Network File System Version 4
Internet-Draft
Intended status: Standards Track
Expires: 30 October 2026

T. Haynes
Hammerspace

28 April 2026

Parallel NFS (pNFS) Flexible File Layout Version 2
draft-haynes-nfsv4-flexfiles-v2-05

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.

Status of This Memo

Table of Contents

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. This document also introduces new operations that allow the client to roll back 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.

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.

3. Motivation

Server-sided erasure coding places the erasure-coding compute at the server, which becomes a bottleneck as the number of concurrent clients grows. Moving the erasure transform to the client parallelizes the compute across all writers: each client encodes locally and fans out the resulting chunks to the data servers directly, keeping the metadata server in its coordinator role for metadata rather than making it a data-path funnel.

Flex Files v1 ([RFC8435]) already places the replication transform at the client via client-side mirroring, but mirroring provides no integrity check: silent byte corruption is undetectable, and repairing a damaged mirror requires choosing a trusted copy essentially blind. Flex Files v2 adds two integrity mechanisms -- a per-chunk CRC32 for on-wire and at-rest bit-flip detection, and the chunk_guard4 compare-and-swap primitive (see Section 24.1) for detecting concurrent-writer inconsistency -- while preserving the client-side compute model. The chunk_guard4 per-chunk header is 8 bytes total (a 32-bit generation id and a 32-bit owning-client short-id); this keeps the metadata-server overhead for maintaining erasure-coding consistency to the smallest value that still admits a CAS tiebreaker.

An alternative to client-side erasure coding is to keep the erasure-coding transform inside the storage system -- that is, on the data servers themselves, or on a server-side pre-ingest stage between the client and the data servers. This approach has real advantages: a single codec is fixed at the storage system, so clients do not have to negotiate codec support; repair never traverses the client; and the wire protocol stays minimal because no on-wire consistency primitives are needed.

Flex Files v2 does not choose that path, for three reasons:

* *Scale bottleneck.* The storage system becomes a scale bottleneck in exactly the way this section opens with: large-scale parallel workloads drive the aggregate erasure-coding compute beyond what a bounded storage tier can supply, while clients are the naturally horizontally scaling resource.

* *Loss of per-file codec flexibility.* A single server-fixed codec forecloses the option of picking different codecs for different files in the same namespace, which matters when files have different durability and performance requirements.

* *Benchmark evidence.* Measurements summarised in Section "Implementation Status" show that client-side encoding with the overhead introduced here is competitive with server-side encoding on realistic workloads, and scales the encoding compute with the writer population rather than with the data-server count.

The right answer for a given deployment is not universal; Section "Design Rationale: Rejected Alternatives" records the alternatives considered and why each was not chosen for Flex Files v2's target workload classes.

Client-side erasure coding turns write-hole recovery into a protocol-level concern rather than an implementer-internal one. In Flex Files v1, the replication transform produces independent full-copy mirrors, so a partial write is detected and repaired by resilvering from a surviving copy. A single server-side coordinator has enough visibility to drive that repair without help from the client. Under a (k, m) erasure code, in contrast, a write transaction fans out across multiple data servers with no single server-side actor holding whole-transaction visibility: when the client fails mid-fan-out, the partial state across data servers must be reconciled by the metadata server, and the reconciliation protocol must be specified on the wire so that any compliant client, data server, or repair agent can participate. The chunk_guard4 CAS primitive, the PENDING / FINALIZED / COMMITTED state machine, the CHUNK_LOCK escrow mechanism, and CB_CHUNK_REPAIR together form that on-wire reconciliation protocol.

Scope note: the consistency goal of Flex Files v2 is RAID consistency across the chunks that make up an encoded stripe, not POSIX write ordering across arbitrary application writes. The protocol does not attempt to make overlapping application writes from different clients atomic: that is the province of file locking ([RFC8881], Section 12) and of application-level coordination. What the protocol does guarantee is that the chunks comprising a given stripe agree on which write produced them, so that readers and repair clients never observe a half-applied stripe. Readers who need cross-write ordering beyond a single stripe MUST use the existing NFSv4 locking primitives.

4. Use Cases

The protocol is designed around three workload classes. The percentages below reflect the expected deployment mix in installations that choose Flex Files v2 for its combination of integrity and performance; individual deployments may diverge.

Single writer, multiple readers (approximately 90% of expected

deployments): The common case is a file written by one client and subsequently read by many. Examples include artifacts deposited by batch jobs, container images, and media files. The protocol is optimized for this case; see Section 12.7.

Multiple writers without sustained contention (approximately 9% of

expected deployments): Files with multiple concurrent writers where races on the same chunk are rare. Examples include shared-directory append-only logs and distributed builds. The chunk_guard4 CAS primitive and per-chunk locking cover this case without penalizing the common single-writer path.

Multiple writers with high-frequency contention, no overwrite

(approximately 1% of expected deployments): High-performance computing (HPC) checkpoint workloads, in which many ranks write disjoint regions of the same file in lockstep. The protocol relies on block alignment to keep per-chunk contention rare despite overall high writer count. Contention that does occur is resolved via the deterministic tiebreaker rule defined in Section 24.1.

Scale targets include multi-thousand-client deployments (on the order of tens of thousands of concurrent clients for HPC checkpointing), parallel-filesystem replacements, and multi-rack shared-storage clusters. The repair protocol (see Section 11.2.4) is designed to let such deployments tolerate data-server failures and concurrent-writer races without blocking the critical path for the first two workload classes.

5. 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 Mojette 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.

6. 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.

6.1. 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.

6.2. 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 8.2), 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 6.3) 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.

6.2.1. 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.

6.2.2. Example of using Synthetic uids/gids

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

The metadata server entry may look like:

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

Figure 1: Metadata’s view of ompha.c

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

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

Figure 2: Data’s view of ompha.c

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

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

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

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

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

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

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

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

6.3. 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.

6.3.1. 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.

6.3.2. 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. For flex files v2 deployments in which the storage devices are NFSv4.2 servers, those facilities are provided by the TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID operations defined in Section 6.4.

The metadata server and a storage device establish that they can use TRUST_STATEID via a two-part handshake, both parts of which MUST succeed before the metadata server may issue TRUST_STATEID against that storage device for production traffic:

1. *Capability probe.* At control-session setup the metadata server sends a TRUST_STATEID against the anonymous stateid (see Section 6.4.1). A storage device that supports tight coupling MUST reject the probe with NFS4ERR_INVAL; a storage device that does not support tight coupling returns NFS4ERR_NOTSUPP and the metadata server falls back to loose coupling. The metadata server records the result per storage device in ffdv_tightly_coupled.

2. *Control-session gating.* The metadata server presents EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID when it opens the control session to the storage device (see Section 6.4.2). The storage device MUST reject any incoming TRUST_STATEID, REVOKE_STATEID, or BULK_REVOKE_STATEID that does not arrive on such a session with NFS4ERR_PERM. This is the authorization mechanism that distinguishes the metadata server from ordinary pNFS clients, which connect with EXCHGID4_FLAG_USE_PNFS_DS or EXCHGID4_FLAG_USE_NON_PNFS and are therefore structurally unable to invoke these operations.

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 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.

6.4. Tight Coupling Control Protocol

When an NFSv4.2 storage device participates in a tightly coupled deployment, the metadata server and the storage devices need a control protocol that:

1. registers the layout stateid with each storage device so the storage device can validate client I/O independently; and

2. revokes trust promptly when the metadata server withdraws the client’s authorization -- for example, on CB_LAYOUTRECALL timeout, lease expiry, or layout return after error.

This specification defines that control protocol as three new NFSv4.2 operations: TRUST_STATEID (Section 25.12), REVOKE_STATEID (Section 25.13), and BULK_REVOKE_STATEID (Section 25.14). These operations are sent by the metadata server to each storage device over a dedicated control session (see Section 6.4.2) and MUST NOT be sent by pNFS clients.

The receiver of these operations is any server the metadata server delegates client-I/O admission to. In this document that is the storage device (DS). The same mechanism applies to a proxy server (PS) as defined in the companion Data Mover draft -- a PS may or may not additionally act as a DS, but in either role it needs the metadata server to register a layout stateid before it can admit client I/O. Where this section says "storage device," read it as "storage device, or proxy server as defined in the companion Data Mover draft"; the flag check and the three operations are identical for both roles.

