4_RET_REC_FSID      = 2;
   const LAYOUT4_RET_REC_ALL       = 3;

   enum layoutreturn_type4 {
           LAYOUTRETURN4_FILE = LAYOUT4_RET_REC_FILE,
           LAYOUTRETURN4_FSID = LAYOUT4_RET_REC_FSID,
           LAYOUTRETURN4_ALL  = LAYOUT4_RET_REC_ALL
   };

struct layoutreturn_file4 {
        offset4         lrf_offset;
        length4         lrf_length;
        stateid4        lrf_stateid;
        /* layouttype4 specific data */
        opaque          lrf_body<>;
};

union layoutreturn4 switch(layoutreturn_type4 lr_returntype) {
        case LAYOUTRETURN4_FILE:
                layoutreturn_file4      lr_layout;
        default:
                void;
};

struct LAYOUTRETURN4args {
        /* CURRENT_FH: file */
        bool                    lora_reclaim;
        layouttype4             lora_layout_type;
        layoutiomode4           lora_iomode;
        layoutreturn4           lora_layoutreturn;
};
```

Figure 13: Layout Return XDR

If the lora_layout_type layout type is LAYOUT4_FLEX_FILES and the lr_returntype is LAYOUTRETURN4_FILE, then the lrf_body opaque value is defined by ff_layoutreturn4 (see Section 10.3). This allows the client to report I/O error information or layout usage statistics back to the metadata server as defined below. Note that while the data structures are built on concepts introduced in NFSv4.2, the effective discriminated union (lora_layout_type combined with ff_layoutreturn4) allows for an NFSv4.1 metadata server to utilize the data.

10.1. I/O Error Reporting

10.1.1. ff_ioerr4

```
/// struct ff_ioerr4 {
///         offset4        ffie_offset;
///         length4        ffie_length;
///         stateid4       ffie_stateid;
///         device_error4  ffie_errors<>;
/// };
///
```

Figure 14: ff_ioerr4

Recall that [RFC7862] defines device_error4 as in Figure 15:

```
struct device_error4 {
        deviceid4       de_deviceid;
        nfsstat4        de_status;
        nfs_opnum4      de_opnum;
};
```

Figure 15: device_error4

The ff_ioerr4 structure is used to return error indications for data files that generated errors during data transfers. These are hints to the metadata server that there are problems with that file. For each error, ffie_errors.de_deviceid, ffie_offset, and ffie_length represent the storage device and byte range within the file in which the error occurred; ffie_errors represents the operation and type of error. The use of device_error4 is described in Section 15.6 of [RFC7862].

Even though the storage device might be accessed via NFSv3 and reports back NFSv3 errors to the client, the client is responsible for mapping these to appropriate NFSv4 status codes as de_status. Likewise, the NFSv3 operations need to be mapped to equivalent NFSv4 operations.

10.2. Layout Usage Statistics

10.2.1. ff_io_latency4

```
/// struct ff_io_latency4 {
///         uint64_t       ffil_ops_requested;
///         uint64_t       ffil_bytes_requested;
///         uint64_t       ffil_ops_completed;
///         uint64_t       ffil_bytes_completed;
///         uint64_t       ffil_bytes_not_delivered;
///         nfstime4       ffil_total_busy_time;
///         nfstime4       ffil_aggregate_completion_time;
/// };
///
```

Figure 16: ff_io_latency4

Both operation counts and bytes transferred are kept in the ff_io_latency4 (see Figure 16. As seen in ff_layoutupdate4 (see Section 10.2.2), READ and WRITE operations are aggregated separately. READ operations are used for the ff_io_latency4 ffl_read. Both WRITE and COMMIT operations are used for the ff_io_latency4 ffl_write. "Requested" counters track what the client is attempting to do, and "completed" counters track what was done. There is no requirement that the client only report completed results that have matching requested results from the reported period.

ffil_bytes_not_delivered is used to track the aggregate number of bytes requested but not fulfilled due to error conditions. ffil_total_busy_time is the aggregate time spent with outstanding RPC calls. ffil_aggregate_completion_time is the sum of all round-trip times for completed RPC calls.

In Section 3.3.1 of [RFC8881], the nfstime4 is defined as the number of seconds and nanoseconds since midnight or zero hour January 1, 1970 Coordinated Universal Time (UTC). The use of nfstime4 in ff_io_latency4 is to store time since the start of the first I/O from the client after receiving the layout. In other words, these are to be decoded as duration and not as a date and time.

Note that LAYOUTSTATS are cumulative, i.e., not reset each time the operation is sent. If two LAYOUTSTATS operations for the same file and layout stateid originate from the same NFS client and are processed at the same time by the metadata server, then the one containing the larger values contains the most recent time series data.

10.2.2. ff_layoutupdate4

