

Network File System Version 4
Internet-Draft
Intended status: Standards Track
Expires: 28 September 2026

T. Haynes
Hammerspace
27 March 2026

Parallel NFS (pNFS) Flexible File Layout Version 2
draft-haynes-nfsv4-flexfiles-v2-03

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 Version 2 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 Erasure Coding algorithms 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/flexfiles-v2>.

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 v2 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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 28 September 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	6
1.1. Definitions	7
1.2. Requirements Language	10
2. Coupling of Storage Devices	11
2.1. LAYOUTCOMMIT	11
2.2. Fencing Clients from the Storage Device	12
2.2.1. Implementation Notes for Synthetic uids/gids	13
2.2.2. Example of using Synthetic uids/gids	13
2.3. State and Locking Models	15
2.3.1. Loosely Coupled Locking Model	15
2.3.2. Tightly Coupled Locking Model	16
3. XDR Description of the Flexible File Layout Type	18
4. Device Addressing and Discovery	19
4.1. ff_device_addr4	19
4.2. Storage Device Multipathing	20
5. Flexible File Version 2 Layout Type	21
5.1. ffv2_coding_type4	22
5.1.1. Encoding Type Interoperability	23
5.2. ffv2_layout4	23
5.2.1. ffv2_flags4	24
5.3. ffv2_file_info4	24
5.4. ffv2_ds_flags4	25
5.5. ffv2_data_server4	25
5.6. ffv2_coding_type_data4	26
5.7. ffv2_key4	27

5.8.	ffv2_mirror4	27
5.9.	ffv2_layout4	28
5.10.	ffv2_data_protection4	29
5.11.	ffv2_layouthint4	30
5.11.1.	Error Codes from LAYOUTGET	33
5.11.2.	Client Interactions with FF_FLAGS_NO_IO_THRU_MDS	34
5.12.	LAYOUTCOMMIT	34
5.13.	Interactions between Devices and Layouts	34
5.14.	Handling Version Errors	34
6.	Striping via Sparse Mapping	35
7.	Recovering from Client I/O Errors	35
8.	Client-Side Protection Modes	36
8.1.	Client-Side Mirroring	36
8.1.1.	Selecting a Mirror	37
8.1.2.	Writing to Mirrors	37
8.1.3.	Metadata Server Resilvering of the File	39
8.2.	Erasur Coding	40
8.2.1.	Encoding a Data Block	40
8.2.2.	Decoding a Data Block	44
8.2.3.	Write Modes	46
8.2.4.	Reading Chunks	51
8.2.5.	Whole File Repair	51
8.3.	Mixing of Coding Types	52
8.4.	Reed-Solomon Vandermonde Encoding (FFV2_ENCODING_RS_VANDERMONDE)	53
8.4.1.	Overview	53
8.4.2.	Galois Field Arithmetic	53
8.4.3.	Encoding Matrix	54
8.4.4.	Encoding	54
8.4.5.	Decoding	54
8.4.6.	RS Interoperability Requirements	55
8.4.7.	RS Shard Sizes	55
8.5.	Mojette Transform Encoding (FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC)	56
8.5.1.	Overview	56
8.5.2.	Grid Structure	56
8.5.3.	Directions	56
8.5.4.	Forward Transform (Encoding)	57
8.5.5.	Katz Reconstruction Criterion	57
8.5.6.	Inverse Transform (Decoding)	57
8.5.7.	Systematic Mojette	58
8.5.8.	Non-Systematic Mojette	58
8.5.9.	Mojette Shard Sizes	58
8.6.	Comparison of Encoding Types	59
8.7.	Handling write holes	59
9.	NFSv4.2 Operations Allowed to Data Files	60
10.	Flexible File Layout Type Return	60

10.1.	I/O Error Reporting	61
10.1.1.	ff_ioerr4	62
10.2.	Layout Usage Statistics	62
10.2.1.	ff_io_latency4	62
10.2.2.	ff_layoutupdate4	63
10.2.3.	ff_iostats4	64
10.3.	ff_layoutreturn4	65
11.	Flexible File Layout Type LAYOUTERROR	66
12.	Flexible File Layout Type LAYOUTSTATS	66
13.	Flexible File Layout Type Creation Hint	66
14.	ff_layouthint4	66
15.	Recalling a Layout	67
15.1.	CB_RECALL_ANY	67
16.	Client Fencing	68
17.	New NFSv4.2 Error Values	68
17.1.	Error Definitions	69
17.1.1.	NFS4ERR_CODING_NOT_SUPPORTED (Error Code 10097)	69
17.1.2.	NFS4ERR_PAYLOAD_NOT_CONSISTENT (Error Code 10098)	69
17.1.3.	NFS4ERR_CHUNK_LOCKED (Error Code 10099)	70
17.1.4.	NFS4ERR_CHUNK_GUARDED (Error Code 10100)	70
17.2.	Operations and Their Valid Errors	70
17.3.	Callback Operations and Their Valid Errors	72
17.4.	Errors and the Operations That Use Them	72
18.	EXCHGID4_FLAG_USE_PNFS_DS	72
19.	New NFSv4.2 Attributes	73
19.1.	Attribute 89: fattr4_coding_block_size	73
20.	New NFSv4.2 Common Data Structures	73
20.1.	chunk_guard4	73
20.2.	chunk_owner4	74
21.	New NFSv4.2 Operations	74
21.1.	Operation 77: CHUNK_COMMIT - Activate Cached Chunk Data	75
21.1.1.	ARGUMENTS	75
21.1.2.	RESULTS	75
21.1.3.	DESCRIPTION	76
21.2.	Operation 78: CHUNK_ERROR - Report Error on Cached Chunk Data	76
21.2.1.	ARGUMENTS	76
21.2.2.	RESULTS	77
21.2.3.	DESCRIPTION	77
21.3.	Operation 79: CHUNK_FINALIZE - Transition Chunks from Pending to Finalized	77
21.3.1.	ARGUMENTS	77
21.3.2.	RESULTS	77
21.3.3.	DESCRIPTION	78
21.4.	Operation 80: CHUNK_HEADER_READ - Read Chunk Header from File	78
21.4.1.	ARGUMENTS	78

21.4.2. RESULTS	78
21.4.3. DESCRIPTION	79
21.5. Operation 81: CHUNK_LOCK - Lock Cached Chunk Data	79
21.5.1. ARGUMENTS	79
21.5.2. RESULTS	79
21.5.3. DESCRIPTION	80
21.6. Operation 82: CHUNK_READ - Read Chunks from File	80
21.6.1. ARGUMENTS	80
21.6.2. RESULTS	80
21.6.3. DESCRIPTION	81
21.7. Operation 83: CHUNK_REPAIRED - Confirm Repair of Errored Chunk Data	83
21.7.1. ARGUMENTS	83
21.7.2. RESULTS	84
21.7.3. DESCRIPTION	84
21.8. Operation 84: CHUNK_ROLLBACK - Rollback Changes on Cached Chunk Data	84
21.8.1. ARGUMENTS	84
21.8.2. RESULTS	84
21.8.3. DESCRIPTION	85
21.9. Operation 85: CHUNK_UNLOCK - Unlock Cached Chunk Data	85
21.9.1. ARGUMENTS	85
21.9.2. RESULTS	86
21.9.3. DESCRIPTION	86
21.10. Operation 86: CHUNK_WRITE - Write Chunks to File	86
21.10.1. ARGUMENTS	86
21.10.2. RESULTS	87
21.10.3. DESCRIPTION	87
21.11. Operation 87: CHUNK_WRITE_REPAIR - Write Repaired Cached Chunk Data	89
21.11.1. ARGUMENTS	89
21.11.2. RESULTS	89
21.11.3. DESCRIPTION	90
22. Security Considerations	91
22.1. CRC32 Integrity Scope	92
22.2. Chunk Lock and Lease Expiry	92
22.3. Error Code Information Disclosure	93
22.4. Transport Layer Security	93
22.5. RPCSEC_GSS and Security Services	93
22.5.1. Loosely Coupled	93
22.5.2. Tightly Coupled	94
23. IANA Considerations	94
Acknowledgments	96
References	97
Normative References	97
Informative References	98
Author's Address	99

1. Introduction

In Parallel NFS (pNFS) (see Section 12 of [RFC8881]), 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. [RFC8435] defined the Flexible File Version 1 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.

This client side mirroring provides for replication of data but does not provide for integrity of data. In the event of an error, a user would be able to repair the file by silvering the mirror contents. I.e., they would pick one of the mirror instances and replicate it to the other instance locations.

However, lacking integrity checks, silent corruptions are not able to be detected and the choice of what constitutes the good copy is difficult. This document updates the Flexible File Layout Type to version 2 by providing error-detection integrity (CRC32) for erasure coding. Data blocks are transformed into a header and a chunk. It introduces new operations that allow the client to rollback writes to the data file.

Using the process detailed in [RFC8178], the revisions in this document become an extension of NFSv4.2 [RFC7862]. They are built on top of the external data representation (XDR) [RFC4506] generated from [RFC7863].

This document defines LAYOUT4_FLEX_FILES_V2, a new and independent layout type that coexists with the Flexible File Layout Type version 1 (LAYOUT4_FLEX_FILES, [RFC8435]). The two layout types are NOT backward compatible: an FFv2 layout cannot be parsed as an FFv1 layout and vice versa. A server MAY support both layout types simultaneously; a client selects the desired layout type in its LAYOUTGET request.

1.1. Definitions

chunk: One of the resulting chunks to be exchanged with a data server after a transformation has been applied to a data block. The resulting chunk may be a different size than the data block.

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.

data block: A block of data in the client's cache for a file.

data file: The data portion of the file, stored on the data server.

replication of data: Data replication is making and storing multiple

copies of data in different locations.

Erasure Coding: A data protection scheme where a block of data is replicated into fragments and additional redundant fragments are added to achieve parity. The new chunks are stored in different locations.

Client Side Erasure Coding: A file based integrity method where copies are maintained in parallel.

(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 5.12.4 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: An erasure coding scheme in which the encoded shards do not contain verbatim copies of the original data. Every read requires decoding, even when no shards are lost. The Mojette non-systematic transform is an example. Non-systematic encodings are typically used for archival workloads where reads are infrequent.

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: An erasure coding scheme in which the first k of the $k+m$ encoded shards are identical to the original k data blocks. A healthy read (no failures) requires no decoding — the data shards are read directly. Decoding is triggered only when data shards are missing. Reed-Solomon Vandermonde and Mojetta systematic are examples.

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: A write hole is a data corruption scenario where either two clients are trying to write to the same chunk or one client is overwriting an existing chunk of data.

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.

The loosely and tightly coupled locking models defined in Section 2.3 of [RFC8435] apply equally to this layout type, including the use of anonymous stateids with loosely coupled storage devices, the handling of lock and delegation stateids, and the mandatory byte-range lock requirements for the tightly coupled model.

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 `ffv2ds_user` and `ffv2ds_group` in Section 5.2), 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 `ffv2ds_user` and `ffv2ds_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 ompa.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_ompa.c
```

Figure 2: Data's view of ompa.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_ompa.c
```

Figure 3: Fenced Data's view of ompa.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. 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 "///".

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`.

While the XDR can be appended to that from [RFC7863], the various code snippets belong in their respective areas of that XDR.

4. 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.

4.1. `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 5.14).

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]).

4.2. 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.

5. Flexible File Version 2 Layout Type

The original `layouttype4` introduced in [RFC5662] is extended as shown in Figure 8.

```

enum layouttype4 {
    LAYOUT4_NFSV4_1_FILES    = 1,
    LAYOUT4_OSD2_OBJECTS     = 2,
    LAYOUT4_BLOCK_VOLUME     = 3,
    LAYOUT4_FLEX_FILES       = 4,
    LAYOUT4_FLEX_FILES_V2    = 5
};

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_V2`. [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, `ffv2_layout4`.

5.1. `ffv2_coding_type4`

```

/// enum ffv2_coding_type4 {
///     FFV2_CODING_MIRRORED           = 1,
///     FFV2_ENCODING_MOJETTE_SYSTEMATIC = 2,
///     FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC = 3,
///     FFV2_ENCODING_RS_VANDERMONDE   = 4
/// };

```

Figure 9: The coding type

The `ffv2_coding_type4` (see Figure 9) encompasses a new IANA registry for 'Flexible Files Version 2 Erasure Coding Type Registry'. I.e., instead of defining a new Layout Type for each Erasure Coding, we define a new Erasure Coding Type. Except for `FFV2_CODING_MIRRORED`, each of the types is expected to employ the new operations in this document.

FFV2_CODING_MIRRORED offers replication of data and not integrity of data. As such, it does not need operations like `CHUNK_WRITE` (see Section 21.10).

5.1.1. Encoding Type Interoperability

The data servers do not interpret erasure-coded data — they store and return opaque chunks. The NFS wire protocol likewise does not depend on the encoding mathematics. However, a client that writes data using one encoding type **MUST** be able to read it back, and a different client implementation **MUST** be able to read data written by the first client if both claim to support the same encoding type.

This interoperability requirement means that each registered encoding type **MUST** fully specify the encoding and decoding mathematics such that two independent implementations produce byte-identical encoded output for the same input. The specification of a new encoding type **MUST** include one of the following:

1. A complete mathematical specification of the encoding and decoding algorithms, including all parameters (e.g., field polynomial, matrix construction, element size) sufficient for an independent implementation to produce interoperable results.
2. A reference to a published patent or pending patent application that contains the algorithm specification. Implementors can then evaluate the licensing terms and decide whether to support the encoding type.
3. A declaration that the encoding type is a proprietary implementation. In this case, the encoding type name **SHOULD** include an organizational prefix (e.g., `FFV2_ENCODING_ACME_FOOBAR`) to signal that interoperability is limited to implementations licensed by that organization.

Option 1 is **RECOMMENDED** for encoding types intended for broad interoperability. Options 2 and 3 allow vendors to register encoding types for use within their own ecosystems while preserving the encoding type namespace.

The rationale for this requirement is that erasure coding moves computation from the server to the client. If the client cannot determine how data was encoded, it cannot decode it. Unlike layout types (where the server controls the storage format), encoding types require client-side agreement on the mathematics.

5.2. `ffv2_layout4`

5.2.1. ffv2_flags4