6.4.1. Capability Discovery

A storage device indicates support for tight coupling implicitly, by processing TRUST_STATEID rather than returning NFS4ERR_NOTSUPP. The metadata server probes each storage device during control-session setup:

```
SEQUENCE + PUTROOTFH + TRUST_STATEID(
    tsa_layout_stateid = ANONYMOUS_STATEID,
    tsa_iomode         = LAYOUTIOMODE4_READ,
    tsa_expire         = 0,
    tsa_principal      = "")
```

Figure 4: TRUST_STATEID capability probe

The anonymous stateid is used deliberately: a correctly implemented storage device MUST reject it (see Section 25.12), so the probe cannot accidentally register garbage in the trust table. The metadata server interprets the probe response as follows:

* NFS4ERR_NOTSUPP: tight coupling is not supported on this storage device. The metadata server falls back to loose coupling (anonymous stateid plus fencing) and sets ffdv_tightly_coupled to false for this storage device.

* NFS4ERR_INVAL: tight coupling is supported. The anonymous stateid was correctly rejected. The metadata server records the capability and sets ffdv_tightly_coupled to true for this storage device.

* NFS4_OK: the storage device accepted an anonymous stateid into its trust table. This is a storage device bug. The metadata server SHOULD log the anomaly. It MAY treat the capability as confirmed to avoid downgrading to loose coupling, but it MUST immediately issue REVOKE_STATEID to remove the bogus entry.

The capability is recorded per storage device, not per file. Partial support across a mirror set is permitted: each ff_device_versions4 entry returned by GETDEVICEINFO carries its own ffdv_tightly_coupled flag, set independently.

6.4.2. Control Session

The metadata server establishes an NFSv4.2 session to each tight-coupling-capable storage device at startup. On this session the metadata server acts as the storage device’s client and presents EXCHGID4_FLAG_USE_PNFS_MDS in its EXCHANGE_ID args.

The storage device MUST verify that any incoming TRUST_STATEID, REVOKE_STATEID, or BULK_REVOKE_STATEID compound arrives on a session whose owning client presented EXCHGID4_FLAG_USE_PNFS_MDS in its EXCHANGE_ID args. Requests that arrive on any other session MUST be rejected with NFS4ERR_PERM. This is the sole access control on these operations; a pNFS client connecting to the storage device does not present EXCHGID4_FLAG_USE_PNFS_MDS and therefore cannot invoke them.

The EXCHGID4_FLAG_USE_PNFS_MDS check replaces any path- or filehandle-level gating. TRUST_STATEID operates on a filehandle that may be any file on the storage device, and the metadata server is the sole authority that can legitimately speak this protocol.

Because the EXCHGID4_FLAG_USE_PNFS_MDS check relies on the owning client's self-declaration at EXCHANGE_ID time, the storage device cannot by itself distinguish a legitimate metadata server from any other host that sets the flag. Deployments are therefore responsible for constraining who can establish a control session in the first place. Two mechanisms are RECOMMENDED:

1. The control session SHOULD use RPCSEC_GSS with a machine principal that the storage device has been configured to accept as a metadata server. The storage device validates the principal before accepting EXCHANGE_ID with EXCHGID4_FLAG_USE_PNFS_MDS.

2. Alternatively, the control session SHOULD run over a network path isolated from pNFS clients (for example, a dedicated management VLAN or mutual TLS ([RFC9289]) with an allowlisted client certificate), such that only configured metadata servers can reach the storage device on that path.

Deploying neither mechanism reduces the authorization strength of TRUST_STATEID and the revocation operations to "any host that can reach the storage device can invoke them"; a strict deployment MUST apply at least one of the above.

6.4.3. Flow at LAYOUTGET

For each new or refreshed layout segment, the metadata server:

1. chooses the layout stateid (as it would without tight coupling);

2. identifies the tight-coupling-capable storage devices in the mirror set (those for which ffdv_tightly_coupled is true);

3. fans out TRUST_STATEID to each such storage device, specifying the layout stateid, the layout iomode, a tsa_expire derived from the metadata server's lease (see Section 6.4.6), and the client's authenticated identity in tsa_principal;

4. waits for all fan-outs to complete (or reach their per-storage-device timeout) before returning the layout.

If every storage device in the mirror set rejects the TRUST_STATEID fan-out, the metadata server MUST NOT return the layout; instead it returns NFS4ERR_LAYOUTTRYLATER. If some storage devices accept and others reject, the metadata server MAY return a layout covering only the accepting storage devices, subject to the mirror-set rules for minimum acceptable coverage. A storage device that returns NFS4ERR_DELAY is retried until either success or the metadata server's LAYOUTGET-response budget is exhausted. If a storage device returns NFS4ERR_NOTSUPP at this time (having accepted the probe earlier), the metadata server MUST clear ffdv_tightly_coupled for this storage device, fall back to loose coupling, and re-issue the layout accordingly.

6.4.4. Principal Binding and the Kerberos Gap

Flex files v1 has a known gap: a client authenticated to the metadata server with Kerberos has no way to present the same authenticated identity to the storage device, because v1 layouts carry only ffds_user / ffds_group (POSIX uid/gid for AUTH_SYS). A strict Kerberos deployment on v1 must either allow AUTH_SYS from the metadata server's subnet or accept that the v1 data path is not Kerberos-protected.

The tsa_principal field in TRUST_STATEID closes that gap. When a client authenticates to the metadata server as a Kerberos principal (e.g., alice@REALM), the metadata server passes that principal name to each storage device in tsa_principal. The storage device then enforces a two-part check on each CHUNK operation that presents the layout stateid:

a. the stateid is in the trust table and has not expired; and

b. the caller's authenticated identity (the RPCSEC_GSS display name on the CHUNK compound) matches tsa_principal.

Both conditions MUST hold. On principal mismatch the storage device MUST return NFS4ERR_ACCESS -- the semantics are "you do not have an authorized layout for this file", which matches the existing fencing error and avoids the confusion of NFS4ERR_WRONGSEC (which directs the client to re-authenticate with a different flavor) or NFS4ERR_BAD_STATEID (which directs the client to return the layout).

The metadata server MUST populate tsa_principal with the RPCSEC_GSS display name of the authenticated client when the client authenticated to the metadata server via RPCSEC_GSS. The metadata server MUST set tsa_principal to the empty string only for AUTH_SYS and TLS clients (for which there is no server- verified per-user identity). Setting tsa_principal to the empty string for an RPCSEC_GSS client disables the principal check on the storage device and silently re-opens the flex files v1 Kerberos gap; it is a metadata server bug, not a protocol option.

If tsa_principal is the empty string, no principal check applies. This is the expected setting for AUTH_SYS and TLS clients:

* AUTH_SYS clients have no server-verified identity. The storage device’s stateid check and the AUTH_SYS uid/gid on the data file together constitute the authorization. In a tightly coupled deployment the data file’s owner/group need not match the metadata file’s, since ffv2ds_user and ffv2ds_group are ignored (see Section 8.8).

* TLS clients have transport-layer authentication via mutual TLS ([RFC9289]). The TLS layer authenticates the client machine; the stateid check confirms the metadata server authorized that machine to access this file. The machine-level authentication is handled beneath the RPC layer and is not reflected in tsa_principal. Opportunistic TLS (STARTTLS without certificate verification) provides encryption but not authentication, and therefore has the same authorization properties as plain AUTH_SYS.

When a client’s I/O is routed through a proxy server (PS) -- that is, the layout the metadata server returns to the client has FFV2_DS_FLAGS_PROXY set on the proxy’s ffv2_data_server4 entry, per the companion Data Mover draft -- the storage device observes CHUNK operations arriving from the PS’s address rather than from the client directly. The tsa_principal the metadata server populates in TRUST_STATEID is the principal the _storage device_ will observe on those CHUNK operations, and the Data Mover draft’s credential-forwarding rules (in particular rule 1, "Credential pass-through") require the PS to forward the client’s credentials verbatim on every CHUNK operation it issues on the client’s behalf. Therefore:

* For an RPCSEC_GSS client whose I/O is proxied through a PS, the metadata server MUST set tsa_principal to the client's RPCSEC_GSS display name (identical to the non-proxied case). The storage device's principal check on CHUNK operations will match against the client's principal on the forwarded compound, not the PS's service identity.