```
/// struct ff_layoutupdate4 {
///         netaddr4       ffl_addr;
///         nfs_fh4        ffl_fhandle;
///         ff_io_latency4 ffl_read;
///         ff_io_latency4 ffl_write;
///         nfstime4       ffl_duration;
///         bool           ffl_local;
/// };
///
```

Figure 17: ff_layoutupdate4

ffl_addr differentiates which network address the client is connected to on the storage device. In the case of multipathing, ffl_fhandle indicates which read-only copy was selected. ffl_read and ffl_write convey the latencies for both READ and WRITE operations, respectively. ffl_duration is used to indicate the time period over which the statistics were collected. If true, ffl_local indicates that the I/O was serviced by the client’s cache. This flag allows the client to inform the metadata server about "hot" access to a file it would not normally be allowed to report on.

10.2.3. ff_iostats4

```
/// struct ff_iostats4 {
///         offset4           ffis_offset;
///         length4           ffis_length;
///         stateid4          ffis_stateid;
///         io_info4          ffis_read;
///         io_info4          ffis_write;
///         deviceid4         ffis_deviceid;
///         ff_layoutupdate4  ffis_layoutupdate;
/// };
///
```

Figure 18: ff_iostats4

[RFC7862] defines io_info4 as in Figure 18.

```
struct io_info4 {
        uint64_t        ii_count;
        uint64_t        ii_bytes;
};
```

Figure 19: io_info4

With pNFS, data transfers are performed directly between the pNFS client and the storage devices. Therefore, the metadata server has no direct knowledge of the I/O operations being done and thus cannot create on its own statistical information about client I/O to optimize the data storage location. ff_iostats4 MAY be used by the client to report I/O statistics back to the metadata server upon returning the layout.

Since it is not feasible for the client to report every I/O that used the layout, the client MAY identify "hot" byte ranges for which to report I/O statistics. The definition and/or configuration mechanism of what is considered "hot" and the size of the reported byte range are out of the scope of this document. For client implementation, providing reasonable default values and an optional run-time management interface to control these parameters is suggested. For example, a client can define the default byte-range resolution to be 1 MB in size and the thresholds for reporting to be 1 MB/second or 10 I/O operations per second.

For each byte range, ffis_offset and ffis_length represent the starting offset of the range and the range length in bytes. ffis_read.ii_count, ffis_read.ii_bytes, ffis_write.ii_count, and ffis_write.ii_bytes represent the number of contiguous READ and WRITE I/Os and the respective aggregate number of bytes transferred within the reported byte range.

The combination of ffis_deviceid and ffl_addr uniquely identifies both the storage path and the network route to it. Finally, ffl_fhandle allows the metadata server to differentiate between multiple read-only copies of the file on the same storage device.

10.3. ff_layoutreturn4

```
/// struct ff_layoutreturn4 {
///         ff_ioerr4     fflr_ioerr_report<>;
///         ff_iostats4   fflr_iostats_report<>;
/// };
///
```

Figure 20: ff_layoutreturn4

When data file I/O operations fail, fflr_ioerr_report<> is used to report these errors to the metadata server as an array of elements of type ff_ioerr4. Each element in the array represents an error that occurred on the data file identified by ffie_errors.de_deviceid. If no errors are to be reported, the size of the fflr_ioerr_report<> array is set to zero. The client MAY also use fflr_iostats_report<> to report a list of I/O statistics as an array of elements of type ff_iostats4. Each element in the array represents statistics for a particular byte range. Byte ranges are not guaranteed to be disjoint and MAY repeat or intersect.

11. Flexible File Layout Type LAYOUTERROR

If the client is using NFSv4.2 to communicate with the metadata server, then instead of waiting for a LAYOUTRETURN to send error information to the metadata server (see Section 10.1), it MAY use LAYOUTERROR (see Section 15.6 of [RFC7862]) to communicate that information. For the flexible file layout type, this means that LAYOUTERROR4args is treated the same as ff_ioerr4.

12. Flexible File Layout Type LAYOUTSTATS

If the client is using NFSv4.2 to communicate with the metadata server, then instead of waiting for a LAYOUTRETURN to send I/O statistics to the metadata server (see Section 10.2), it MAY use LAYOUTSTATS (see Section 15.7 of [RFC7862]) to communicate that information. For the flexible file layout type, this means that LAYOUTSTATS4args.lsa_layoutupdate is overloaded with the same contents as in ffis_layoutupdate.

13. Flexible File Layout Type Creation Hint

The layouthint4 type is defined in the [RFC8881] as in Figure 21.