```

/// const FFV2_FLAGS_NO_LAYOUTCOMMIT    = FF_FLAGS_NO_LAYOUTCOMMIT;
/// const FFV2_FLAGS_NO_IO_THRU_MDS     = FF_FLAGS_NO_IO_THRU_MDS;
/// const FFV2_FLAGS_NO_READ_IO         = FF_FLAGS_NO_READ_IO;
/// const FFV2_FLAGS_WRITE_ONE_MIRROR    = FF_FLAGS_WRITE_ONE_MIRROR;
/// const FFV2_FLAGS_ONLY_ONE_WRITER     = 0x00000010;
///
/// typedef uint32_t                      ffv2_flags4;

```

Figure 10: The ffv2_flags4

The ffv2_flags4 in Figure 10 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.

FFV2_FLAGS_NO_LAYOUTCOMMIT: can be set to indicate that the client is not required to send LAYOUTCOMMIT to the metadata server.

FFV2_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.

FFV2_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.

FFV2_FLAGS_WRITE_ONE_MIRROR: can be set to indicate that the client only needs to update one of the mirrors (see Section 8.1).

FFV2_FLAGS_ONLY_ONE_WRITER: can be set to indicate that the client only needs to use a CHUNK_WRITE to update the chunks in the data file. I.e., keep the ability to rollback in case of a write hole caused by overwriting. If this flag is not set, then the client MUST write chunks with CHUNK_WRITE with the cwa_guard set in order to prevent collision across the data servers.

5.3. ffv2_file_info4

```

/// struct ffv2_file_info4 {
///     stateid4          fffi_stateid;
///     nfs_fh4           fffi_fh_vers;
/// };

```

Figure 11: The ffv2_file_info4

The `ffv2_file_info4` is a new structure to help with the `stateid` issue discussed in Section 5.1 of [RFC8435]. I.e., in version 1 of the Flexible File Layout Type, there was the singleton `ffv2ds_stateid` combined with the `ffv2ds_fh_vers` array. I.e., each NFSv4 version has its own `stateid`. In Figure 11, each NFSv4 filehandle has a one-to-one correspondence to a `stateid`.

5.4. `ffv2_ds_flags4`

```

/// const FFV2_DS_FLAGS_ACTIVE      = 0x00000001;
/// const FFV2_DS_FLAGS_SPARE       = 0x00000002;
/// const FFV2_DS_FLAGS_PARITY      = 0x00000004;
/// const FFV2_DS_FLAGS_REPAIR      = 0x00000008;
/// typedef uint32_t                ffv2_ds_flags4;

```

Figure 12: The `ffv2_ds_flags4`

The `ffv2_ds_flags4` (in Figure 12) flags details the state of the data servers. With Erasure Coding algorithms, there are both Systematic and Non-Systematic approaches. In the Systematic, the bits for integrity are placed amongst the resulting transformed chunk. Such an implementation would typically see `FFV2_DS_FLAGS_ACTIVE` and `FFV2_DS_FLAGS_SPARE` data servers. The `FFV2_DS_FLAGS_SPARE` ones allow the client to repair a payload without engaging the metadata server. I.e., if one of the `FFV2_DS_FLAGS_ACTIVE` did not respond to a `WRITE_BLOCK`, the client could fail the chunk to the `FFV2_DS_FLAGS_SPARE` data server.

With the Non-Systematic approach, the data and integrity live on different data servers. Such an implementation would typically see `FFV2_DS_FLAGS_ACTIVE` and `FFV2_DS_FLAGS_PARITY` data servers. If the implementation wanted to allow for local repair, it would also use `FFV2_DS_FLAGS_SPARE`.

The `FFV2_DS_FLAGS_REPAIR` flag can be used by the metadata server to inform the client that the indicated data server is a replacement data server as far as existing data is concerned.

```

// Fill in
//
// -- Tom

```

See [Plank97] for further reference to storage layouts for coding.

5.5. `ffv2_data_server4`

```

/// struct ffv2_data_server4 {
///     deviceid4          ffv2ds_deviceid;
///     uint32_t           ffv2ds_efficiency;
///     ffv2_file_info4    ffv2ds_file_info<>;
///     fattr4_owner       ffv2ds_user;
///     fattr4_owner_group ffv2ds_group;
///     ffv2_ds_flags4     ffv2ds_flags;
/// };

```

Figure 13: The ffv2_data_server4

The ffv2_data_server4 (in Figure 13) describes a data file and how to access it via the different NFS protocols.

5.6. ffv2_coding_type_data4

```

/// union ffv2_coding_type_data4 switch
///     (ffv2_coding_type4 fctd_coding) {
///     case FFV2_CODING_MIRRORED:
///         ffv2_data_protection4 fctd_protection;
///     default:
///         ffv2_data_protection4 fctd_protection;
/// };

```

Figure 14: The ffv2_coding_type_data4

The ffv2_coding_type_data4 (in Figure 14) describes the data protection geometry for the layout. All coding types carry an ffv2_data_protection4 (Figure 20) specifying the number of data and parity shards. The coding type enum determines how the shards are encoded; the protection structure determines how many shards there are.

Although the FFV2_CODING_MIRRORED case and the default case currently carry the same type, the union form is intentional. Future revisions of this specification may assign distinct arm types to specific coding types; using a union now avoids an incompatible change to the XDR at that time.

For FFV2_CODING_MIRRORED, fdp_data is 1 and fdp_parity is the number of additional copies (e.g., fdp_parity=2 for 3-way mirroring). Erasure coding types registered in companion documents (e.g., Reed-Solomon Vandermonde, Mojette systematic) use fdp_data >= 2 and fdp_parity >= 1.

```

/// enum ffv2_stripping {
///     FFV2_STRIPING_NONE = 0,
///     FFV2_STRIPING_SPARSE = 1,
///     FFV2_STRIPING_DENSE = 2
/// };
///
/// struct ffv2_stripes4 {
///     ffv2_data_server4      ffs_data_servers<>;
/// };

```

Figure 15: The stripes v2

Each stripe contains a set of data servers in `ffs_data_servers`. If the stripe is part of a `ffv2_coding_type_data4` of `FFV2_CODING_MIRRORED`, then the length of `ffs_data_servers` MUST be 1.

5.7. ffv2_key4

```

/// typedef uint64_t ffv2_key4;

```

Figure 16: The ffv2_key4

The `ffv2_key4` is an opaque 64-bit identifier used to associate a mirror instance with its backing storage key. The value is assigned by the metadata server and is opaque to the client.

5.8. ffv2_mirror4

```

/// struct ffv2_mirror4 {
///     ffv2_coding_type_data4  ffm_coding_type_data;
///     ffv2_key4               ffm_key;
///     ffv2_stripping          ffm_stripping;
///     uint32_t                ffm_stripping_unit_size; // The minimum stripe u
nit size is 64 bytes.
///     uint32_t                ffm_client_id;
///     ffv2_stripes4           ffm_stripes<>; // Length of this array is the s
trip count
/// };

```

Figure 17: The ffv2_mirror4

The `ffv2_mirror4` (in Figure 17) describes the Flexible File Layout Version 2 specific fields. The `ffm_client_id` tells the client which id to use when interacting with the data servers.

The `ffm_coding_type_data` is which encoding type is used by the mirror.

The `ffm_stripping` is which striping method is used by the mirror.

The `ffm_stripping_unit_size` is the stripe unit size used by the mirror. If the value of `ffm_stripping` is `FFV2_STRIPING_NONE`, then the value of `ffm_stripping_unit_size` MUST be 1.

The `ffm_stripes` is the array of stripes for the mirror. If there is no striping or the `ffm_coding_type_data` is `FFV2_CODING_MIRRORED`, then the length of `ffm_stripes` MUST be 1.

```
// Nuke ffm_client_id?  
//  
// -- Tom
```

5.9. `ffv2_layout4`

```
/// struct ffv2_layout4 {  
///     ffv2_mirror4          ffl_mirrors<>;  
///     ffv2_flags4          ffl_flags;  
///     uint32_t              ffl_stats_collect_hint;  
/// };
```

Figure 18: The `ffv2_layout4`

The `ffv2_layout4` (in Figure 18) describes the Flexible File Layout Version 2.

The `ffl_mirrors` field is the array of mirrored storage devices that provide the storage for the current stripe; see Figure 19.

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.

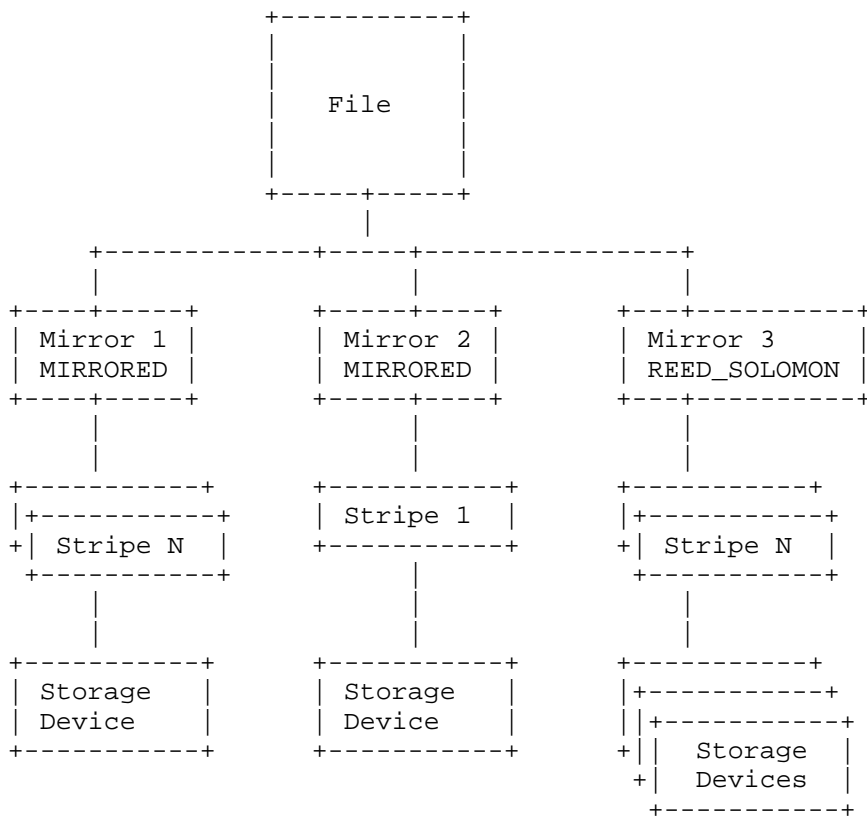


Figure 19: The Relationship between MDS and DSes

As shown in Figure 19 if the `ffm_coding_type_data` is `FFV2_CODING_MIRRORED`, then each of the stripes MUST only have 1 storage device. I.e., the length of `ffs_data_servers` MUST be 1. The other encoding types can have any number of storage devices.

The abstraction here is that for `FFV2_CODING_MIRRORED`, each stripe describes exactly one data server. And for all other encoding types, each of the stripes describes a set of data servers to which the chunks are distributed. Further, the payload length can be different per stripe.

5.10. `ffv2_data_protection4`

```

/// struct ffv2_data_protection4 {
///     uint32_t fdp_data;    /* data shards (k) */
///     uint32_t fdp_parity; /* parity/redundancy shards (m) */
/// };

```

Figure 20: The `ffv2_data_protection4`

The `ffv2_data_protection4` (in Figure 20) describes the data protection geometry as a pair of counts: the number of data shards (`fdp_data`, also known as k) and the number of parity or redundancy shards (`fdp_parity`, also known as m). This structure is used in both layout hints and layout responses, and applies uniformly to all coding types:

Protection Mode	<code>fdp_data</code>	<code>fdp_parity</code>	Total DSes	Description
Mirroring (3-way)	1	2	3	3 copies, no encoding
Striping (6-way)	6	0	6	Parallel I/O, no redundancy
RS Vandermonde 4+2	4	2	6	Tolerates 2 DS failures
Mojette-sys 8+2	8	2	10	Tolerates 2 DS failures

Table 1: Example data protection configurations

By expressing all protection modes as (`fdp_data`, `fdp_parity`) pairs, a single structure serves mirroring, striping, and all erasure coding types. The coding type (Figure 9) determines HOW the shards are encoded; the protection structure determines HOW MANY shards there are.

The total number of data servers required is `fdp_data` + `fdp_parity`. The storage overhead is `fdp_parity` / `fdp_data` (e.g., 50% for 4+2, 25% for 8+2).

5.11. `ffv2_layouthint4`

```

/// struct ffv2_layouthint4 {
///     ffv2_coding_type4      fflh_supported_types<>;
///     ffv2_data_protection4  fflh_preferred_protection;
/// };

```

Figure 21: The `ffv2_layouthint4`

The `ffv2_layouthint4` (in Figure 21) describes the `layout_hint` (see Section 5.12.4 of [RFC8881]) that the client can provide to the metadata server.

The client provides two hints:

`fflh_supported_types` An ordered list of coding types the client supports, with the most preferred type first. The server SHOULD select a type from this list but MAY choose any type it supports. If the server does not support any of the listed types, it returns `NFS4ERR_CODING_NOT_SUPPORTED`, and the client can retry with a different list to discover the overlapping set.

`fflh_preferred_protection` The client's preferred data protection geometry as a (`fdp_data`, `fdp_parity`) pair. The server SHOULD honor this hint but MAY override it based on server-side policy. A server that manages data protection via administrative policy (e.g., per-directory or per-export objectives) will typically ignore this hint and return the geometry dictated by policy.

For example, a client that prefers Mojette systematic with 8+2 protection would send:

```
fflh_supported_types = { FFFV2_CODING_MIRRORED,  
                        FFFV2_ENCODING_MOJETTE_SYSTEMATIC,  
                        FFFV2_ENCODING_RS_VANDERMONDE }  
fflh_preferred_protection = { fdp_data = 8, fdp_parity = 2 }
```

A server with a policy of RS 4+2 for this directory would ignore both hints and return a layout with `FFFV2_ENCODING_RS_VANDERMONDE` and (`fdp_data`=4, `fdp_parity`=2). A server without erasure coding might return `FFFV2_CODING_MIRRORED` with (`fdp_data`=1, `fdp_parity`=2) for 3-way mirroring.

Note: In Figure 18 `ffv2_coding_type_data4` is an enumerated union with the payload of each arm being defined by the protection type. `ffm_client_id` tells the client which id to use when interacting with the data servers.

The `ffv2_layout4` structure (see Figure 18) 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 18, each `layout4` element returned in the `logr_layout` array of `LAYOUTGET4res` (see Section 18.43.2 of [RFC8881]) describes a layout segment. Hence,

each `ffv2_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 `ffm_striping_unit_size` field (inside each `ffv2_mirror4`) is the stripe unit size in use for that mirror. The number of stripes is given by the number of elements in `ffs_data_servers` within each `ffv2_stripes4`. If the number of stripes is one, then the value for `ffm_striping_unit_size` MUST default to zero. The only supported mapping scheme is sparse and is detailed in Section 6. 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 represents an array of state information for each mirrored copy of the current layout segment. Each element is described by a `ffv2_mirror4` type.

`ffv2ds_deviceid` provides the deviceid of the storage device holding the data file.

`ffv2ds_file_info` is an array of `ffv2_file_info4` structures, each pairing a filehandle (`fffi_fh_vers`) with a stateid (`fffi_stateid`). There MUST be exactly as many elements in `ffv2ds_file_info` 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 and stateids for different combinations of version, minor version, and coupling strength. See Section 5.14 for how to handle versioning issues between the client and storage devices.

For tight coupling, `fffi_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 `fffi_stateid` to be the anonymous stateid.

This specification of the `fffi_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

By pairing each `fffi_fh_vers` with its own `fffi_stateid` inside `ffv2_file_info4`, the v2 layout addresses the v1 limitation where a singleton `stateid` was shared across all filehandles. Each open file on the storage device can now have its own `stateid`, eliminating the ambiguity present in the v1 structure.

For loosely coupled storage devices, `ffv2ds_user` and `ffv2ds_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 4.1) is set, then the client MUST ignore both `ffv2ds_user` and `ffv2ds_group`.

The allowed values for both `ffv2ds_user` and `ffv2ds_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.

`ffv2ds_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 8.1.1.

5.11.1. 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.

5.11.2. 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.

5.12. `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]).

5.13. 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 each of the `fffi_fh_vers` values within `ffv2ds_file_info` are different, there are multiple copies of the file for the current layout segment available.

5.14. Handling Version Errors

When the metadata server provides the `ffda_versions` array in the `ff_device_addr4` (see Section 4.1), 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.

6. 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 ffs_data_servers

S: number of bytes in a stripe

S = W * ffm_striping_unit_size

N: stripe number

N = L / S

Figure 22: Stripe Mapping Math

7. 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.

8. Client-Side Protection Modes

8.1. Client-Side 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 19).

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 5.11), 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 8.1.2). 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 8.1.3, during the resilvering, the layout is recalled, and the client has to make modifications via the metadata server.

8.1.1. 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 `ffv2ds_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.

8.1.2. Writing to Mirrors

8.1.2.1. 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 8.1.2.3). The client's responsibility with respect to COMMIT is explained in Section 8.1.2.4. The client may choose any one of the mirrors and may use `ffv2ds_efficiency` as described in Section 8.1.1 when making this choice.

8.1.2.2. 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.

8.1.2.3. 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.

8.1.2.4. 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.

8.1.3. 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.

8.2. Erasure Coding

Erasure Coding takes a data block and transforms it to a payload to send to the data servers (see Figure 23). It generates a metadata header and transformed block per data server. The header is metadata information for the transformed block. From now on, the metadata is simply referred to as the header and the transformed block as the chunk. The payload of a data block is the set of generated headers and chunks for that data block.

The guard is a unique identifier generated by the client to describe the current write transaction (see Section 20.1). The intent is to have a unique and non-opaque value for comparison. The `payload_id` describes the position within the payload. Finally, the `crc32` is the 32 bit crc calculation of the header (with the `crc32` field being 0) and the chunk. By combining the two parts of the payload, integrity is ensured for both the parts.

While the data block might have a length of 4kB, that does not necessarily mean that the length of the chunk is 4kB. That length is determined by the erasure coding type algorithm. For example, Reed Solomon might have 4kB chunks with the data integrity being compromised by parity chunks. Another example would be the Mojette Transformation, which might have 1kB chunk lengths.

The payload contains redundancy which will allow the erasure coding type algorithm to repair chunks in the payload as it is transformed back to a data block (see Figure 28). A payload is consistent when all of the contained headers share the same guard. It has integrity when it is consistent and the combinations of headers and chunks all pass the `crc32` checks.

The erasure coding algorithm itself might not be sufficient to detect errors in the chunks. The `crc32` checks will allow the data server to detect chunks with issues and then the erasure decoding algorithm can reconstruct the missing chunk.

8.2.1. Encoding a Data Block

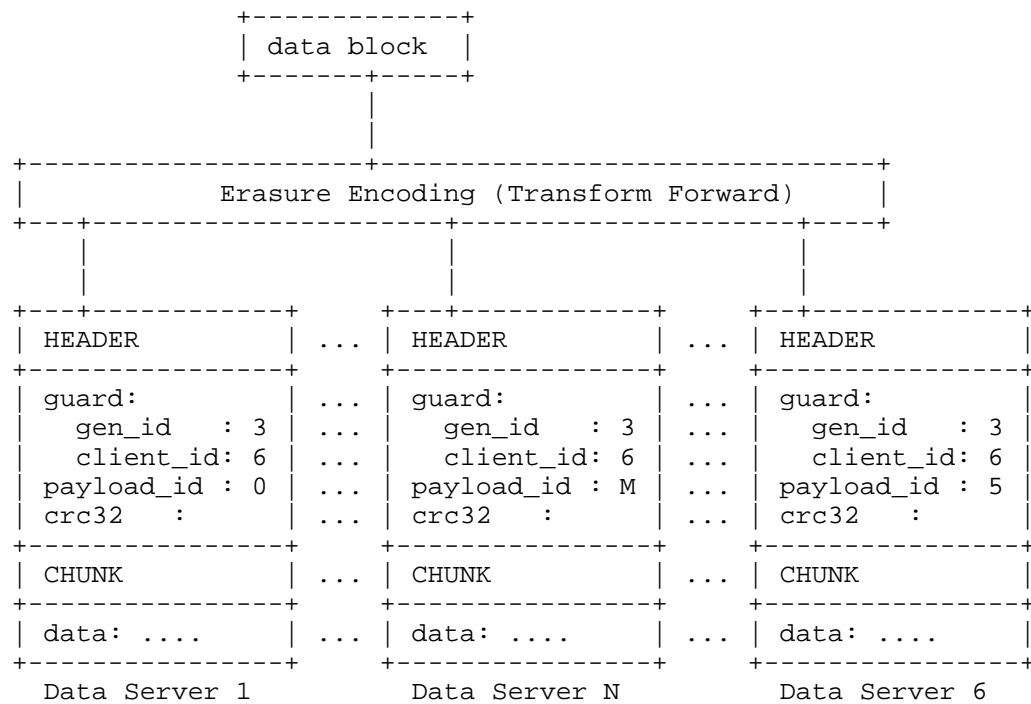


Figure 23: Encoding a Data Block

Each data block of the file resident in the client's cache of the file will be encoded into N different payloads to be sent to the data servers as shown in Figure 23. As CHUNK_WRITE (see Section 21.10) can encode multiple write_chunk4 into a single transaction, a more accurate description of a CHUNK_WRITE is in Figure 24.

```

+-----+
| CHUNK_WRITEargs |
+-----+
| cwa_stateid: 0   |
| cwa_offset: 1    |
| cwa_stable: FILE_SYNC4 |
| cwa_payload_id: 0 |
| cwa_owner:       |
|     co_guard:    |
|         cg_gen_id   : 3 |
|         cg_client_id: 6 |
| cwa_chunk_size   : 1048 |
| cwa_crc32s:      |
|     [0]: 0x32ef89 |
|     [1]: 0x56fa89 |
|     [2]: 0x7693af |
| cwa_chunks   : ..... |
+-----+

```

Figure 24: Example of CHUNK_WRITE_args

This describes a 3 block write of data from an offset of 1 block in the file. As each block shares the `cwa_owner`, it is only presented once. I.e., the data server will be able to construct the header for the *i*'th chunk from the `cwa_chunks` from the `cwa_payload_id`, the `cwa_owner`, and the *i*'th `crc32` from the `cw_crc32s`. The `cwa_chunks` are sent together as a byte stream to increase performance.

Assuming that there were no issues, Figure 25 illustrates the results. The payload sequence id is implicit in the `CHUNK_WRITEargs`.

```

+-----+
| CHUNK_WRITEresok |
+-----+
| cwr_count: 3 |
| cwr_committed: FILE_SYNC4 |
| cwr_writeverf: 0xf1234abc |
| cwr_owners[0]: |
|   co_chunk_id: 1 |
|   co_guard: |
|     cg_gen_id : 3 |
|     cg_client_id: 6 |
| cwr_owners[1]: |
|   co_chunk_id: 2 |
|   co_guard: |
|     cg_gen_id : 3 |
|     cg_client_id: 6 |
| cwr_owners[2]: |
|   co_chunk_id: 3 |
|   co_guard: |
|     cg_gen_id : 3 |
|     cg_client_id: 6 |
+-----+

```

Figure 25: Example of CHUNK_WRITE_res

8.2.1.1. Calculating the CRC32

```

+-----+
| HEADER |
+-----+
| guard: |
|   gen_id : 7 |
|   client_id: 6 |
| payload_id : 0 |
| crc32 : 0 |
+-----+
| CHUNK |
+-----+
| data: .... |
+-----+

```

Data Server 1

Figure 26: CRC32 Before Calculation

Assuming the header and payload as in Figure 26, the `crc32` needs to be calculated in order to fill in the `cw_crc` field. In this case, the `crc32` is calculated over the 4 fields as shown in the header and the `cw_chunk`. In this example, it is calculated to be `0x21de8`. The resulting `CHUNK_WRITE` is shown in Figure 27.

```
+-----+
| CHUNK_WRITEargs |
+-----+
| cwa_stateid: 0   |
| cwa_offset: 1   |
| cwa_stable: FILE_SYNC4 |
| cwa_payload_id: 0 |
| cwa_owner:      |
|     co_guard:   |
|         cg_gen_id   : 7 |
|         cg_client_id: 6 |
| cwa_chunk_size  : 1048 |
| cwa_crc32s:     |
|     [0]: 0x21de8 |
| cwa_chunks   : ..... |
+-----+
```

Figure 27: CRC32 After Calculation

8.2.2. Decoding a Data Block

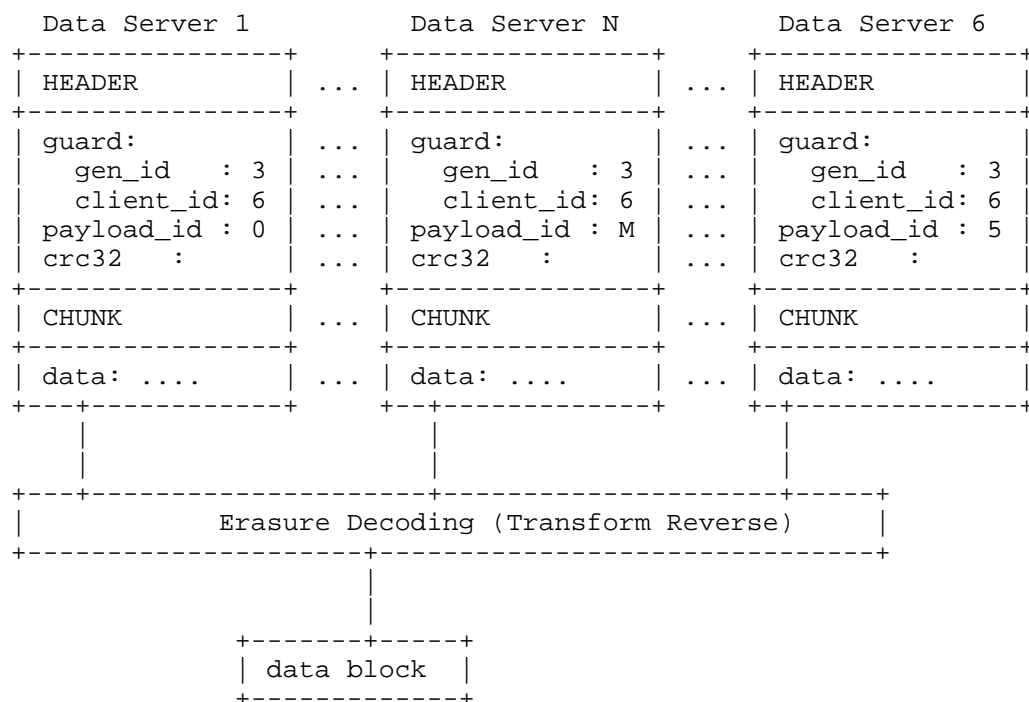


Figure 28: Decoding a Data Block

When reading chunks via a `CHUNK_READ` operation, the client will decode them into data blocks as shown in Figure 28.

At this time, the client could detect issues in the integrity of the data. The handling and repair are out of the scope of this document and MUST be addressed in the document describing each erasure coding type.

8.2.2.1. Checking the CRC32

```

+-----+
| CHUNK_READresok |
+-----+
| crr_eof: false |
| crr_chunks[0]: |
|   cr_crc: 0x21de8 |
|   cr_owner:      |
|     co_guard:    |
|       cg_gen_id  : 7 |
|       cg_client_id: 6 |
|   cr_chunk   : ..... |
+-----+

```

Figure 29: CRC32 on the Wire

Assuming the `CHUNK_READ` results as in Figure 29, the `crc32` needs to be checked in order to ensure data integrity. Conceptually, a header and payload can be built as shown in Figure 30. The `crc32` is calculated over the 4 fields as shown in the header and the `cr_chunk`. In this example, it is calculated to be `0x21de8`. Thus this payload for the data server has data integrity.