* For an AUTH_SYS client whose I/O is proxied through a PS, the metadata server MUST set tsa_principal to the empty string (identical to the non-proxied case). The PS forwards the client's AUTH_SYS uid/gid; the storage device's stateid check plus the forwarded AUTH_SYS uid/gid constitute the authorization.

The metadata server MUST NOT set tsa_principal to the PS's own service principal. Doing so would require the PS to authenticate to the storage device as itself (bypassing credential forwarding) which is explicitly prohibited by the Data Mover draft's rule 4 ("PS service identity is for the control plane only").

6.4.5. Client-Detected Trust Gap

A window exists between a successful TRUST_STATEID fan-out and the client's first I/O to the storage device. A transient failure may cause the storage device to forget or reject the entry before the client's first CHUNK_WRITE arrives. The client cannot distinguish this case from legitimate revocation; both surface as NFS4ERR_BAD_STATEID on the storage device.

The recovery path:

1. The client sends LAYOUTERROR(layout_stateid, device_id, NFS4ERR_BAD_STATEID) to the metadata server.

2. The metadata server retries TRUST_STATEID against the reporting storage device. If the retry succeeds, the metadata server returns NFS4_OK for LAYOUTERROR. The client retries the original I/O.

3. If the retry fails -- the storage device is unreachable or returns a hard error -- the metadata server issues CB_LAYOUTRECALL for that device and the client returns the layout segment covering that storage device. The client is expected to re-request via LAYOUTGET.

This is the same LAYOUTERROR path used for NFS4ERR_ACCESS or NFS4ERR_PERM in the fencing model (see Section 6.2), with the metadata server's action being "retry TRUST_STATEID" instead of "rotate uid/gid".

6.4.6. Lease and Renewal

tsa_expire in a TRUST_STATEID request is a wall-clock expiry instant expressed as an nfstime4. The metadata server MUST set tsa_expire to the current wall-clock time plus the metadata server's client lease period.

The metadata server MUST re-issue TRUST_STATEID for an entry before tsa_expire while the corresponding layout is outstanding. The RECOMMENDED trigger is: when an entry is within half the lease period of its tsa_expire, re-issue TRUST_STATEID with a refreshed tsa_expire. Renewing on every SEQUENCE that keeps the layout stateid alive is correct but produces metadata-server-to-storage-device traffic proportional to the client's SEQUENCE rate, which is undesirable in steady state.

If an entry expires on the storage device before the metadata server renews it -- for example, because the metadata server is partitioned from the storage device for longer than the lease period -- the storage device MUST return NFS4ERR_BAD_STATEID to the client on the next CHUNK operation. The client returns the layout to the metadata server and re-requests. This is the same recovery path as the trust gap described above.

6.4.7. Storage Device Crash Recovery

The trust table is volatile. The storage device MUST NOT persist trust entries across restarts; a storage device restart therefore empties the trust table.

The client detects a storage device restart via NFS4ERR_BADSESSION or NFS4ERR_STALE_CLIENTID on its data server session. The client returns the affected layout segment to the metadata server via LAYOUTRETURN and re-requests via LAYOUTGET. The metadata server then fans out fresh TRUST_STATEID operations to the recovered storage device.

Planned storage device restarts (software upgrade, etc.) SHOULD drain in-flight CHUNK operations before shutting down.

6.4.8. Metadata Server Crash Recovery

When the metadata server restarts, its control sessions to the storage devices are lost. Trust entries remain on the storage devices until tsa_expire, but the metadata server is no longer renewing them; the entries are effectively orphaned until the metadata server completes grace.

When the metadata server reconnects to a storage device with a new boot epoch -- that is, the EXCHANGE_ID returns a new server owner on the storage device's view of the metadata server -- the storage device SHOULD mark all trust entries established under the prior metadata-server epoch as pending-revalidation. While an entry is pending-revalidation:

* I/O that presents the entry's stateid MUST receive NFS4ERR_DELAY, not NFS4ERR_BAD_STATEID. NFS4ERR_DELAY tells the client to retry with the same stateid -- the metadata server is recovering and may yet revalidate the entry. NFS4ERR_BAD_STATEID would instead cause the client to return the layout immediately, producing a thundering herd against the metadata server during grace.

* An entry remains pending-revalidation until the metadata server either re-issues TRUST_STATEID for it (which transitions it back to trusted) or until the entry's tsa_expire elapses (which removes it).

The metadata server's recovery sequence is:

1. Reconnect to each storage device and establish a fresh control session.

2. Optionally issue BULK_REVOKE_STATEID with an all-zeros clientid to each storage device. This clears the prior trust table eagerly; skipping this step is correct, because orphan entries expire via tsa_expire.

3. Enter grace and accept RECLAIM operations from clients. For each reclaimed layout, fan out TRUST_STATEID to the relevant storage devices.

4. Exit grace. Clients that did not reclaim in time have their state revoked; the metadata server issues REVOKE_STATEID or BULK_REVOKE_STATEID on their behalf.

Metadata servers SHOULD persist the set of outstanding TRUST_STATEID entries (clientid, layout stateid, storage device address, tsa_expire) to stable storage. With this persistence the metadata server can re-issue TRUST_STATEID for all known entries immediately upon reconnecting to each storage device, before clients begin reclaiming. This shrinks the window during which the storage device returns NFS4ERR_DELAY for client I/O. Persistence is a latency optimization, not a correctness requirement: the re-layout path handles recovery in all cases.

6.4.9. Backward Compatibility

* NFSv3 storage devices are unchanged. They are always treated as loosely coupled; TRUST_STATEID does not exist on NFSv3 servers.

* NFSv4.2 storage devices for which the TRUST_STATEID probe returns NFS4ERR_NOTSUPP are treated as loosely coupled; fencing is the only revocation mechanism, the same as for NFSv3.

* NFSv4.2 storage devices for which the probe returns NFS4ERR_INVAL support tight coupling; the metadata server uses TRUST_STATEID at LAYOUTGET and REVOKE_STATEID or BULK_REVOKE_STATEID for revocation instead of fencing.

A single deployment MAY contain a mix of tight-coupled and loose-coupled storage devices; each is negotiated independently via the probe.

7. 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.

7.1. ff_device_addr4

The ff_device_addr4 data structure (see Figure 6) 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.

The ff_device_versions4 and ff_device_addr4 structures are reused unchanged from [RFC8435]; they are reproduced here for reader convenience and are not part of the XDR extracted from this document.

```
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 5: ff_device_versions4 (reused from RFC 8435)

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

Figure 6: ff_device_addr4 (reused from RFC 8435)

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 8.14).

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 the metadata server is tightly coupled with this storage device. 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. For an NFSv4.2 storage device, the metadata server sets ffdv_tightly_coupled to true only after confirming the storage device implements the TRUST_STATEID control protocol via the capability probe described in Section 6.4.1. An NFSv4.2 storage device that does not implement TRUST_STATEID (returning NFS4ERR_NOTSUPP to the probe) MUST be advertised with ffdv_tightly_coupled set to false.

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]).

7.2. 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.

8. Flexible File Version 2 Layout Type

The original layouttype4 introduced in [RFC5662] is extended as shown in Figure 7. The layout_content4 and layout4 structures are reused unchanged from [RFC5662]; the layouttype4 enum is extended with the new LAYOUT4_FLEX_FILES_V2 value. The full enum and surrounding structures below are reproduced for reader convenience; only the new constant LAYOUT4_FLEX_FILES_V2 is part of the XDR extracted from this document (see Figure 8).

```
enum layouttype4 {
    LAYOUT4_NFSV4_1_FILES   = 1,
    LAYOUT4_OSD2_OBJECTS    = 2,
    LAYOUT4_BLOCK_VOLUME    = 3,
    LAYOUT4_FLEX_FILES      = 4,
    LAYOUT4_SCSI            = 5,
    LAYOUT4_FLEX_FILES_V2   = 6
};

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 7: The original layout type (illustrative; reused from RFC 5662 with extension)

The extracted XDR contribution for this extension is the new layouttype4 constant alone:

```
/// const LAYOUT4_FLEX_FILES_V2 = 6;
```

Figure 8: New layouttype4 value (extracted)

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.

8.1. 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.