```
struct layouthint4 {
    layouttype4        loh_type;
    opaque             loh_body<>;
};
```

Figure 21: layouthint4 v1

{{fig-layouthint4-v1}}

The layouthint4 structure is used by the client to pass a hint about the type of layout it would like created for a particular file. If the loh_type layout type is LAYOUT4_FLEX_FILES, then the loh_body opaque value is defined by the ff_layouthint4 type.

14. ff_layouthint4

```
/// union ff2_mirrors_hint switch (ff2_protection_type ffmh_type) {
///     case FF2_PROTECTION_TYPE_MOJETTE:
///         void;
///     case FF2_PROTECTION_TYPE_MIRRORED:
///         void;
/// };
///
/// /*
///  * We could have this be simply ff2_protection_type
///  * for the client to state what protection algorithm
///  * it wants.
///  */
/// struct ff2_layouthint4 {
///     ff2_protection_type fflh_supported_types<>;
///     ff2_mirrors_hint fflh_mirrors_hint;
/// };

union ff_mirrors_hint switch (bool ffmc_valid) {
    case TRUE:
        uint32_t    ffmc_mirrors;
    case FALSE:
        void;
};

struct ff_layouthint4 {
    ff_mirrors_hint    fflh_mirrors_hint;
};
```

Figure 22: ff_layouthint4 v2

This type conveys hints for the desired data map. All parameters are optional so the client can give values for only the parameter it cares about.

15. Recalling a Layout

While Section 12.5.5 of [RFC8881] discusses reasons independent of layout type for recalling a layout, the flexible file layout type metadata server should recall outstanding layouts in the following cases:

* When the file’s security policy changes, i.e., ACLs or permission mode bits are set.

* When the file’s layout changes, rendering outstanding layouts invalid.

* When existing layouts are inconsistent with the need to enforce locking constraints.

* When existing layouts are inconsistent with the requirements regarding resilvering as described in Section 9.3.

15.1. CB_RECALL_ANY

The metadata server can use the CB_RECALL_ANY callback operation to notify the client to return some or all of its layouts. Section 22.3 of [RFC8881] defines the allowed types of the "NFSv4 Recallable Object Types Registry".

```
/// const RCA4_TYPE_MASK_FF2_LAYOUT_MIN     = 20;
/// const RCA4_TYPE_MASK_FF2_LAYOUT_MAX     = 21;
///
```

Figure 23: RCA4 masks for v2

```
struct  CB_RECALL_ANY4args      {
    uint32_t        craa_layouts_to_keep;
    bitmap4         craa_type_mask;
};
```

Figure 24: CB_RECALL_ANY4args XDR

Typically, CB_RECALL_ANY will be used to recall client state when the server needs to reclaim resources. The craa_type_mask bitmap specifies the type of resources that are recalled, and the craa_layouts_to_keep value specifies how many of the recalled flexible file layouts the client is allowed to keep. The mask flags for the flexible file layout type are defined as in Figure 25.