```

+---+-----+
| HEADER |
+---+-----+
| guard: |
|   gen_id  : 7 |
|   client_id: 6 |
| payload_id : 0 |
| crc32     : 0 |
+---+-----+
| CHUNK |
+---+-----+
| data:  .... |
+---+-----+

```

Data Server 1

Figure 30: CRC32 Being Checked

8.2.3. Write Modes

There are two basic writing modes for erasure coding and they depend on the metadata server using `FFV2_FLAGS_ONLY_ONE_WRITER` in the `ffl_flags` in the `ffv2_layout4` (see Figure 18) to inform the client whether it is the only writer to the file or not. If it is the only writer, then `CHUNK_WRITE` with the `cwa_guard` not set can be used to write chunks. In this scenario, there is no write contention, but write holes can occur as the client overwrites old data. Thus the

client does not need guarded writes, but it does need the ability to rollback writes. If it is not the only writer, then `CHUNK_WRITE` with the `cwa_guard` set **MUST** be used to write chunks. In this scenario, the write holes can also be caused by multiple clients writing to the same chunk. Thus the client needs guarded writes to prevent overwrites and it does need the ability to rollback writes.

In both modes, clients **MUST NOT** overwrite payloads which already contain inconsistency. This directly follows from Section 8.2.4 and **MUST** be handled as discussed there. Once consistency in the payload has been detected, the client can use those chunks as a basis for read/modify/update.

`CHUNK_WRITE` is a two pass operation in cooperation with `CHUNK_FINALIZE` (Section 21.3) and `CHUNK_ROLLBACK` (Section 21.8). It writes to the data file and the data server is responsible for retaining a copy of the old header and chunk. A subsequent `CHUNK_READ` would return the new chunk. However, until either the `CHUNK_FINALIZE` or `CHUNK_ROLLBACK` is presented, a subsequent `CHUNK_WRITE` **MUST** result in the locking of the chunk, as if a `CHUNK_LOCK` (Section 21.5) had been performed on the chunk. As such, further `CHUNK_WRITES` by any client **MUST** be denied until the chunk is unlocked by `CHUNK_UNLOCK` (Section 21.9).

If the `CHUNK_WRITE` results in a consistent data block, then the client will send a `CHUNK_FINALIZE` in a subsequent compound to inform the data server that the chunk is consistent and can be overwritten by another `CHUNK_WRITE`.

If the `CHUNK_WRITE` results in an inconsistent data block or if the data server returned `NFS4ERR_CHUNK_LOCKED`, then the client sends a `LAYOUTERROR` to the metadata server with a code of `NFS4ERR_PAYLOAD_NOT_CONSISTENT`. The metadata server then selects a client (or data server) to repair the data block.

```
// Since we don't have all potential chunks available, it can either
// chose the winner or pick a random client/data server. If the
// client is the winner, then the process is to use
// CHUNK_WRITE_REPAIR to overwrite the chunks which are not
// consistent. If it is a random client, then the client should just
// CHUNK_ROLLBACK and CHUNK_UNLOCK until it gets back to the original
// chunk.
//
// -- Tom
```

The client which is repairing the chunk can decide to rollback to the previous chunk via `CHUNK_ROLLBACK`. Note that `CHUNK_ROLLBACK` does not unlock the chunk, that has to be explicitly done via `CHUNK_UNLOCK`.

8.2.3.1. Single Writer Mode

In single writer mode, the metadata server sets `FFV2_FLAGS_ONLY_ONE_WRITER` in `ffl_flags`, indicating that no other client holds a write layout for the file. The client sends `CHUNK_WRITE` with `cwa_guard.cwg_check` set to `FALSE`, omitting the guard value. Because only one writer is active, there is no risk of two clients overwriting the same chunk concurrently.

The single writer write sequence is:

1. The client issues `CHUNK_WRITE` (`cwa_guard.cwg_check = FALSE`) for each shard. The data server places the written block in the `PENDING` state and retains a copy of the previous block for rollback.
2. The client issues `CHUNK_FINALIZE` to advance the blocks from `PENDING` to `FINALIZED`, validating the per-block CRC32.
3. The client issues `CHUNK_COMMIT` to advance the blocks from `FINALIZED` to `COMMITTED`, persisting the block metadata to stable storage.

If the client detects an error after `CHUNK_WRITE` but before `CHUNK_FINALIZE` (e.g., a CRC mismatch on a subsequent `CHUNK_READ`), it issues `CHUNK_ROLLBACK` to restore the previous block content. `CHUNK_ROLLBACK` does not lock the chunk; the next `CHUNK_WRITE` is permitted immediately.

8.2.3.2. Repairing Single Writer Payloads

In single writer mode, inconsistent blocks arise from a client or data server failure during a `CHUNK_WRITE` / `CHUNK_FINALIZE` sequence. Because no other writer is active, the repair client is always the original writer (or a substitute designated by the metadata server after lease expiry).

The repair sequence for a single writer payload is:

1. The repair client issues `CHUNK_READ` to identify which blocks are in an inconsistent state (`PENDING` with a CRC mismatch, or in the errored state set by a prior `CHUNK_ERROR`).

2. For each errored block, the repair client reconstructs the correct data using the erasure coding algorithm (RS matrix inversion or Mojette back-projection) from the surviving consistent blocks.
3. The repair client issues `CHUNK_WRITE_REPAIR` (Section 21.11) to write the reconstructed data. `CHUNK_WRITE_REPAIR` bypasses the guard check and applies different data server policies (e.g., allowing writes to blocks in the errored state).
4. The repair client issues `CHUNK_FINALIZE` and `CHUNK_COMMIT` to persist the repaired blocks.
5. The repair client issues `CHUNK_REPAIRED` (Section 21.7) to clear the errored state and make the blocks available for normal reads.

8.2.3.3. Multiple Writer Mode

In multiple writer mode, the metadata server does not set `FFV2_FLAGS_ONLY_ONE_WRITER`, indicating that concurrent writers may hold write layouts for the file. The client sends `CHUNK_WRITE` with `cwa_guard.cwg_check` set to `TRUE`, supplying a `chunk_guard4` in `cwa_guard.cwg_guard` that uniquely identifies this write transaction across all data servers.

The multiple writer write sequence is:

1. The client selects a unique `chunk_guard4` for this transaction. The `cg_client_id` identifies the client (derived from the client's `clientid4`); the `cg_gen_id` is a per-client generation counter incremented for each new transaction.
2. The client issues `CHUNK_WRITE` (`cwa_guard.cwg_check = TRUE`) for each shard. The data server checks that no other client's block is in the `PENDING` state for this chunk. If another client's block is already pending, the data server returns `NFS4ERR_CHUNK_LOCKED` with the `clr_owner` field identifying the lock holder.
3. On `NFS4ERR_CHUNK_LOCKED`, the client MUST back off. It issues `CHUNK_ROLLBACK` for any shards it has already written in this transaction, then retries after a delay.
4. If all `CHUNK_WRITES` succeed, the client issues `CHUNK_FINALIZE` and `CHUNK_COMMIT` as in single writer mode.

The guard ensures that the complete set of shards forming a consistent erasure-coded block all carry the same `chunk_guard4`. A reader that encounters shards with different guard values knows the payload is not yet consistent and MUST either retry or report `NFS4ERR_PAYLOAD_NOT_CONSISTENT`.

8.2.3.4. Repairing Multiple Writer Payloads

In multiple writer mode, inconsistent blocks can arise from two sources: a client failure leaving some shards in `PENDING` state, or two clients writing different data to the same chunk before one has committed.

The metadata server coordinates repair by designating a repair client (identified in the layout via `FFV2_DS_FLAGS_REPAIR` on the target data server). The repair sequence is:

1. The repair client issues `CHUNK_LOCK` (Section 21.5) on the affected block range of each data server. If any lock attempt returns `NFS4ERR_CHUNK_LOCKED`, the repair client records the existing lock holder's `chunk_owner4` and proceeds; the lock holder's data is a candidate for the winning payload.
2. The repair client issues `CHUNK_READ` on all data servers to retrieve the current payload. It examines the `chunk_owner4` of each shard to identify which transaction (if any) produced a consistent set across all `k` data shards.
3. If a consistent set is found (all `k` data shards carry the same `chunk_guard4`), that payload is the winner. The repair client issues `CHUNK_WRITE_REPAIR` to copy the winner's data to any data servers whose shard is inconsistent, followed by `CHUNK_FINALIZE` and `CHUNK_COMMIT`.
4. If no consistent set exists (all available payloads are partial), the repair client selects one transaction's payload as authoritative (typically the one with the most complete set of shards, or the most recent `cg_gen_id`) and proceeds as above.
5. After all data servers carry consistent, finalized, committed data, the repair client issues `CHUNK_REPAIRED` to clear the errored state and `CHUNK_UNLOCK` to release the locks acquired in step 1.
6. The repair client reports success to the metadata server via `LAYOUTRETURN`.

8.2.4. Reading Chunks

The client reads chunks from the data file via `CHUNK_READ`. The number of chunks in the payload that need to be consistent depend on both the Erasure Coding Type and the level of protection selected. If the client has enough consistent chunks in the payload, then it can proceed to use them to build a data block. If it does not have enough consistent chunks in the payload, then it can either decide to return a `LAYOUTERROR` of `NFS4ERR_PAYLOAD_NOT_CONSISTENT` to the metadata server or it can retry the `CHUNK_READ` until there are enough consistent chunks in the payload.

As another client might be writing to the chunks as they are being read, it is entirely possible to read the chunks while they are not consistent. As such, it might even be the non-consistent chunks which contain the new data and a better action than building the data block is to retry the `CHUNK_READ` to see if new chunks are overwritten.

8.2.5. Whole File Repair

```
// Describe how a repair client can be assigned with missing
// FFV2_DS_FLAGS_ACTIVE data servers and a number of
// FFV2_DS_FLAGS_REPAIR data servers. Then the client will either
// move chunks from FFV2_DS_FLAGS_SPARE data servers to the
// FFV2_DS_FLAGS_REPAIR data servers or reconstruct the chunks for
// the FFV2_DS_FLAGS_REPAIR based on the decoded data blocks, The
// client indicates success by returning the layout.
//
// -- Tom

// For a slam dunk, introduce the concept of a proxy repair client.
// I.e., the client appears as a single FFV2_CODING_MIRRORED file to
// other clients. As it receives WRITES, it encodes them to the real
// set of data servers. As it receives READs, it decodes them from
// the real set of data servers. Once the proxy repair is finished,
// the metadata server will start pushing out layouts for the real
// set of data servers.
//
// -- Tom
```

8.3. Mixing of Coding Types

Multiple coding types can be present in a Flexible File Version 2 Layout Type layout. The `ffv2_layout4` has an array of `ffv2_mirror4`, each of which has a `ffv2_coding_type4`. The main reason to allow for this is to provide for either the assimilation of a non-erasure coded file to an erasure coded file or the exporting of an erasure coded file to a non-erasure coded file.

Assume there is an additional `ffv2_coding_type4` of `FFV2_CODING_REED_SOLOMON` and it needs 8 active chunks. The user wants to actively assimilate a regular file. As such, a layout might be as represented in Figure 31. As this is an assimilation, most of the data reads will be satisfied by `READ` (see Section 18.22 of [RFC8881]) calls to index 0. However, as this is also an active file, there could also be `CHUNK_READ` (see Section 21.6) calls to the other indexes.