The 32-bit ffv2_coding_type4 value space is partitioned by intended scope -- Standards Track, Experimental, Vendor (open), and Private / proprietary -- with different allocation policies per range, so that vendors can assign codec values without consuming standards-track codepoints. See Table 13 and the accompanying prose in Section 28 for the range assignments and allocation policies.

FFV2_CODING_MIRRORED offers replication of data and not integrity of data. As such, it does not need operations like CHUNK_WRITE (see Section 25.10.

8.1.1. 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.

8.2. ffv2_layout4

8.2.1. 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.

Each flag below describes both the semantics when set and the normative requirement it places on the client. When a flag is not set, the client MUST follow the default behavior described for its unset state.

FFV2_FLAGS_NO_LAYOUTCOMMIT: When set, the client MAY omit the LAYOUTCOMMIT to the metadata server. When unset, the client MUST send LAYOUTCOMMIT per [RFC8881] Section 18.42.

FFV2_FLAGS_NO_IO_THRU_MDS: When set, the client MUST NOT proxy I/O operations through the metadata server, even after detecting a network disconnect to a storage device. When unset, the client MAY retry failed I/O via the metadata server.

FFV2_FLAGS_NO_READ_IO: When set, the client MUST NOT issue READ against layouts of iomode LAYOUTIOMODE4_RW, and MUST instead request a separate layout of iomode LAYOUTIOMODE4_READ for any read I/O. When unset, the client MAY issue READ against either iomode.

FFV2_FLAGS_WRITE_ONE_MIRROR: When set, the client MAY update only one mirror of each layout segment (see Section 11.1) and rely on the metadata server or a peer data server to propagate the update to the remaining mirrors. When unset, the client MUST update all mirrors.

FFV2_FLAGS_ONLY_ONE_WRITER: When set, the client is the exclusive writer for the layout and MAY issue CHUNK_WRITE without setting cwa_guard, retaining the ability to use CHUNK_ROLLBACK in the event of a write hole caused by overwriting. When unset, the client MUST set cwa_guard on every CHUNK_WRITE so that chunk_guard4 CAS can prevent collisions across concurrent writers.

8.3. 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.

8.4. 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 informs the client that the indicated data server is a replacement for a previously failed ACTIVE data server, whose content has been (or is being) reconstructed from the surviving shards of the mirror set. A REPAIR data server differs from a SPARE in two ways:

* A SPARE is standing by with no payload; the client MAY fail over to it at write time without metadata-server coordination.

* A REPAIR has been promoted by the metadata server to replace a failed ACTIVE, and its payload was placed there by a repair client executing the flow in Section 11.2.4 rather than directly by the original writer. The flag is the client's indication that reads from this data server return erasure-decoded content rather than content produced by the original write.

Clients that rely on write-provenance information (for example, deployments that track which client wrote which generation) SHOULD be aware of the REPAIR flag so they do not treat the reconstructed payload as if it had been written directly by the cg_client_id recorded in the chunk_guard4; the guard values still match across the mirror set by construction, but the physical write path differs.

Over the lifetime of a file, a single data server MAY transition ACTIVE -> REPAIR (on replacement) or REPAIR -> ACTIVE (once the metadata server has accepted the reconstructed content as authoritative and the fail-over is complete); the metadata server reflects the current flag set in the next layout it returns.

8.5. 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.

8.6. 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_striping {
///     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.

8.7. 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.

8.8. ffv2_mirror4

```
/// struct ffv2_mirror4 {
///         ffv2_coding_type_data4  ffm_coding_type_data;
///         ffv2_key4               ffm_key;
///         ffv2_striping           ffm_striping;
///         uint32_t                ffm_striping_unit_size;
///         uint32_t                ffm_client_id;
///         ffv2_stripes4           ffm_stripes<>;
/// };
```

Figure 17: The ffv2_mirror4

The ffv2_mirror4 (in Figure 17) describes the Flexible File Layout Version 2 specific fields.

The ffm_client_id is a 32-bit value, assigned by the metadata server at layout-grant time, that the client MUST use as the cg_client_id field of chunk_guard4 (see Section 24.1) in every CHUNK_WRITE it issues against the mirror's data servers. Its purpose is to satisfy the 32-bit-per-field budget of chunk_guard4 while preserving the guarantee that concurrent writers on the same file are distinguishable:

* The NFSv4 clientid4 ([RFC8881]) is a 64-bit structured value whose low 32 bits (a slot index) are not guaranteed unique across clients that hold layouts on the same file. Folding clientid4 to 32 bits locally at each client could therefore collide with another client's folded value and violate the uniqueness contract on chunk_guard4.

* Only the metadata server has the information needed to avoid such collisions: it sees every layout it grants on a file and can assign a dense 32-bit ffm_client_id that is guaranteed distinct from the ffm_client_ids assigned to other clients holding concurrent write layouts on the same file. The metadata server MUST assign ffm_client_id subject to this uniqueness rule.

* Because cg_client_id participates in the deterministic tiebreaker for racing writers (see Section 24.1), having the metadata server assign it also lets the metadata server influence which client wins contention by choosing the numeric ordering of the values it hands out. Specific ordering policies are implementation-defined and out of scope for this document, but the protocol mechanism is present.

An ffm_client_id is scoped to the file and layout for which it was granted. A client that holds layouts on two different files may receive two different ffm_client_ids from the same metadata server, and a client that relinquishes and later re-acquires a layout on a

given file MAY be assigned a different ffm_client_id. ffm_client_id does NOT survive a metadata server restart: the metadata server reassigns values as clients reclaim layouts during the grace period.

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

The ffm_striping selects the striping method used by the mirror. The three permissible values are FFV2_STRIPING_NONE (the mirror is not striped), FFV2_STRIPING_SPARSE (stripe units are mapped to the same physical offset on every data server, leaving holes), and FFV2_STRIPING_DENSE (stripe units are packed contiguously on each data server without holes). See Section 9 for the mapping math for each option.

The ffm_striping_unit_size is the stripe unit size used by the mirror. The minimum stripe unit size is 64 bytes. If the value of ffm_striping is FFV2_STRIPING_NONE, then the value of ffm_striping_unit_size MUST be 1.

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

8.9. 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.

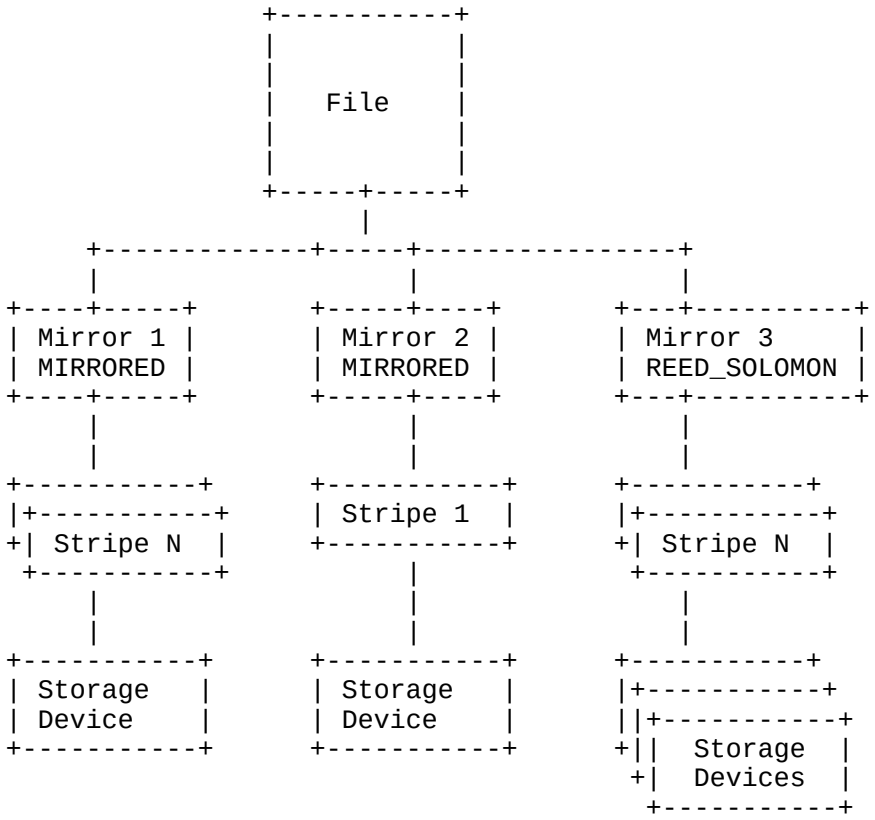


Figure 19: The Relationship between MDS and DSes

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.