```
/// enum ff_cb_recall_any_mask {
///     PNFS_FF_RCA4_TYPE_MASK_READ = 20,
///     PNFS_FF_RCA4_TYPE_MASK_RW   = 21
/// };
///
```

Figure 25: Recall Mask Flags for v2

The flags represent the iomode of the recalled layouts. In response, the client SHOULD return layouts of the recalled iomode that it needs the least, keeping at most craa_layouts_to_keep flexible file layouts.

The PNFS_FF_RCA4_TYPE_MASK_READ flag notifies the client to return layouts of iomode LAYOUTIOMODE4_READ. Similarly, the PNFS_FF_RCA4_TYPE_MASK_RW flag notifies the client to return layouts of iomode LAYOUTIOMODE4_RW. When both mask flags are set, the client is notified to return layouts of either iomode.

16. Client Fencing

In cases where clients are uncommunicative and their lease has expired or when clients fail to return recalled layouts within a lease period, the server MAY revoke client layouts and reassign these resources to other clients (see Section 12.5.5 of [RFC8881]). To avoid data corruption, the metadata server MUST fence off the revoked clients from the respective data files as described in Section 2.2.

17. Security Considerations

The combination of components in a pNFS system is required to preserve the security properties of NFSv4.1+ with respect to an entity accessing data via a client. The pNFS feature partitions the NFSv4.1+ file system protocol into two parts: the control protocol and the data protocol. As the control protocol in this document is NFS, the security properties are equivalent to the version of NFS being used. The flexible file layout further divides the data protocol into metadata and data paths. The security properties of the metadata path are equivalent to those of NFSv4.1x (see Sections 1.7.1 and 2.2.1 of [RFC8881]). And the security properties of the data path are equivalent to those of the version of NFS used to access the storage device, with the provision that the metadata server is responsible for authenticating client access to the data file. The metadata server provides appropriate credentials to the client to access data files on the storage device. It is also responsible for revoking access for a client to the storage device.

The metadata server enforces the file access control policy at LAYOUTGET time. The client should use RPC authorization credentials for getting the layout for the requested iomode ((LAYOUTIOMODE4_READ or LAYOUTIOMODE4_RW), and the server verifies the permissions and ACL for these credentials, possibly returning NFS4ERR_ACCESS if the client is not allowed the requested iomode. If the LAYOUTGET operation succeeds, the client receives, as part of the layout, a set of credentials allowing it I/O access to the specified data files corresponding to the requested iomode. When the client acts on I/O operations on behalf of its local users, it MUST authenticate and authorize the user by issuing respective OPEN and ACCESS calls to the metadata server, similar to having NFSv4 data delegations.

The combination of filehandle, synthetic uid, and gid in the layout is the way that the metadata server enforces access control to the data server. The client only has access to filehandles of file objects and not directory objects. Thus, given a filehandle in a layout, it is not possible to guess the parent directory filehandle. Further, as the data file permissions only allow the given synthetic uid read/write permission and the given synthetic gid read permission, knowing the synthetic ids of one file does not necessarily allow access to any other data file on the storage device.

The metadata server can also deny access at any time by fencing the data file, which means changing the synthetic ids. In turn, that forces the client to return its current layout and get a new layout if it wants to continue I/O to the data file.

If access is allowed, the client uses the corresponding (read-only or read/write) credentials to perform the I/O operations at the data file's storage devices. When the metadata server receives a request to change a file's permissions or ACL, it SHOULD recall all layouts for that file and then MUST fence off any clients still holding outstanding layouts for the respective files by implicitly invalidating the previously distributed credential on all data file comprising the file in question. It is REQUIRED that this be done before committing to the new permissions and/or ACL. By requesting new layouts, the clients will reauthorize access against the modified access control metadata. Recalling the layouts in this case is intended to prevent clients from getting an error on I/Os done after the client was fenced off.

17.1. Transport Layer Security

17.2. RPCSEC_GSS and Security Services

_Why we don't want to support RPCSEC_GSS._

Because of the special use of principals within the loosely coupled model, the issues are different depending on the coupling model.