```
+-----+
| ffv2_layout4:                                     |
+-----+
|   ffl_mirrors[0]:                                |
|     ffs_data_servers:                            |
|       ffv2_data_server4[0]                      |
|         ffv2ds_flags: 0                          |
|       ffm_coding: FFV2_CODING_MIRRORED           |
+-----+
|   ffl_mirrors[1]:                                |
|     ffs_data_servers:                            |
|       ffv2_data_server4[0]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_ACTIVE      |
|       ffv2_data_server4[1]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_ACTIVE      |
|       ffv2_data_server4[2]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_ACTIVE      |
|       ffv2_data_server4[3]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_ACTIVE      |
|       ffv2_data_server4[4]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_PARITY       |
|       ffv2_data_server4[5]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_PARITY       |
|       ffv2_data_server4[6]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_SPARE        |
|       ffv2_data_server4[7]                      |
|         ffv2ds_flags: FFV2_DS_FLAGS_SPARE        |
|       ffm_coding: FFV2_CODING_REED_SOLOMON       |
+-----+
```

Figure 31: Example of Mixed Coding Types in a Layout

When performing I/O via a FFFV2_CODING_MIRRORED coding type, the non-transformed data will be used, Whereas with other coding types, a metadata header and transformed block will be sent. Further, when reading data from the instance files, the client MUST be prepared to have one of the coding types supply data and the other type not to supply data. I.e., the CHUNK_READ call to the data servers in mirror 1 might return rlr_eof set to true (see Figure 65), which indicates that there is no data, where the READ call to the data server in mirror 0 might return eof to be false, which indicates that there is data. The client MUST determine that there is in fact data. An example use case is the active assimilation of a file to ensure integrity. As the client is helping to translated the file to the new coding scheme, it is actively modifying the file. As such, it might be sequentially reading the file in order to translate. The READ calls to mirror 0 would be returning data and the CHUNK_READ calls to mirror 1 would not be returning data. As the client overwrites the file, the WRITE call and CHUNK_WRITE call would have data sent to all of the data servers. Finally, if the client reads back a section which had been modified earlier, both the READ and CHUNK_READ calls would return data.

8.4. Reed-Solomon Vandermonde Encoding (FFFV2_ENCODING_RS_VANDERMONDE)

8.4.1. Overview

Reed-Solomon (RS) codes are Maximum Distance Separable (MDS) codes: for a $(k+m, k)$ code, any k of the $k+m$ encoded shards suffice to recover the original data. The code tolerates the simultaneous loss of up to m shards.

8.4.2. Galois Field Arithmetic

All RS operations are performed over $GF(2^8)$, the Galois field with 256 elements. Each element is represented as a byte.

Irreducible Polynomial The field is constructed using the irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$ (0x11d in hexadecimal). The primitive element (generator) is $g = 2$, which has multiplicative order 255.

Addition Addition in $GF(2^8)$ is bitwise XOR.

Multiplication Multiplication uses log/antilog tables. For non-zero elements a and b : $a * b = \exp(\log(a) + \log(b))$, where the exp table is doubled to 512 entries to avoid modular reduction on the index sum.

These are the classical constructions from Berlekamp (1968) and Peterson & Weldon (1972). The log/antilog table approach for $GF(2^8)$ multiplication predates all known patents on SIMD-accelerated GF arithmetic. Implementors considering SIMD acceleration of $GF(2^8)$ operations should be aware of US Patent 8,683,296 (StreamScale), which covers certain SIMD-based GF multiplication techniques.

8.4.3. Encoding Matrix

The encoding process uses a $(k+m) \times k$ Vandermonde matrix, normalized so that its top k rows form the identity matrix:

1. Construct a $(k+m) \times k$ Vandermonde matrix V where $V[i][j] = j^i$ in $GF(2^8)$.
2. Extract the top $k \times k$ sub-matrix T from V .
3. Compute $T_{inv} = T^{-1}$ using Gaussian elimination in $GF(2^8)$.
4. Multiply: $E = V * T_{inv}$. The result has an identity block on top (rows 0 through $k-1$) and the parity generation matrix P on the bottom (rows k through $k+m-1$).

The identity block makes the code systematic: data shards pass through unchanged, and only the parity sub-matrix P is needed during encoding.

8.4.4. Encoding

Given k data shards, each of `shard_len` bytes, encoding produces m parity shards, each also `shard_len` bytes:

For each byte position j in $[0, \text{shard_len})$:

For each parity shard i in $[0, m)$:

`parity[i][j] = sum over s in [0, k) of P[i][s] * data[s][j]`

where the sum and product are in $GF(2^8)$. All shards (data and parity) are the same size.

8.4.5. Decoding

When one or more shards are lost (up to m), reconstruction proceeds by matrix inversion:

1. Select k available shards (from the $k+m$ total).
2. Form a $k \times k$ sub-matrix S of the encoding matrix E by selecting the rows corresponding to the available shards.

3. Compute $S_{\text{inv}} = S^{-1}$ using Gaussian elimination in $\text{GF}(2^8)$.
4. Multiply S_{inv} by the vector of available shard data at each byte position to recover the original k data shards.
5. If any parity shards are also missing, regenerate them by re-encoding from the recovered data shards.

The reconstruction cost is dominated by the matrix inversion, which is $O(k^2)$ in $\text{GF}(2^8)$ multiplications.

8.4.6. RS Interoperability Requirements

For two implementations of `FFV2_ENCODING_RS_VANDERMONDE` to interoperate, they MUST agree on all of the following parameters. Any deviation produces a different encoding matrix and renders data unrecoverable by a different implementation.

- * Irreducible polynomial: $x^8 + x^4 + x^3 + x^2 + 1$ (0x11d)
- * Primitive element: $g = 2$
- * Vandermonde evaluation points: $V[i][j] = j^i$ in $\text{GF}(2^8)$
- * Matrix normalization: $E = V * (V[0..k-1])^{-1}$

These four parameters fully determine the encoding matrix for any (k, m) configuration.

8.4.7. RS Shard Sizes

All RS shards (data and parity) are exactly `shard_len` bytes. This simplifies the `CHUNK` operation protocol: `chunk_size` is exactly the shard size for all mirrors.

Configuration	File Size	Shard Size	Total Storage	Overhead
4+2	4 KB	1 KB	6 KB	50%
4+2	1 MB	256 KB	1.5 MB	50%
8+2	4 KB	512 B	5 KB	25%
8+2	1 MB	128 KB	1.25 MB	25%

Table 2: RS shard sizes for common configurations

8.5. Mojette Transform Encoding (FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC)

8.5.1. Overview

The Mojette Transform is an erasure coding technique based on discrete geometry rather than algebraic field operations. It computes 1D projections of a 2D grid along selected directions. Given enough projections, the original grid can be reconstructed exactly.

The transform operates on unsigned integer elements using modular addition. The element size is an implementation choice: 128-bit elements leverage SSE SIMD instructions; 64-bit elements are compatible with NEON and AVX2 vector widths. No Galois field operations are required.

8.5.2. Grid Structure

Data is arranged as a $P \times Q$ grid of unsigned integer elements, where P is the number of columns and Q is the number of rows. For k data shards of S bytes each with W -byte elements:

$$P = S / W \quad (\text{columns per row})$$

$$Q = k \quad (\text{rows = data shards})$$

8.5.3. Directions

A direction is a pair of coprime integers (p_i, q_i) . Implementations SHOULD use $q_i = 1$ for all directions [PARREIN]. For $n = k + m$ total shards, n directions are generated with non-zero p values symmetric around zero:

* For $n = 4$: $p = \{-2, -1, 1, 2\}$

* For $n = 6$: $p = \{-3, -2, -1, 1, 2, 3\}$

8.5.4. Forward Transform (Encoding)

For each direction (p_i, q_i) , the forward transform computes a 1D projection. Each bin sums the grid elements along a discrete line:

Projection(b, p, q) = SUM over all (row, col) where
 $col * p - row * q + offset = b$
 of Grid[row][col]

The number of bins B in a projection is:

$$B(p, q, P, Q) = |p| * (Q - 1) + |q| * (P - 1) + 1$$

For $q = 1$, this simplifies to:

$$B = \text{abs}(p) * (Q - 1) + P$$

The byte size of the projection is $B * W$.

8.5.5. Katz Reconstruction Criterion

Reconstruction is possible if and only if the Katz criterion [KATZ] holds:

$$\text{SUM}(i=1..n) |q_i| \geq Q \quad \text{OR} \quad \text{SUM}(i=1..n) |p_i| \geq P$$

When all $q_i = 1$, the q -sum simplifies to $n \geq Q$.

8.5.6. Inverse Transform (Decoding)

The inverse uses the corner-peeling algorithm:

1. Count how many unknown elements contribute to each bin.
2. Find any bin with exactly one contributor (singleton).
3. Recover the element, subtract from all projections.
4. Repeat until all elements are recovered.

The algorithm is $O(n * P * Q)$.

8.5.7. Systematic Mojette

In the systematic form (FFV2_ENCODING_MOJETTE_SYSTEMATIC), the first k shards are the original data rows and the remaining m shards are projections. Healthy reads require no decoding.

Reconstruction of missing data rows:

1. Load available parity projections.
2. Subtract contributions of present data rows (residual).
3. Corner-peel the residual to recover missing rows.

Reconstruction cost is $O(m * k)$ — a fundamental advantage over RS at wide geometries ($k \geq 8$).

8.5.8. Non-Systematic Mojette

In the non-systematic form (FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC), all $k + m$ shards are projections. Every read requires the full inverse transform. This provides constant performance regardless of failure count, but at higher baseline read cost than systematic.

8.5.9. Mojette Shard Sizes

Unlike RS, Mojette parity shard sizes vary by direction:

Direction (p, q)	Bins (B) for P=512, Q=4	Size (bytes, 64-bit elements)
(-3, 1)	521	4168
(-2, 1)	518	4144
(-1, 1)	515	4120
(1, 1)	515	4120
(2, 1)	518	4144
(3, 1)	521	4168

Table 3: Mojette projection sizes for 4+2, 4KB shards, 64-bit elements

When using CHUNK operations, the `chunk_size` is a nominal stride; the last chunk in a parity shard MAY be shorter than the stride.

8.6. Comparison of Encoding Types

Property	Reed-Solomon	Mojette Systematic	Mojette Non-Systematic
MDS guarantee	Yes	Yes (Katz)	Yes (Katz)
Shard sizes	Uniform	Variable	Variable
Reconstruction cost	$O(k^2)$	$O(m * k)$	$O(m * k)$
Healthy read cost	Zero	Zero	Full decode
GF operations	Yes ($GF(2^8)$)	No	No
Recommended k	$k \leq 6$	$k \geq 4$	Archive only

Table 4: Comparison of erasure encoding types

At small k ($k \leq 6$), RS is the conservative choice with uniform shard sizes. At wider geometries ($k \geq 8$), systematic Mojette offers lower reconstruction cost. Non-systematic Mojette is suitable only for archive workloads where reads are infrequent.

8.7. Handling write holes

A write hole occurs when a client begins writing a stripe but does not successfully write all $k+m$ shards before a failure. Some data servers will hold new data while others still hold old data, producing an inconsistent payload.

The `CHUNK_WRITE` / `CHUNK_ROLLBACK` mechanism addresses this. When a client issues `CHUNK_WRITE`, the data server retains a copy of the previous shard and places the new data in the `PENDING` state. If any shard write fails, the client issues `CHUNK_ROLLBACK` to each data server that received a `CHUNK_WRITE`, restoring the previous content. The payload remains consistent from the reader's perspective throughout, because `PENDING` blocks carry the new `chunk_guard4` value and `CHUNK_READ` returns the last `COMMITTED` or `FINALIZED` block when a `PENDING` block exists.

In the multiple writer model, a write hole can also arise when two clients are racing. The `chunk_guard4` value on each shard identifies which transaction wrote it. A reader that finds shards with different guard values detects the inconsistency and either retries (if a concurrent write is still in progress) or reports `NFS4ERR_PAYLOAD_NOT_CONSISTENT` to the metadata server to trigger repair.

9. NFSv4.2 Operations Allowed to Data Files

```
// In Flexible File Version 1 Layout Type, the emphasis was on NFSv3
// DSes. We limited the operations that clients could send to data
// files to be COMMIT, READ, and WRITE. We further limited the MDS
// to GETATTR, SETATTR, CREATE, and REMOVE. (Funny enough, this is
// not mandated here.) We need to call this out in this draft and
// also we need to limit the NFSv4.2 operations. Besides the ones
// created here, consider: READ, WRITE, and COMMIT for mirroring
// types and ALLOCATE, CLONE, COPY, DEALLOCATE, GETFH, PUTFH,
// READ_PLUS, RESTOREFH, SAVEFH, SEEK, and SEQUENCE for all types.
//
// -- Tom

// Of special merit is SETATTR. Do we want to allow the clients to
// be able to truncate the data files? Which also brings up
// DEALLOCATE. Perhaps we want CHUNK_DEALLOCATE? That way we can
// swap out chunks with the client file. CHUNK_DEALLOCATE_GUARD.
// Really need to determine capabilities of XFS swap!
//
// -- Tom
```

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 32).

```

/* 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 32: 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

10.1.1.1. ff_ioerr4

```
/// struct ffv2_ioerr4 {  
///     offset4      ffie_offset;  
///     length4      ffie_length;  
///     stateid4     ffie_stateid;  
///     device_error4 ffie_errors<>;  
/// };  
///
```

Figure 33: ff_ioerr4

Recall that [RFC7862] defines device_error4 as in Figure 34:

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

Figure 34: 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 ffv2_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 35: ff_io_latency4

Both operation counts and bytes transferred are kept in the `ff_io_latency4` (see Figure 35). 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 ffv2_layoutupdate4 {
///     netaddr4      ffl_addr;
///     nfs_fh4       ffl_fhandle;
///     ffv2_io_latency4 ffl_read;
///     ffv2_io_latency4 ffl_write;
///     nfstime4      ffl_duration;
///     bool          ffl_local;
/// };
///

```

Figure 36: 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 ffv2_iostats4 {
///     offset4      ffis_offset;
///     length4      ffis_length;
///     stateid4     ffis_stateid;
///     io_info4     ffis_read;
///     io_info4     ffis_write;
///     deviceid4    ffis_deviceid;
///     ffv2_layoutupdate4 ffis_layoutupdate;
/// };
///

```

Figure 37: ff_iostats4

[RFC7862] defines io_info4 as in Figure 37.

```

struct io_info4 {
    uint64_t    ii_count;
    uint64_t    ii_bytes;
};

```

Figure 38: 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 ffv2_layoutreturn4 {
///     ffv2_ioerr4      fflr_ioerr_report<>;
///     ffv2_iostats4    fflr_iostats_report<>;
/// };
///
```

Figure 39: `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 40.

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

Figure 40: `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 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 41: ff_layouthint4 (v1 compatibility)

The ff_layouthint4 is retained for backwards compatibility with Flex Files v1 layouts. For Flex Files v2 layouts, clients SHOULD use ffv2_layouthint4 (Figure 21) instead, which provides coding type selection and data protection geometry hints via ffv2_data_protection4 (Figure 20).

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 8.1.3.

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 42: RCA4 masks for v2

```

struct CB_RECALL_ANY4args {
    uint32_t      craa_layouts_to_keep;
    bitmap4       craa_type_mask;
};

```

Figure 43: 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 44.