8.10. 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	fdp_data	fdp_parity	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).

8.11. 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 = { FFV2_CODING_MIRRORED,
                         FFV2_ENCODING_MOJETTE_SYSTEMATIC,
                         FFV2_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 FFV2_ENCODING_RS_VANDERMONDE and (fdp_data=4, fdp_parity=2). A server without erasure coding might return FFV2_CODING_MIRRORED with (fdp_data=1, fdp_parity=2) for 3-way mirroring.

8.11.1. Codec Negotiation

Because the coding-type registry is expected to grow over time (new erasure codes are added, older ones fall out of favour, vendors register private codes; see Section 28), neither clients nor metadata servers are required to implement every registered codec. The protocol uses ffv2_layouthint4 as the negotiation surface:

Client-side advertisement: A client that wishes to influence codec selection SHOULD send the set of codecs it actually implements in fflh_supported_types. A client MUST NOT claim support for a codec it cannot encode or decode: a false advertisement produces silent data unavailability when the resulting layout is issued.

Metadata-server selection: The metadata server SHOULD select a codec from the client's fflh_supported_types list when the server's policy permits. The server MAY override the hint when its policy dictates a specific codec (for example, per-export objectives); in that case the server issues a layout with the policy-dictated codec and the client MUST either honour it or fail its I/O with NFS4ERR_CODING_NOT_SUPPORTED.

Fallback when no overlap exists: If the server's policy cannot be satisfied by any codec the client supports, the metadata server has three options:

1. Return NFS4ERR_CODING_NOT_SUPPORTED on the LAYOUTGET. The client MAY retry with a different (possibly empty) fflh_supported_types list to learn the server's codec repertoire through the errors returned.

2. Fall back to I/O via the metadata server itself, so the client's reads and writes are satisfied by the MDS translating to the underlying DS codec on the client's behalf (see Section 6.2 for the MDS-I/O fallback). This is correct but serializes all I/O for the codec-ignorant client through a single actor.

3. Route the client through a *translating proxy* that understands both the file's native codec and a codec the client does support. The MDS issues a layout with the proxy's data-server entry carrying FFV2_DS_FLAGS_PROXY and a coding_type the client does support (typically FFV2_CODING_MIRRORED for a minimal NFSv4.2 client, or a flat NFSv3 surface for an NFSv3 client). The proxy encodes and decodes on the fly against the real DSes. This preserves parallel I/O for the codec-ignorant client that the MDS-I/O fallback loses. The proxy registration, directive, and credential-forwarding rules are defined in the companion Data Mover design; this draft defines only the layout-flag surface (FFV2_DS_FLAGS_PROXY in Section 8.4) that makes the proxy visible to the client.

Options (1), (2), and (3) are not mutually exclusive: a given deployment MAY implement any combination. A deployment that supports (3) covers all the clients that (1) and (2) would cover and additionally preserves parallel I/O for codec-ignorant clients.

Runtime codec change: If a metadata server changes its codec policy

after layouts have been issued (for example, a deployment upgrade that retires an older codec), the metadata server MUST recall the affected layouts via CB_LAYOUTRECALL and may re-issue new layouts with the new codec. Clients that do not support the new codec LAYOUTRETURN with NFS4ERR_CODING_NOT_SUPPORTED, and the server either grants a layout using a mutually- supported codec or the client falls back to I/O via the metadata server.

This mechanism deliberately avoids a separate capability-bit handshake at EXCHANGE_ID. ffv2_layouthint4 already provides per-request negotiation surface; adding a session-level capability set would duplicate it and would complicate codec upgrades without additional value, because a client that genuinely upgrades its codec set at runtime can simply update the fflh_supported_types on its next LAYOUTGET.

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 mapping scheme (sparse or dense) is selected per mirror by ffm_striping and is detailed in Section 9. 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 8.14 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. The metadata server registers fffi_stateid with each tight-coupling-capable storage device via TRUST_STATEID (see Section 6.4) before returning the layout; the storage device validates subsequent CHUNK operations against its trust table.

For loose coupling and an NFSv4 storage device, the client MUST use the anonymous stateid to perform I/O on the storage device, because the metadata server stateid has no meaning to a storage device that is not participating in the control protocol. In this case the metadata server MUST set fffi_stateid to the anonymous stateid.

For an NFSv3 storage device (ffdv_version = 3), the tight-coupling model does not apply: Section 7.1 requires ffdv_tightly_coupled to be FALSE whenever ffdv_version equals 3, because NFSv3 has no wire encoding for stateids. The corresponding fffi_stateid element in the ffv2ds_file_info array MUST therefore be the anonymous stateid and is unused; an NFSv3 data server uses the synthetic-uid fencing model (see Section 6.2) rather than a stateid-based trust table.

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 7.1) 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 11.1.1.

8.11.2. 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.

8.11.3. 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.

8.12. 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]).

8.13. 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.

8.14. Handling Version Errors

When the metadata server provides the ffda_versions array in the ff_device_addr4 (see Section 7.1), 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 14.1.1) inside either the LAYOUTRETURN (see Section 18.44 of [RFC8881]) or the LAYOUTERROR (see Section 15.6 of [RFC7862] and Section 15 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.

9. Striping

The flexible file layout type version 2 inherits the dense and sparse striping dispositions defined by the file layout type in Section 13.4 of [RFC8881]. The disposition for a given mirror is selected by the ffm_striping field (see Section 8.8) and applies to every data server in that mirror's ffs_data_servers list. Three values are permitted:

FFV2_STRIPING_NONE: The mirror is not striped. ffm_striping_unit_size MUST be 1 and ffm_stripes MUST contain exactly one stripe. The entire mirror lives on that stripe's single data server list, with no offset transformation.

FFV2_STRIPING_SPARSE: Logical offsets within the file map to the same numeric offset on each data server. A data server that does not own the stripe unit at a given logical offset presents a hole at that offset. This is the simpler model and matches the mental picture of "the file is laid out end-to-end on each data server, but each data server stores only its stripe units".

FFV2_STRIPING_DENSE: Stripe units owned by a given data server are packed contiguously on that data server, with no holes. The logical offset is transformed into a compact physical offset on the target data server. This matches pre-existing deployments that follow the dense layout convention of Section 13.4.4 of [RFC8881].

The mapping math for sparse and dense is given in Figure 22. Common definitions apply to both.

```
L: logical offset within the file (bytes)
U: stripe-unit size in bytes  = ffm_striping_unit_size
W: stripe width               = length of ffs_data_servers
S: stripe size in bytes       = W * U
N: stripe number              = L / S
i: index (0-based) of the data server that owns L
                              = (L / U) mod W
R: byte offset within the stripe unit
                              = L mod U

FFV2_STRIPING_SPARSE:
  physical offset on data server i:
      P_sparse(L) = L
  other data servers see a hole at offset L.

FFV2_STRIPING_DENSE:
  physical offset on data server i:
      P_dense(L) = N * U + R
             = (L / S) * U + (L mod U)
  each data server stores only the stripe units it owns,
  packed contiguously.
```

Figure 22: Sparse and dense stripe mapping math

10. 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 14.1.1).

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.

11. Client-Side Protection Modes

11.1. 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 8.11), 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 11.1.2). 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 11.1.3, during the resilvering, the layout is recalled, and the client has to make modifications via the metadata server.

11.1.1. 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.

11.1.2. Writing to Mirrors

11.1.2.1. Single Storage Device Updates Mirrors

If the FF_FLAGS_WRITE_ONE_MIRROR flag in ffl_flags is set, the client MAY update just 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 11.1.2.3). The client's responsibility with respect to COMMIT is explained in Section 11.1.2.4. The client may choose any one of the mirrors and may use ffv2ds_efficiency as described in Section 11.1.1 when making this choice.

11.1.2.2. 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 MUST 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.

11.1.2.3. 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.

11.1.2.4. 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.

11.1.3. 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.

11.2. 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 an unique identifier generated by the client to describe the current write transaction (see Section 24.1). 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).