17.2.1. Loosely Coupled

RPCSEC_GSS version 3 (RPCSEC_GSSv3) [RFC7861] contains facilities that would allow it to be used to authorize the client to the storage device on behalf of the metadata server. Doing so would require that each of the metadata server, storage device, and client would need to implement RPCSEC_GSSv3 using an RPC-application-defined structured privilege assertion in a manner described in Section 4.9.1 of [RFC7862]. The specifics necessary to do so are not described in this document. This is principally because any such specification would require extensive implementation work on a wide range of storage devices, which would be unlikely to result in a widely usable specification for a considerable time.

As a result, the layout type described in this document will not provide support for use of RPCSEC_GSS together with the loosely coupled model. However, future layout types could be specified, which would allow such support, either through the use of RPCSEC_GSSv3 or in other ways.

17.2.2. Tightly Coupled

With tight coupling, the principal used to access the metadata file is exactly the same as used to access the data file. The storage device can use the control protocol to validate any RPC credentials. As a result, there are no security issues related to using RPCSEC_GSS with a tightly coupled system. For example, if Kerberos V5 Generic Security Service Application Program Interface (GSS-API) [RFC4121] is used as the security mechanism, then the storage device could use a control protocol to validate the RPC credentials to the metadata server.

18. IANA Considerations

[RFC8881] introduced the "pNFS Layout Types Registry"; new layout type numbers in this registry need to be assigned by IANA. This document defines the protocol associated with an existing layout type number: LAYOUT4_FLEX_FILES (see Table 1).

Layout Type Name	Value	RFC	How	Minor Versions
LAYOUT4_FLEX_FILES_V2	0x6	RFCTBD10	L	1

Table 1: Layout Type Assignments

[RFC8881] also introduced the "NFSv4 Recallable Object Types Registry". This document defines new recallable objects for RCA4_TYPE_MASK_FF2_LAYOUT_MIN and RCA4_TYPE_MASK_FF2_LAYOUT_MAX (see Table 2).

Recallable Object Type Name	Value	RFC	How	Minor Versions
RCA4_TYPE_MASK_FF2_LAYOUT_MIN	20	RFCTBD10	L	1
RCA4_TYPE_MASK_FF2_LAYOUT_MAX	21	RFCTBD10	L	1

Table 2: Recallable Object Type Assignments

Acknowledgments

The following from Hammerspace were instrumental in driving Flex Files v2: David Flynn, Trond Myklebust, Didier Feron, Jean-Pierre Monchanin, Pierre Evenou, and Brian Pawlowski.

Christoph Helwig was instrumental in making sure Flex Files v2 was applicable to more than the Mojette Transformation.

Pierre Evenou provided the sections for the Mojette Transformation.

Open Action Items

This section is to be removed before publishing as an RFC.

1. How to describe projection header?

2. It is Little Endian, so not good for XDR?

3. If we add XDR, how does v3 handle it?

4. IANA registration for new Protection Types

5. Proxy registration

6. TLS

References

Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, DOI 10.17487/RFC4121, July 2005, <https://www.rfc-editor.org/rfc/rfc4121>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/rfc/rfc4506>.

[RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, DOI 10.17487/RFC5531, May 2009, <https://www.rfc-editor.org/rfc/rfc5531>.

[RFC5662] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, DOI 10.17487/RFC5662, January 2010, <https://www.rfc-editor.org/rfc/rfc5662>.

[RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <https://www.rfc-editor.org/rfc/rfc7530>.

[RFC7861] Adamson, A. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", RFC 7861, DOI 10.17487/RFC7861, November 2016, <https://www.rfc-editor.org/rfc/rfc7861>.

[RFC7862] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 Protocol", RFC 7862, DOI 10.17487/RFC7862, November 2016, <https://www.rfc-editor.org/rfc/rfc7862>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8434] Haynes, T., "Requirements for Parallel NFS (pNFS) Layout Types", RFC 8434, DOI 10.17487/RFC8434, August 2018, <https://www.rfc-editor.org/rfc/rfc8434>.

[RFC8435] Halevy, B. and T. Haynes, "Parallel NFS (pNFS) Flexible File Layout", RFC 8435, DOI 10.17487/RFC8435, August 2018, <https://www.rfc-editor.org/rfc/rfc8435>.

[RFC8881] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <https://www.rfc-editor.org/rfc/rfc8881>.

Informative References

[RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <https://www.rfc-editor.org/rfc/rfc1813>.

[RFC4519] Sciberras, A., Ed., "Lightweight Directory Access Protocol (LDAP): Schema for User Applications", RFC 4519, DOI 10.17487/RFC4519, June 2006, <https://www.rfc-editor.org/rfc/rfc4519>.

Author’s Address

Thomas Haynes
Hammerspace
Email: loghyr@gmail.com