```

/// enum ffv2_cb_recall_any_mask {
///     PNFS_FF_RCA4_TYPE_MASK_READ = 20,
///     PNFS_FF_RCA4_TYPE_MASK_RW   = 21
/// };
///

```

Figure 44: 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. New NFSv4.2 Error Values

```

///
/// /* Erasure Coding error constants; added to nfsstat4 enum */
///
/// const NFS4ERR_CODING_NOT_SUPPORTED      = 10097;
/// const NFS4ERR_PAYLOAD_NOT_CONSISTENT    = 10098;
/// const NFS4ERR_CHUNK_LOCKED              = 10099;
/// const NFS4ERR_CHUNK_GUARDED             = 10100;
///

```

Figure 45: Errors XDR

The new error codes are shown in Figure 45.

17.1. Error Definitions

Error	Number	Description
NFS4ERR_CODING_NOT_SUPPORTED	10097	Section 17.1.1
NFS4ERR_PAYLOAD_NOT_CONSISTENT	10098	Section 17.1.2
NFS4ERR_CHUNK_LOCKED	10099	Section 17.1.3
NFS4ERR_CHUNK_GUARDED	10100	Section 17.1.4

Table 5: X

17.1.1. NFS4ERR_CODING_NOT_SUPPORTED (Error Code 10097)

The client requested a `ffv2_coding_type4` which the metadata server does not support. I.e., if the client sends a `layout_hint` requesting an erasure coding type that the metadata server does not support, this error code can be returned. The client might have to send the `layout_hint` several times to determine the overlapping set of supported erasure coding types.

17.1.2. NFS4ERR_PAYLOAD_NOT_CONSISTENT (Error Code 10098)

The client encountered a payload in which the blocks were inconsistent and stays inconsistent. As the client can not tell if another client is actively writing, it informs the metadata server of this error via `LAYOUTERROR`. The metadata server can then arrange for repair of the file.

17.1.3. NFS4ERR_CHUNK_LOCKED (Error Code 10099)

The client tried an operation on a chunk which resulted in the data server reporting that the chunk was locked. The client will then inform the metadata server of this error via LAYOUTERROR. The metadata server can then arrange for repair of the file.

17.1.4. NFS4ERR_CHUNK_GUARDED (Error Code 10100)

The client tried a guarded CHUNK_WRITE on a chunk which did not match the guard on the chunk in the data file. As such, the CHUNK_WRITE was rejected and the client should refresh the chunk it has cached.

```
// This really points out either we need an array of errors in the
// chunk operation responses or we need to not send an array of
// chunks in the requests. The arrays were picked in order to reduce
// the header to data cost, but really do not make sense.
//
// -- Tom

// Trying out an array of errors.
//
// -- Tom
```

17.2. Operations and Their Valid Errors

The operations and their valid errors are presented in Table 6. All error codes not defined in this document are defined in Section 15 of [RFC8881] and Section 11 of [RFC7862].

Operation	Errors
CHUNK_COMMIT	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_ERROR	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_FINALIZE	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO,

	NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_HEADER_READ	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_IO, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_LOCK	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_CHUNK_LOCKED, NFS4ERR_INVALID, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_READ	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_IO, NFS4ERR_NOTSUPP, NFS4ERR_PAYLOAD_NOT_CONSISTENT, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_REPAIRED	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_INVALID, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_ROLLBACK	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_INVALID, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_UNLOCK	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_INVALID, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_WRITE	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_CHUNK_GUARDED, NFS4ERR_CHUNK_LOCKED, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_IO, NFS4ERR_NOSPC, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_WRITE_REPAIR	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_IO, NFS4ERR_NOSPC, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE

Table 6: Operations and Their Valid Errors

17.3. Callback Operations and Their Valid Errors

The callback operations and their valid errors are presented in Table 7. All error codes not defined in this document are defined in Section 15 of [RFC8881] and Section 11 of [RFC7862].

Callback Operation	Errors
CB_CHUNK_REPAIR	NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DEADSESSION, NFS4ERR_DELAY, NFS4ERR_CODING_NOT_SUPPORTED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_ISDIR, NFS4ERR_LOCKED, NFS4ERR_NOTSUPP, NFS4ERR_OLD_STATEID, NFS4ERR_SERVERFAULT, NFS4ERR_STALE,

Table 7: Callback Operations and Their Valid Errors

17.4. Errors and the Operations That Use Them

The operations and their valid errors are presented in Table 8. All operations not defined in this document are defined in Section 18 of [RFC8881] and Section 15 of [RFC7862].

Error	Operations
NFS4ERR_CODING_NOT_SUPPORTED	CB_CHUNK_REPAIR, LAYOUTGET

Table 8: Errors and the Operations That Use Them

18. EXCHGID4_FLAG_USE_PNFS_DS

```
/// const EXCHGID4_FLAG_USE_ERASURE_DS      = 0x00100000;
```

Figure 46: The EXCHGID4_FLAG_USE_PNFS_DS

When a data server connects to a metadata server it can via `EXCHANGE_ID` (see Section 18.35 of [RFC8881]) state its pNFS role. The data server can use `EXCHGID4_FLAG_USE_ERASURE_DS` (see Figure 46) to indicate that it supports the new NFSv4.2 operations introduced in this document. Section 13.1 of [RFC8881] describes the interaction of the various pNFS roles masked by `EXCHGID4_FLAG_MASK_PNFS`. However, that does not mask out `EXCHGID4_FLAG_USE_ERASURE_DS`. I.e., `EXCHGID4_FLAG_USE_ERASURE_DS` can be used in combination with all of the pNFS flags.

If the data server sets `EXCHGID4_FLAG_USE_ERASURE_DS` during the `EXCHANGE_ID` operation, then it MUST support all of the operations in Table 9. Further, this support is orthogonal to the Erasure Coding Type selected. The data server is unaware of which type is driving the I/O.

19. New NFSv4.2 Attributes

19.1. Attribute 89: `fattr4_coding_block_size`

```
/// typedef uint64_t                fattr4_coding_block_size;
///
/// const FATTR4_CODING_BLOCK_SIZE  = 89;
///
```

Figure 47: XDR for `fattr4_coding_block_size`

The new attribute `fattr4_coding_block_size` (see Figure 47) is an OPTIONAL to NFSv4.2 attribute which MUST be supported if the metadata server supports the Flexible File Version 2 Layout Type. By querying it, the client can determine the data block size it is to use when coding the data blocks to chunks.

20. New NFSv4.2 Common Data Structures

20.1. `chunk_guard4`

```
/// struct chunk_guard4 {
///     uint32_t    cg_gen_id;
///     uint32_t    cg_client_id;
/// };
```

Figure 48: XDR for `chunk_guard4`

The `chunk_guard4` (see Figure 48) is effectively a 64 bit value, with the upper 32 bits, `cg_gen_id`, being the current generation id of the chunk on the DS and the lower 32 bits, `cg_client_id`, being an unique id established when the client did the `EXCHANGE_ID` operation (see

Section 18.35 of [RFC8881]) with the metadata server. The lower 32 bits are set passed back in the LAYOUTGET operation (see Section 18.43 of [RFC8881]) as the `ffm_client_id` (see Section 5.8).

20.2. `chunk_owner4`

```
/// struct chunk_owner4 {
///     chunk_guard4    co_guard;
///     uint32_t        co_id;
/// };
```

Figure 49: XDR for `chunk_owner4`

The `chunk_owner4` (see Figure 49) is used to determine when and by whom a block was written. The `co_id` is used to identify the block and MUST be the index of the chunk within the file. I.e., it is the offset of the start of the chunk divided by the chunk length. The `co_guard` is a `chunk_guard4` (see Section 20.1), used to identify a given transaction.

The `co_guard` is like the change attribute (see Section 5.8.1.4 of [RFC8881]) in that each chunk write by a given client has to have a unique `co_guard`. I.e., it can be determined which transaction across all data files that a chunk corresponds.

21. New NFSv4.2 Operations

```
///
/// /* New operations for Erasure Coding start here */
///
/// OP_CHUNK_COMMIT          = 77,
/// OP_CHUNK_ERROR           = 78,
/// OP_CHUNK_FINALIZE        = 79,
/// OP_CHUNK_HEADER_READ     = 80,
/// OP_CHUNK_LOCK            = 81,
/// OP_CHUNK_READ            = 82,
/// OP_CHUNK_REPAIRED        = 83,
/// OP_CHUNK_ROLLBACK        = 84,
/// OP_CHUNK_UNLOCK          = 85,
/// OP_CHUNK_WRITE           = 86,
/// OP_CHUNK_WRITE_REPAIR    = 87,
///
```

Figure 50: Operations XDR

Operation	Number	Target Server	Description
CHUNK_COMMIT	77	DS	Section 21.1
CHUNK_ERROR	78	DS	Section 21.2
CHUNK_FINALIZE	79	DS	Section 21.3
CHUNK_HEADER_READ	80	DS	Section 21.4
CHUNK_LOCK	81	DS	Section 21.5
CHUNK_READ	82	DS	Section 21.6
CHUNK_REPAIRED	83	DS	Section 21.7
CHUNK_ROLLBACK	84	DS	Section 21.8
CHUNK_UNLOCK	85	DS	Section 21.9
CHUNK_WRITE	86	DS	Section 21.10
CHUNK_WRITE_REPAIR	87	DS	Section 21.11

Table 9: Protocol OPs

21.1. Operation 77: CHUNK_COMMIT - Activate Cached Chunk Data

21.1.1. ARGUMENTS

```

/// struct CHUNK_COMMIT4args {
///     /* CURRENT_FH: file */
///     offset4      cca_offset;
///     count4       cca_count;
///     chunk_owner4 cca_chunks<>;
/// };

```

Figure 51: XDR for CHUNK_COMMIT4args

21.1.2. RESULTS

```

/// struct CHUNK_COMMIT4resok {
///     verifier4      ccr_writeverf;
///     nfsstat4       ccr_status<>;
/// };

```

Figure 52: XDR for CHUNK_COMMIT4resok

```

/// union CHUNK_COMMIT4res switch (nfsstat4 ccr_status) {
///     case NFS4_OK:
///         CHUNK_COMMIT4resok    ccr_resok4;
///     default:
///         void;
/// };

```

Figure 53: XDR for CHUNK_COMMIT4res

21.1.3. DESCRIPTION

CHUNK_COMMIT is COMMIT (see Section 18.3 of [RFC8881]) with additional semantics over the chunk_owner activating the blocks. As such, all of the normal semantics of COMMIT directly apply.

The main difference between the two operations is that CHUNK_COMMIT works on blocks and not a raw data stream. As such cca_offset is the starting block offset in the file and not the byte offset in the file. Some erasure coding types can have different block sizes depending on the block type. Further, cca_count is a count of blocks to activate and not bytes to activate.

Further, while it may appear that the combination of cca_offset and cca_count are redundant to cca_chunks, the purpose of cca_chunks is to allow the data server to differentiate between potentially multiple pending blocks.