The protocol provides two levels of payload integrity, consumed at different points in the read path:

Consistency: A payload is *consistent* when all of the chunks that belong to it carry the same chunk_guard4 value (see Section 24.1). Consistency alone does NOT imply the bytes are free of corruption; it means only that every chunk in the payload came from the same write transaction. A reader detects inconsistency when it assembles a payload and finds differing chunk_guard4 values across chunks.

Integrity: A payload has *integrity* when it is consistent AND every contained chunk passes its CRC32 check. Integrity is the precondition for returning the payload’s data block to the application.

The separation matters because the two checks detect different failure modes. Consistency detects protocol-level failures (racing writers, partial writes, rollback windows); the CRC32 detects byte-level corruption (network errors, media errors, software bugs in the erasure transform). Neither subsumes the other.

The two-level integrity model also reflects a deeper property of distributed writes: *last-writer-wins does not apply to a payload spread across independent data servers.* The ordering of writes arriving at one data server may differ from the ordering arriving at another; the "last" write on DSa may well be the "first" on DSc. The chunk_guard4 CAS primitive (see Section 24.1) resolves this by serializing concurrent writers per chunk rather than by imposing a global order.

The erasure coding algorithm itself might not be sufficient to detect all byte-level errors in the chunks. The CRC32 checks allow the data server to detect chunks with integrity issues; the erasure decoding algorithm can then reconstruct the affected chunks from the remaining integral chunks in the payload.

11.2.1. Encoding a Data Block

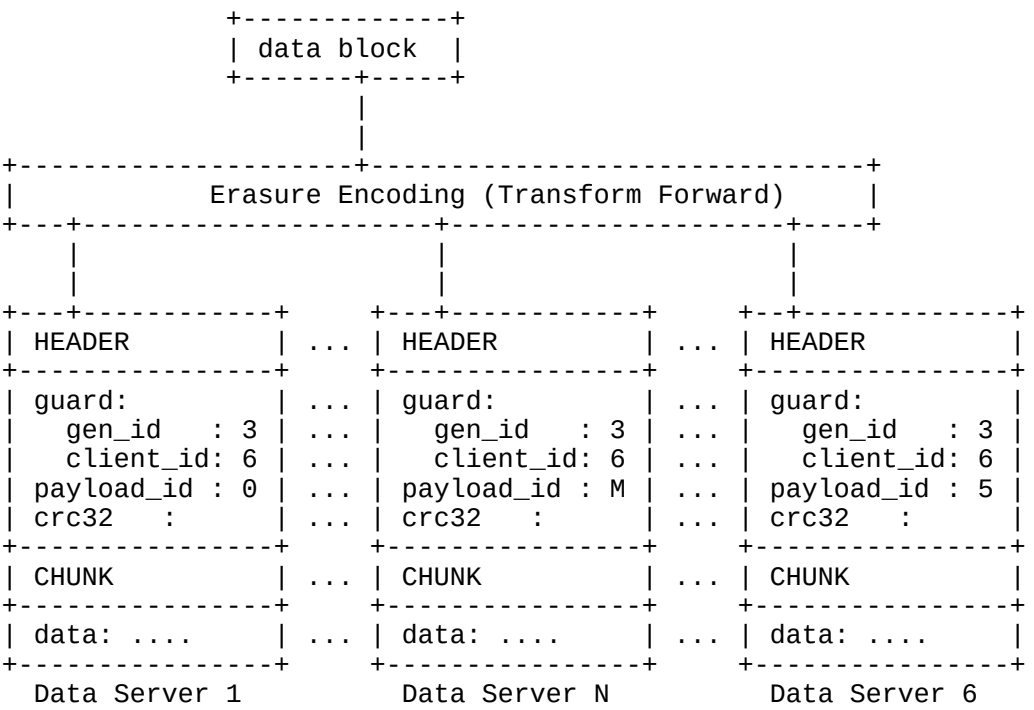


Figure 23: Encoding a Data Block

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 25.10) can encode multiple write_chunk4 into a single transaction, a more accurate description of a CHUNK_WRITE is in Figure 24.

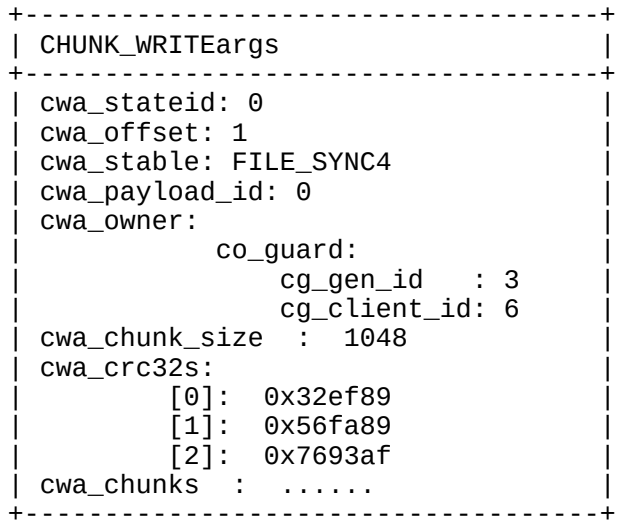


Figure 24: Example of CHUNK_WRITE_args

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.

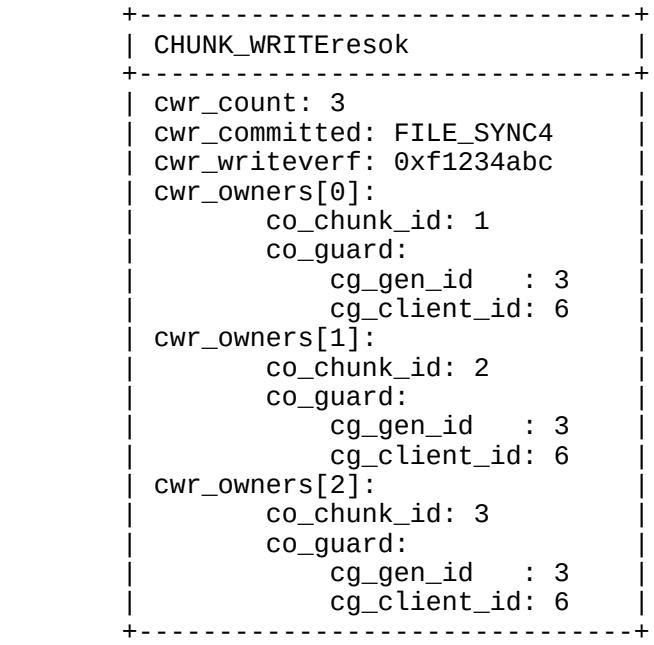


Figure 25: Example of CHUNK_WRITE_res

11.2.1.1. Calculating the CRC32

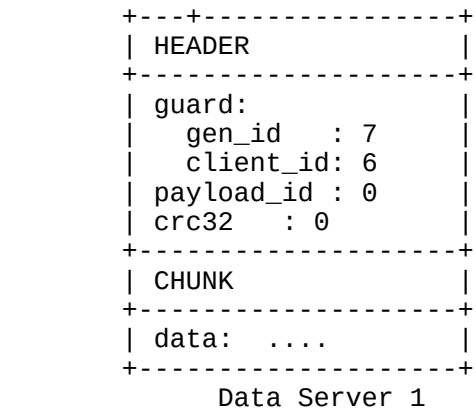


Figure 26: CRC32 Before Calculation

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.

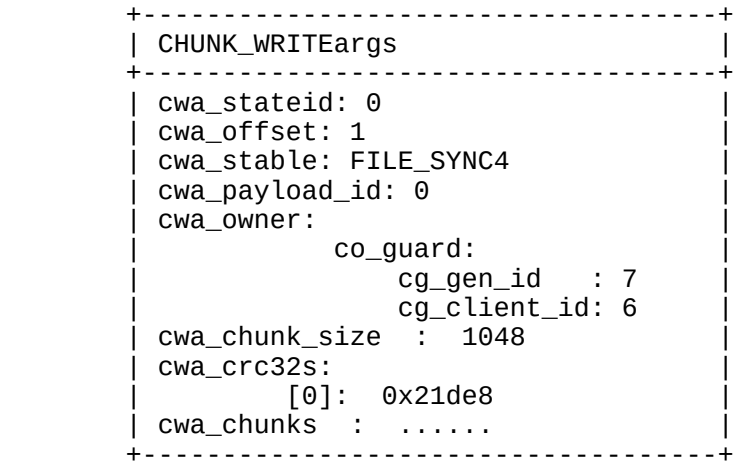


Figure 27: CRC32 After Calculation

11.2.2. Decoding a Data Block

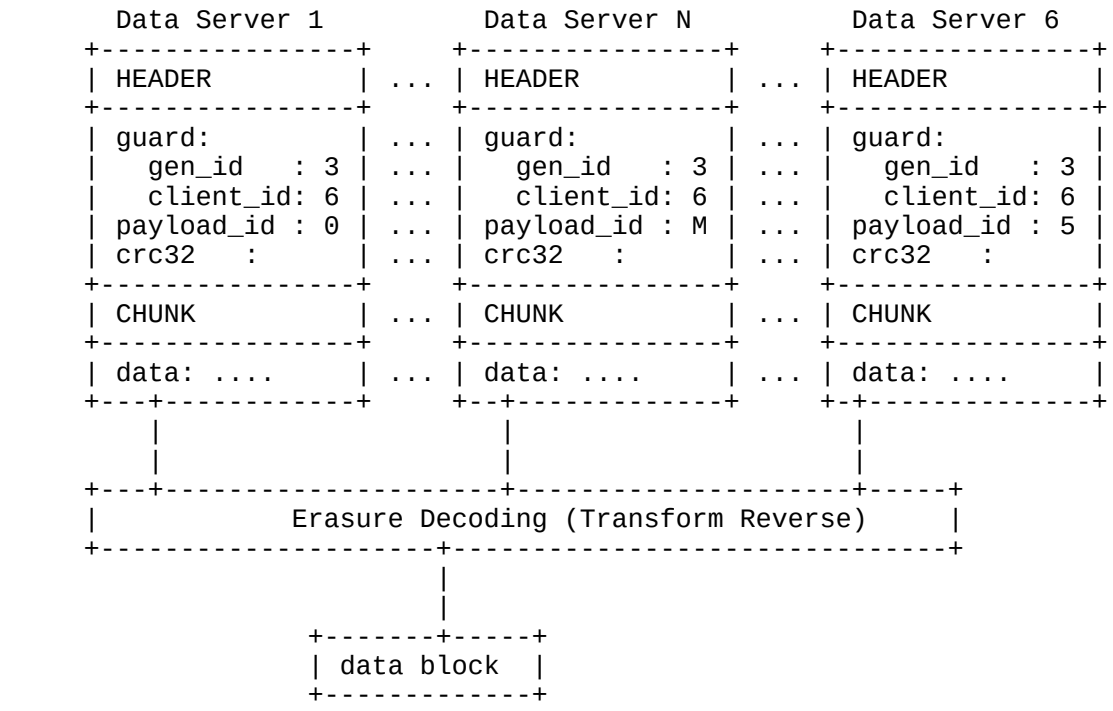


Figure 28: Decoding a Data Block

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.

11.2.2.1. Checking the CRC32

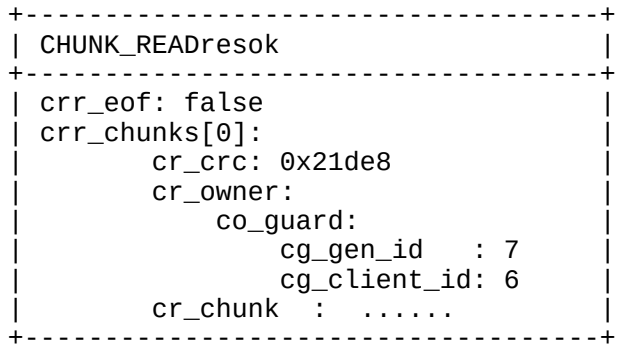


Figure 29: CRC32 on the Wire

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.

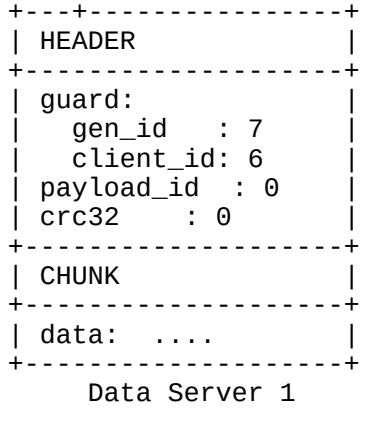


Figure 30: CRC32 Being Checked

11.2.3. 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 over writes 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 11.2.7 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.

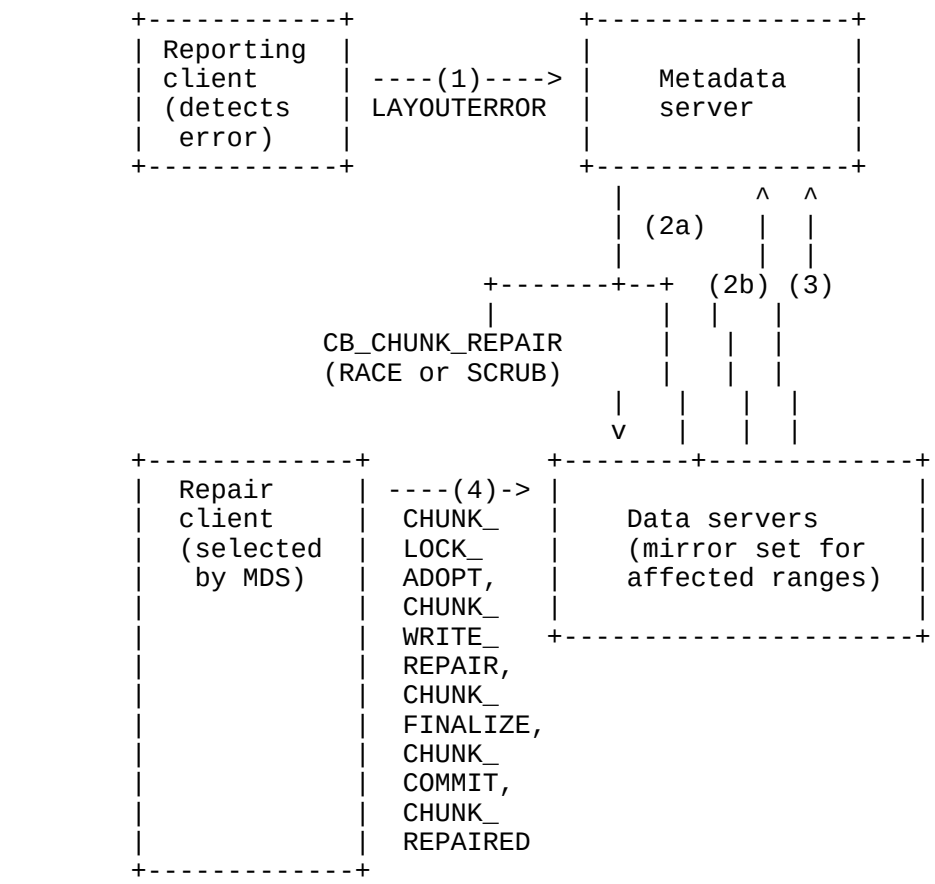
CHUNK_WRITE is a two pass operation in cooperation with CHUNK_FINALIZE (Section 25.3) and CHUNK_ROLLBACK (Section 25.8). It writes to the data file and the data server is responsible for retaining a copy of the old header and chunk. A subsequent CHUNK_READ would return the new chunk. However, until either 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 25.5) 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 25.9).

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 returns NFS4ERR_CHUNK_LOCKED, the client reports the condition to the metadata server via LAYOUTERROR with an error code of NFS4ERR_PAYLOAD_NOT_CONSISTENT.

11.2.4. Selecting the Repair Client

The repair topology involves three actors communicating along distinct paths, as shown in Figure 31.



(1) Reporter LAYOUTERRORs the MDS.
(2a) MDS selects a repair client (may be same as reporter).
(2b) MDS escrows the chunk lock and issues CB_CHUNK_REPAIR.
(3) Repair client adopts the lock and drives the repair.
(4) Repair client issues CHUNK_* ops against the mirror set.