```

// Describe how CHUNK_COMMIT and CHUNK_FINALIZE interact. How does
// CHUNK_COMMIT interact with a locked chunk?
//
// -- Tom

```

21.2. Operation 78: CHUNK_ERROR - Report Error on Cached Chunk Data

21.2.1. ARGUMENTS

```

/// struct CHUNK_ERROR4args {
///     /* CURRENT_FH: file */
///     stateid4      cea_stateid;
///     offset4       cea_offset;
///     count4        cea_count;
///     nfsstat4      cea_error;
///     chunk_owner4  cea_owner;
/// };

```

Figure 54: XDR for CHUNK_ERROR4args

21.2.2. RESULTS

```

/// struct CHUNK_ERROR4res {
///     nfsstat4          cer_status;
/// };

```

Figure 55: XDR for CHUNK_ERROR4res

21.2.3. DESCRIPTION

CHUNK_ERROR allows a client to report that one or more chunks at the specified block range are in error. The cea_offset is the starting block offset and cea_count is the number of blocks affected. The cea_error indicates the type of error detected (e.g., NFS4ERR_PAYLOAD_NOT_CONSISTENT for a CRC mismatch).

The data server records the error state for the affected blocks. Once marked as errored, the blocks are not returned by CHUNK_READ until they are repaired via CHUNK_WRITE_REPAIR (Section 21.11) and the repair is confirmed via CHUNK_REPAIRED (Section 21.7).

The client SHOULD report errors via CHUNK_ERROR before reporting them to the metadata server via LAYOUTERROR. This allows the data server to prevent other clients from reading corrupt data while the metadata server coordinates repair.

21.3. Operation 79: CHUNK_FINALIZE - Transition Chunks from Pending to Finalized

21.3.1. ARGUMENTS

```

/// struct CHUNK_FINALIZE4args {
///     /* CURRENT_FH: file */
///     offset4          cfa_offset;
///     count4           cfa_count;
///     chunk_owner4     cfa_chunks<>;
/// };

```

Figure 56: XDR for CHUNK_FINALIZE4args

21.3.2. RESULTS

```

/// struct CHUNK_FINALIZE4resok {
///     verifier4        cfr_writeverf;
///     nfsstat4         cfr_status<>;
/// };

```

Figure 57: XDR for CHUNK_FINALIZE4resok

```

/// union CHUNK_FINALIZE4res switch (nfsstat4 cfr_status) {
///     case NFS4_OK:
///         CHUNK_FINALIZE4resok    cfr_resok4;
///     default:
///         void;
/// };

```

Figure 58: XDR for CHUNK_FINALIZE4res

21.3.3. DESCRIPTION

CHUNK_FINALIZE transitions blocks from the PENDING state (set by CHUNK_WRITE) to the FINALIZED state. A finalized block is visible to the owning client for reads and is eligible for CHUNK_COMMIT.

The cfa_offset is the starting block offset and cfa_count is the number of blocks to finalize. The cfa_chunks array lists the chunk_owner4 entries whose blocks are to be finalized. Each owner's blocks at the specified offsets MUST be in the PENDING state; if not, the corresponding entry in the per-owner status array ccr_status is set to NFS4ERR_INVAL.

CHUNK_FINALIZE serves as the CRC validation checkpoint: the data server SHOULD have validated the CRC32 of each block at CHUNK_WRITE time. After CHUNK_FINALIZE, the block metadata (CRC, owner, state) is persisted to stable storage so that it survives data server restarts.

Blocks that have been finalized but not yet committed MAY be rolled back via CHUNK_ROLLBACK (Section 21.8).

21.4. Operation 80: CHUNK_HEADER_READ - Read Chunk Header from File

21.4.1. ARGUMENTS

```

/// struct CHUNK_HEADER_READ4args {
///     /* CURRENT_FH: file */
///     stateid4    chra_stateid;
///     offset4     chra_offset;
///     count4      chra_count;
/// };

```

Figure 59: XDR for CHUNK_HEADER_READ4args

21.4.2. RESULTS

```

/// struct CHUNK_HEADER_READ4resok {
///     bool          chrr_eof;
///     nfsstat4       chrr_status<>;
///     bool          chrr_locked<>;
///     chunk_owner4   chrr_chunks<>;
/// };

```

Figure 60: XDR for CHUNK_HEADER_READ4resok

```

// Do we want to have a chunk_owner for reads versus writes?  Instead
// of co-arrays, have one single in the responses?
//
// -- Tom

```

```

/// union CHUNK_HEADER_READ4res switch (nfsstat4 chrr_status) {
///     case NFS4_OK:
///         CHUNK_HEADER_READ4resok      chrr_resok4;
///     default:
///         void;
/// };

```

Figure 61: XDR for CHUNK_HEADER_READ4resok

21.4.3. DESCRIPTION

CHUNK_HEADER_READ differs from CHUNK_READ in that it only reads chunk headers in the desired data range.

21.5. Operation 81: CHUNK_LOCK - Lock Cached Chunk Data

21.5.1. ARGUMENTS

```

/// struct CHUNK_LOCK4args {
///     /* CURRENT_FH: file */
///     stateid4      cla_stateid;
///     offset4       cla_offset;
///     count4        cla_count;
///     chunk_owner4   cla_owner;
/// };

```

Figure 62: XDR for CHUNK_LOCK4args

21.5.2. RESULTS

```

    /// union CHUNK_LOCK4res switch (nfsstat4 clr_status) {
    ///     case NFS4_OK:
    ///         void;
    ///     case NFS4ERR_CHUNK_LOCKED:
    ///         chunk_owner4    clr_owner;
    ///     default:
    ///         void;
    /// };

```

Figure 63: XDR for CHUNK_LOCK4res

21.5.3. DESCRIPTION

CHUNK_LOCK acquires an exclusive lock on the block range specified by `cla_offset` and `cla_count`. While locked, other clients' `CHUNK_WRITE` operations to the same block range will fail with `NFS4ERR_CHUNK_LOCKED`. The lock is associated with the `chunk_owner4` in `cla_owner`.

If the blocks are already locked by a different owner, the operation returns `NFS4ERR_CHUNK_LOCKED` with the `clr_owner` field identifying the current lock holder.

CHUNK_LOCK is used in the multiple writer mode (Section 8.2.3.3) to coordinate concurrent access to the same block range. A client that needs to repair chunks SHOULD acquire the lock before writing replacement data.

The lock is released by `CHUNK_UNLOCK` (Section 21.9) or implicitly when the client's lease expires.

21.6. Operation 82: CHUNK_READ - Read Chunks from File

21.6.1. ARGUMENTS

```

    /// struct CHUNK_READ4args {
    ///     /* CURRENT_FH: file */
    ///     stateid4    cra_stateid;
    ///     offset4     cra_offset;
    ///     count4      cra_count;
    /// };

```

Figure 64: XDR for CHUNK_READ4args

21.6.2. RESULTS


```

/// struct read_chunk4 {
///     uint32_t      cr_crc;
///     uint32_t      cr_effective_len;
///     chunk_owner4   cr_owner;
///     uint32_t      cr_payload_id;
///     bool           cr_locked<>; // TDH - make a flag
///     nfsstat4      cr_status<>;
///     opaque         cr_chunk<>;
/// };

```

Figure 65: XDR for read_chunk4

```

/// struct CHUNK_READ4resok {
///     bool          crr_eof;
///     read_chunk4   crr_chunks<>;
/// };

```

Figure 66: XDR for CHUNK_READ4resok

```

/// union CHUNK_READ4res switch (nfsstat4 crr_status) {
///     case NFS4_OK:
///         CHUNK_READ4resok      crr_resok4;
///     default:
///         void;
/// };

```

Figure 67: XDR for CHUNK_READ4res

21.6.3. DESCRIPTION

CHUNK_READ is READ (see Section 18.22 of [RFC8881]) with additional semantics over the chunk_owner. As such, all of the normal semantics of READ directly apply.

The main difference between the two operations is that CHUNK_READ works on blocks and not a raw data stream. As such cra_offset is the starting block offset in the file and not the byte offset in the file. Some erasure coding types can have different block sizes depending on the block type. Further, cra_count is a count of blocks to read and not bytes to read.

When reading a set of blocks across the data servers, it can be the case that some data servers do not have any data at that location. In that case, the server either returns crr_eof if the cra_offset exceeds the number of blocks that the data server is aware of or it returns an empty block for that block.

For example, in Figure 68, the client asks for 4 blocks starting with the 3rd block in the file. The second data server responds as in Figure 69. The client would read this as there is valid data for blocks 2 and 4, there is a hole at block 3, and there is no data for block 5. The data server MUST calculate a valid `cr_crc` for block 3 based on the generated fields.

```
      Data Server 2
+-----+
| CHUNK_READ4args |
+-----+
| cra_stateid: 0   |
| cra_offset: 2    |
| cra_count: 4     |
+-----+
```

Figure 68: Example: `CHUNK_READ4args` parameters

```

Data Server 2
+-----+
| CHUNK_READ4resok |
+-----+
|
| crr_eof: true
| crr_chunks[0]:
|   cr_crc: 0x3faddace
|   cr_owner:
|     co_chunk_id: 2
|     co_guard:
|       cg_gen_id   : 3
|       cg_client_id: 6
|   cr_payload_id: 1
|   cr_chunk: ....
| crr_chunks[0]:
|   cr_crc: 0xdeade4e5
|   cr_owner:
|     co_chunk_id: 3
|     co_guard:
|       cg_gen_id   : 0
|       cg_client_id: 0
|   cr_payload_id: 1
|   cr_chunk: 0000...00000
| crr_chunks[0]:
|   cr_crc: 0x7778abcd
|   cr_owner:
|     co_chunk_id: 4
|     co_guard:
|       cg_gen_id   : 3
|       cg_client_id: 6
|   cr_payload_id: 1
|   cr_chunk: ....
|
+-----+

```

Figure 69: Example: Resulting CHUNK_READ4resok reply

21.7. Operation 83: CHUNK_REPAIRED - Confirm Repair of Errored Chunk Data

21.7.1. ARGUMENTS

```

/// struct CHUNK_REPAIRED4args {
///   /* CURRENT_FH: file */
///   stateid4      cpa_stateid;
///   offset4       cpa_offset;
///   count4        cpa_count;
///   chunk_owner4  cpa_owner;
/// };

```

Figure 70: XDR for CHUNK_REPAIRED4args

21.7.2. RESULTS

```

/// union CHUNK_REPAIRED4res switch (nfsstat4 cpr_status) {
///     case NFS4_OK:
///         void;
///     default:
///         void;
/// };

```

Figure 71: XDR for CHUNK_REPAIRED4res

21.7.3. DESCRIPTION

CHUNK_REPAIRED signals that blocks previously marked as errored (via CHUNK_ERROR, Section 21.2) have been repaired. The repair client writes replacement data via CHUNK_WRITE_REPAIR (Section 21.11), then calls CHUNK_REPAIRED to clear the error state and make the blocks available for normal reads.

The cpa_offset and cpa_count identify the repaired block range. The cpa_owner identifies the repair client that performed the repair. The data server verifies that the blocks were previously in error and that the repair data has been written and finalized.

If the blocks are not in the errored state, the operation returns NFS4ERR_INVALID.

21.8. Operation 84: CHUNK_ROLLBACK - Rollback Changes on Cached Chunk Data

21.8.1. ARGUMENTS

```

/// struct CHUNK_ROLLBACK4args {
///     /* CURRENT_FH: file */
///     offset4          crb_offset;
///     count4           crb_count;
///     chunk_owner4     crb_chunks<>;
/// };

```

Figure 72: XDR for CHUNK_ROLLBACK4args

21.8.2. RESULTS

```

/// struct CHUNK_ROLLBACK4resok {
///     verifier4        crr_writeverf;
/// };

```

Figure 73: XDR for CHUNK_ROLLBACK4resok

```

/// union CHUNK_ROLLBACK4res switch (nfsstat4 crr_status) {
///     case NFS4_OK:
///         CHUNK_ROLLBACK4resok    crr_resok4;
///     default:
///         void;
/// };

```

Figure 74: XDR for CHUNK_ROLLBACK4res

21.8.3. DESCRIPTION

CHUNK_ROLLBACK reverts blocks from the PENDING or FINALIZED state back to their previous state, effectively undoing a CHUNK_WRITE that has not yet been committed via CHUNK_COMMIT.

The crb_offset is the starting block offset and crb_count is the number of blocks to roll back. The crb_chunks array lists the chunk_owner4 entries whose blocks are to be rolled back. Each owner's blocks at the specified offsets MUST be in the PENDING or FINALIZED state; blocks that have already been committed via CHUNK_COMMIT cannot be rolled back.

CHUNK_ROLLBACK is used in two scenarios:

1. A client discovers an encoding error after CHUNK_WRITE and before CHUNK_COMMIT, and needs to undo the write to try again.
2. A repair client needs to undo a repair attempt that was found to be incorrect before committing it.

The data server deletes the pending chunk data and restores the block metadata to EMPTY. If the block was in the FINALIZED state, the persisted metadata is also removed.

21.9. Operation 85: CHUNK_UNLOCK - Unlock Cached Chunk Data

21.9.1. ARGUMENTS

```

/// struct CHUNK_UNLOCK4args {
///     /* CURRENT_FH: file */
///     stateid4      cua_stateid;
///     offset4       cua_offset;
///     count4        cua_count;
///     chunk_owner4  cua_owner;
/// };

```

Figure 75: XDR for CHUNK_UNLOCK4args

21.9.2. RESULTS

```
/// union CHUNK_UNLOCK4res switch (nfsstat4 cur_status) {  
///     case NFS4_OK:  
///         void;  
///     default:  
///         void;  
/// };
```

Figure 76: XDR for CHUNK_UNLOCK4res

21.9.3. DESCRIPTION

CHUNK_UNLOCK releases the exclusive lock on the block range previously acquired by CHUNK_LOCK (Section 21.5). The cua_owner MUST match the owner that acquired the lock; otherwise the operation returns NFS4ERR_INVALID.

If the blocks are not locked, the operation returns NFS4_OK (idempotent).

A client SHOULD release chunk locks promptly after completing its write or repair operation. Chunk locks are also released implicitly when the client's lease expires.

21.10. Operation 86: CHUNK_WRITE - Write Chunks to File

21.10.1. ARGUMENTS

```
/// union write_chunk_guard4 switch (bool cwg_check) {  
///     case TRUE:  
///         chunk_guard4    cwg_guard;  
///     case FALSE:  
///         void;  
/// };
```

Figure 77: XDR for write_chunk_guard4

```

/// const CHUNK_WRITE_FLAGS_ACTIVATE_IF_EMPTY = 0x00000001;
///
/// struct CHUNK_WRITE4args {
///     /* CURRENT_FH: file */
///     stateid4          cwa_stateid;
///     offset4           cwa_offset;
///     stable_how4        cwa_stable;
///     chunk_owner4       cwa_owner;
///     uint32_t           cwa_payload_id;
///     uint32_t           cwa_flags;
///     write_chunk_guard4 cwa_guard;
///     uint32_t           cwa_chunk_size;
///     uint32_t           cwa_crc32s<>;
///     opaque             cwa_chunks<>;
/// };

```

Figure 78: XDR for CHUNK_WRITE4args

21.10.2. RESULTS

```

/// struct CHUNK_WRITE4resok {
///     count4          cwr_count;
///     stable_how4      cwr_committed;
///     verifier4        cwr_writeverf;
///     nfsstat4         cwr_block_status<>;
///     bool             cwr_block_activated<>;
///     chunk_owner4     cwr_owners<>;
/// };

```

Figure 79: XDR for CHUNK_WRITE4resok

```

/// union CHUNK_WRITE4res switch (nfsstat4 cwr_status) {
///     case NFS4_OK:
///         CHUNK_WRITE4resok    cwr_resok4;
///     default:
///         void;
/// };

```

Figure 80: XDR for CHUNK_WRITE4res

21.10.3. DESCRIPTION

CHUNK_WRITE is WRITE (see Section 18.32 of [RFC8881]) with additional semantics over the chunk_owner and the activation of blocks. As such, all of the normal semantics of WRITE directly apply.

The main difference between the two operations is that `CHUNK_WRITE` works on blocks and not a raw data stream. As such `cwa_offset` is the starting block offset in the file and not the byte offset in the file. Some erasure coding types can have different block sizes depending on the block type. Further, `cwr_count` is a count of written blocks and not written bytes.

If `cwa_stable` is `FILE_SYNC4`, the data server MUST commit the written header and block data plus all file system metadata to stable storage before returning results. This corresponds to the NFSv2 protocol semantics. Any other behavior constitutes a protocol violation. If `cwa_stable` is `DATA_SYNC4`, then the data server MUST commit all of the header and block data to stable storage and enough of the metadata to retrieve the data before returning. The data server implementer is free to implement `DATA_SYNC4` in the same fashion as `FILE_SYNC4`, but with a possible performance drop. If `cwa_stable` is `UNSTABLE4`, the data server is free to commit any part of the header and block data and the metadata to stable storage, including all or none, before returning a reply to the client. There is no guarantee whether or when any uncommitted data will subsequently be committed to stable storage. The only guarantees made by the data server are that it will not destroy any data without changing the value of `writeverf` and that it will not commit the data and metadata at a level less than that requested by the client.

The activation of header and block data interacts with the `co_activated` for each of the written blocks. If the data is not committed to stable storage then the `co_activated` field MUST NOT be set to true. Once the data is committed to stable storage, then the data server can set the block's `co_activated` if one of these conditions apply:

- * it is the first write to that block and the `CHUNK_WRITE_FLAGS_ACTIVATE_IF_EMPTY` flag is set
- * the `CHUNK_COMMIT` is issued later for that block.

There are subtle interactions with write holes caused by racing clients. One client could win the race in each case, but because it used a `cwa_stable` of `UNSTABLE4`, the subsequent writes from the second client with a `cwa_stable` of `FILE_SYNC4` can be awarded the `co_activated` being set to true for each of the blocks in the payload.

Finally, the interaction of `cwa_stable` can cause a client to mistakenly believe that by the time it gets the response of `co_activated` of false, that the blocks are not activated. A subsequent `CHUNK_READ` or `HEADER_READ` might show that the `co_activated` is true without any interaction by the client via `CHUNK_COMMIT`.

21.10.3.1. Guarding the Write

A guarded `CHUNK_WRITE` is when the writing of a block **MUST** fail if `cwa_guard.cwg_check` is `TRUE` and the target chunk does not have the same `cg_gen_id` as `cwa_guard.cwg_guard.cg_gen_id`. This is useful in read-update-write scenarios. The client reads a block, updates it, and is prepared to write it back. It guards the write such that if another writer has modified the block, the data server will reject the modification.

As the `chunk_guard4` (see Figure 48) does not have a `chunk_id` and the `CHUNK_WRITE` applies to all blocks in the range of `cwa_offset` to the length of `cwa_data`, then each of the target blocks **MUST** have the same `cg_gen_id` and `cg_client_id`. The client **SHOULD** present the smallest set of blocks as possible to meet this requirement.

```
// Is the DS supposed to vet all blocks first or proceed to the first
// error? Or do all blocks and return an array of errors? (This
// last one is a no-go.) Also, if we do the vet first, what happens
// if a CHUNK_WRITE comes in after the vetting? Are we to lock the
// file during this process. Even if we do that, we still have the
// issue of multiple DSes.
//
// -- Tom
```

21.11. Operation 87: `CHUNK_WRITE_REPAIR` - Write Repaired Cached Chunk Data

21.11.1. ARGUMENTS

```
/// struct CHUNK_WRITE_REPAIR4args {
///     /* CURRENT_FH: file */
///     stateid4          cwra_stateid;
///     offset4           cwra_offset;
///     stable_how4       cwra_stable;
///     chunk_owner4      cwra_owner;
///     uint32_t          cwra_payload_id;
///     uint32_t          cwra_chunk_size;
///     uint32_t          cwra_crc32s<>;
///     opaque            cwra_chunks<>;
/// };
```

Figure 81: XDR for `CHUNK_WRITE_REPAIR4args`

21.11.2. RESULTS

```

/// struct CHUNK_WRITE_REPAIR4resok {
///     count4          cwrr_count;
///     stable_how4      cwrr_committed;
///     verifier4        cwrr_writeverf;
///     nfsstat4         cwrr_status<>;
/// };

```

Figure 82: XDR for CHUNK_WRITE_REPAIR4resok

```

/// union CHUNK_WRITE_REPAIR4res switch (nfsstat4 cwrr_status) {
///     case NFS4_OK:
///         CHUNK_WRITE_REPAIR4resok    cwrr_resok4;
///     default:
///         void;
/// };

```

Figure 83: XDR for CHUNK_WRITE_REPAIR4res

21.11.3. DESCRIPTION

CHUNK_WRITE_REPAIR has the same semantics as CHUNK_WRITE (Section 21.10) but is used specifically for writing reconstructed chunk data to a replacement data server during repair operations.

The repair workflow is:

1. The repair client reads surviving chunks from the remaining data servers via CHUNK_READ.
2. The client reconstructs the missing chunks using the erasure coding algorithm (RS matrix inversion or Mojette corner-peeling).
3. The client acquires a CHUNK_LOCK (Section 21.5) on the target data server to prevent concurrent writes during repair.
4. The client writes the reconstructed data via CHUNK_WRITE_REPAIR.
5. The client calls CHUNK_FINALIZE and CHUNK_COMMIT to persist the repair.
6. The client calls CHUNK_REPAIRED (Section 21.7) to clear the error state.
7. The client releases the lock via CHUNK_UNLOCK (Section 21.9).

CHUNK_WRITE_REPAIR is distinguished from CHUNK_WRITE to allow the data server to apply different policies to repair writes (e.g., bypassing guard checks, logging repair activity, or prioritizing repair I/O). The CRC32 validation on the repair data follows the same rules as CHUNK_WRITE.

The target blocks SHOULD be in the errored state (set by CHUNK_ERROR) or EMPTY. If the blocks are in the COMMITTED state with valid data, the data server MAY reject the repair to prevent overwriting good data.

22. 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 MUST 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.

22.1. CRC32 Integrity Scope

The CRC32 values carried in `CHUNK_WRITE` and returned from `CHUNK_READ` are intended to detect accidental data corruption during storage or transmission — for example, bit flips in storage media or network errors. CRC32 is not a cryptographic hash and does not protect against intentional modification: an adversary with access to the network path could replace a chunk and recompute a valid CRC32 to match. The "data integrity" provided by the CRC32 mechanism in this document refers to error detection, not protection against an active attacker. Deployments requiring protection against active attackers SHOULD use RPC-over-TLS (see Section 22.4) or `RPCSEC_GSS`.

22.2. Chunk Lock and Lease Expiry

When a client holds a chunk lock (acquired via `CHUNK_LOCK`) and its lease expires or the client crashes, the lock is released implicitly by the data server. This opens a window in which another client may write to the previously locked range before the original client's repair is complete. Implementations SHOULD ensure that the lease period for chunk locks is sufficient to complete repair operations, and SHOULD implement `CHUNK_UNLOCK` explicitly on abort paths. The metadata server's `LAYOUTERROR` and `LAYOUTRETURN` mechanisms provide the

coordination point for detecting and resolving such races.

22.3. Error Code Information Disclosure

The new error codes `NFS4ERR_CHUNK_LOCKED` (10099) and `NFS4ERR_PAYLOAD_NOT_CONSISTENT` (10098) convey information about chunk state to the caller. Both of these errors MAY be returned to callers whose credentials have not been verified by the data server (e.g., when the `AUTH_SYS` uid presented does not match the synthetic uid on the data file). The information they reveal — that a chunk is locked, or that a CRC mismatch occurred — does not directly disclose file contents but may indicate concurrent write activity. Implementations that are concerned about this level of disclosure SHOULD require that operations on `CHUNK` ops only succeed after credential verification and return `NFS4ERR_ACCESS` for unverified callers rather than the more specific error codes.

22.4. Transport Layer Security

RPC-over-TLS [RFC9289] MAY be used to protect traffic between the client and the metadata server and between the client and data servers. When RPC-over-TLS is in use on the data server path, the synthetic uid/gid credentials carried in `AUTH_SYS` remain the access control mechanism; TLS provides confidentiality and integrity for the transport but does not replace the fencing model described in Section 2.2. Servers that require transport security SHOULD advertise this via the `SECINFO` mechanism rather than silently dropping connections.

22.5. 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.

22.5.1. 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.

22.5.2. 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.

23. 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 a new layout type number: LAYOUT4_FLEX_FILES_V2 (see Table 10).

Layout Type Name	Value	RFC	How	Minor Versions
LAYOUT4_FLEX_FILES_V2	0x5	RFCTBD10	L	1

Table 10: 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 11).

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 11: Recallable Object Type Assignments

This document introduces the 'Flexible File Version 2 Layout Type Erasure Coding Type Registry'. The registry uses a 32-bit value space partitioned into ranges based on the intended scope of the encoding type (see Table 12).

Range	Purpose	Allocation Policy
0x00000x00FF	Standards Track	IETF Review (RFC required)
0x01000x0FFF	Experimental	Expert Review
0x10000x7FFF	Vendor (open)	First Come First Served
0x80000xFFFE	Private/proprietary	No registration required
0xFFFF	Reserved	—

Table 12: Erasure Coding Type Value Ranges

Standards Track (0x00000x00FF) Encoding types intended for broad interoperability. The specification MUST include a complete mathematical description sufficient for independent interoperable implementations (see Section 5.1.1). Allocated by IETF Review.

Experimental (0x01000x0FFF) Encoding types under development or evaluation. An Internet-Draft is sufficient for allocation. The specification SHOULD include enough detail for interoperability testing. Allocated by Expert Review.

Vendor (open) (0x10000x7FFF) Encoding types with a published specification or patent reference. Interoperability is expected among implementations that license or implement the specification. The registration MUST include either a math specification or a patent reference. Allocated First Come First Served.

Private/proprietary (0x80000xFFFE) Encoding types for use within a single vendor's ecosystem. No registration is required. Interoperability with other implementations is not expected. The encoding type name SHOULD include an organizational identifier (e.g., FFFV2_ENCODING_ACME_FOOBAR). A client that encounters a value in this range from an unrecognized server SHOULD treat it as an unsupported encoding type.

This partitioning prevents contention for small numbers in the Standards Track range and provides a clear signal to clients about what level of interoperability to expect.

This document defines the FFFV2_CODING_MIRRORED type for Client-Side Mirroring (see Table 13).

Erasure Coding Type Name	Value	RFC	How	Minor Versions
FFV2_CODING_MIRRORED	1	RFCTBD10	L	2
FFV2_ENCODING_MOJETTE_SYSTEMATIC	2	RFCTBD10	L	2
FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC	3	RFCTBD10	L	2
FFV2_ENCODING_RS_VANDERMONDE	4	RFCTBD10	L	2

Table 13: Flexible File Version 2 Layout Type Erasure Coding Type Assignments

Acknowledgments

The following from Hammerspace were instrumental in driving Flexible File Version 2 Layout Type: David Flynn, Trond Myklebust, Didier Feron, Jean-Pierre Monchanin, Pierre Evenou, and Brian Pawlowski.

Pierre Evenou contributed the Mojette Transform encoding type specification, drawing on the work of Nicolas Normand, Benoit Parrein, and the discrete geometry research group at the University of Nantes.

Christoph Helwig was instrumental in making sure the Flexible File Version 2 Layout Type was applicable to more than the Mojette Transformation.

Chris Inacio, Brian Pawlowski, and Gorrry Fairhurst guided this process.

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>>.
- [RFC7863] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 External Data Representation Standard (XDR) Description", RFC 7863, DOI 10.17487/RFC7863, November 2016, <<https://www.rfc-editor.org/rfc/rfc7863>>.

- [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>>.
- [RFC8178] Noveck, D., "Rules for NFSv4 Extensions and Minor Versions", RFC 8178, DOI 10.17487/RFC8178, July 2017, <<https://www.rfc-editor.org/rfc/rfc8178>>.
- [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>>.
- [RFC9289] Myklebust, T. and C. Lever, Ed., "Towards Remote Procedure Call Encryption by Default", RFC 9289, DOI 10.17487/RFC9289, September 2022, <<https://www.rfc-editor.org/rfc/rfc9289>>.

Informative References

- [KATZ] Katz, M., "Questions of Uniqueness and Resolution in Reconstruction from Projections", Springer , 1978.
- [NORMAND] Normand, N., Kingston, A., and P. Evenou, "A Geometry Driven Reconstruction Algorithm for the Mojette Transform", LNCS 4245, pp. 122-133, DGC I 2006, 2006.
- [PARREIN] Parrein, B., Normand, N., and J.-P. Guedon, "Multiple Description Coding Using Exact Discrete Radon Transform", IEEE Data Compression Conference (DCC), 2001.
- [Plank97] Plank, J., "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like System", September 1997.
- [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