Figure 31: Repair topology

The metadata server is the authority that selects which client (or, in a tightly coupled deployment, which data server) repairs an inconsistent payload. This is analogous to the way the metadata server assigns per-mirror priority via ffv2ds_efficiency (see Section 11.1.1): the protocol does not prescribe the selection algorithm, and each deployment MAY tune its policy.

Implementations MAY consider factors such as:

* Whether a client holds an active write layout on the affected payload (the client most likely to hold surviving shards in cache).

* Whether a client has previously reported consistent shards to the metadata server via LAYOUTSTATS or a prior LAYOUTERROR.

* Whether the layout exposes a data server carrying FFV2_DS_FLAGS_REPAIR as a target for reconstructed shards.

* Network proximity, observed latency, or recent client load -- the same class of information that informs ffv2ds_efficiency.

The selection algorithm is not normative. What is normative is that every client MUST be prepared to:

1. Receive a repair request for a payload that the client does not have an outstanding write layout on, and did not write; and

2. Continue its own workload after reporting NFS4ERR_PAYLOAD_NOT_CONSISTENT without itself being selected to repair the payload it reported.

The metadata server signals the selected client via the CB_CHUNK_REPAIR callback (Section 26.1), which identifies the file, the affected ranges (each with its own triggering nfsstat4), and a wall-clock deadline. A client that receives CB_CHUNK_REPAIR for a file for which it does not already hold a layout MUST acquire a layout via LAYOUTGET before attempting the repair.

Operational expectations for CB_CHUNK_REPAIR: CB_CHUNK_REPAIR is an exceptional path, triggered only by concurrent-writer races or data-server failures. It is not a steady-state operation and its frequency is a function of racing-writer and data-server-failure rates in the deployment rather than of normal client workload. Implementations SHOULD treat the CB_CHUNK_REPAIR handler as rare-path code and avoid over-optimising it. Implementations SHOULD, however, provision enough client-side compute to handle a repair transaction without stalling their foreground I/O, because foreground throughput during repair is the externally observable cost of this callback.

11.2.5. Repair Protocol: Normative vs. Informative

The selection algorithm is non-normative and deployment-tunable. The externally-observable state transitions of the repair flow are normative. The line between the two is drawn at what another party on the wire -- the metadata server, another client, a reader -- can observe. What no other party can see (client-internal ordering, retry policy, whether to CHUNK_READ first to confirm the failure) is left to implementations.

The following requirements are normative. An implementation that violates any of these can leak inconsistency or write-holes into the cluster:

1. *Final state flat.* Every shard in every range identified in a CB_CHUNK_REPAIR MUST reach either the COMMITTED state (repaired) or the EMPTY state (rolled back). No shard is left in PENDING or FINALIZED indefinitely.

2. *Lock before write.* The repair client MUST adopt the lock on every affected range via CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT (Section 25.5) before issuing any CHUNK_WRITE_REPAIR, CHUNK_ROLLBACK, or CHUNK_WRITE on a chunk in that range. The lock on the affected chunks is held continuously from the failure that triggered CB_CHUNK_REPAIR through the adoption; at no point is the range unlocked.

3. *Clear the errored state.* On the reconstruction path, the repair client MUST issue CHUNK_REPAIRED (Section 25.7) after CHUNK_COMMIT. Without it, readers continue to see holes regardless of on-disk state.

4. *Release locks explicitly.* CHUNK_ROLLBACK does not release chunk locks. On the rollback path the client MUST issue CHUNK_UNLOCK (Section 25.9) on each affected chunk. A client that walks away without either completing CHUNK_REPAIRED or issuing CHUNK_UNLOCK holds the locks until lease expiry, blocking progress for other writers.

5. *Deadline honored.* The client MUST drive every range to its final flat state before ccra_deadline, or MUST respond to the CB_CHUNK_REPAIR with NFS4ERR_DELAY (requesting an extension), NFS4ERR_CODING_NOT_SUPPORTED (declining), or NFS4ERR_PAYLOAD_LOST (declaring the data unrecoverable). A deadline that elapses without any of these leaves the metadata server free to re-select; the client MUST NOT continue repair-related CHUNK operations after the deadline without first re-verifying its layout and the chunk lock state.

6. *Terminal return codes.* NFS4ERR_CODING_NOT_SUPPORTED MUST mean "decline; select another client." NFS4ERR_PAYLOAD_LOST MUST mean "the data is not recoverable; do not retry." The metadata server relies on these to decide whether to re-issue.

The following aspects are informative / implementation-defined:

* Choice between the reconstruction path (CHUNK_WRITE_REPAIR) and the rollback path (CHUNK_ROLLBACK) on a given range. The protocol MUST support both; the client MAY use either based on its local state and whether reconstruction is feasible from surviving shards.

* Ordering among multiple affected ranges in a single CB_CHUNK_REPAIR (parallel or serial).

* Whether to issue CHUNK_READ to confirm the failure mode before reconstructing.

* Retry policy on transient CHUNK_WRITE_REPAIR errors below the deadline cutoff.

* How the repair status is surfaced to local filesystem API callers.

11.2.6. Carrying Out the Repair

With the normative framing above, the reconstruction path is:

1. CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT on each affected range (Section 25.5).

2. CHUNK_WRITE_REPAIR (Section 25.11) with the reconstructed data for each inconsistent shard. The client’s chunk_owner4 on this and all subsequent operations is the one it presented in the CHUNK_LOCK ADOPT above; prior owners’ generation ids are now historical.

3. CHUNK_FINALIZE (Section 25.3) and CHUNK_COMMIT (Section 25.1) to persist the repaired shards.

4. CHUNK_REPAIRED (Section 25.7) to clear the errored state.

The rollback path, when reconstruction is not possible:

1. CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT on each affected range.

2. CHUNK_ROLLBACK (Section 25.8) on each affected shard to restore the previously committed content.

3. CHUNK_UNLOCK (Section 25.9) on each shard.

In both paths, the repair client SHOULD target reconstructed shards according to the following fallback order: first, any data server in the layout carrying FFV2_DS_FLAGS_REPAIR; then the data server that reported the failure (the one carrying the failing shard at the range identified by ccr_offset and ccr_count in the CB_CHUNK_REPAIR argument); then, if both of the above are unreachable, a data server carrying FFV2_DS_FLAGS_SPARE. If none of the above are available, the client MUST return NFS4ERR_PAYLOAD_LOST on the CB_CHUNK_REPAIR response.

11.2.6.1. 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.

11.2.6.2. 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 original writer is the typical choice for repair, but the metadata server MAY designate any client according to the rules in Section 11.2.4. A designated client that did not originate the writes MUST follow the rollback path of that section if it cannot reconstruct the payload from surviving shards.

The repair sequence when the selected client is the original writer 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 25.11) 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 25.7) to clear the errored state and make the blocks available for normal reads.

11.2.6.3. 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.

11.2.6.4. 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 according to the rules in Section 11.2.4. The FFV2_DS_FLAGS_REPAIR flag, when present on a data server in the layout, identifies the target data server into which reconstructed shards should be written; it does not by itself identify the repair client. The repair sequence is:

1. The repair client issues CHUNK_LOCK (Section 25.5) 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.

11.2.7. 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.

11.2.8. Whole File Repair

Whole-file repair is the case in which too many data servers have failed, or too many chunks have been lost, for the per-range repair flow defined in Section 11.2.4 to reconstruct the file in place. In this case the metadata server MUST either:

1. Construct a new layout backed by replacement data servers and drive the reconstruction via the *Data Mover* mechanism (a designated data server acts as the source of truth for client I/O

during the transition, pushing reconstructed content to the replacement data servers in the background). The Data Mover mechanism also covers the non-repair cases where a file's layout must change while remaining available to clients -- policy-driven layout transitions, data server maintenance evacuation, administrative ingest, TLS coverage transition, and filehandle-backend migration.

2. If the metadata server has no data-mover-capable data server available, or the surviving shards are insufficient to reconstruct any portion of the file, terminate the affected byte ranges with NFS4ERR_PAYLOAD_LOST (see Section 21.1.5).

The Data Mover mechanism is specified in the companion Proxy Server document [I-D.haynes-nfsv4-flexfiles-v2-proxy-server].

Implementations that do not support the Data Mover mechanism can still perform recovery for cases where per-range repair suffices, using CB_CHUNK_REPAIR (Section 26.1) and the repair client selection rules in Section 11.2.4. Such implementations will surface NFS4ERR_PAYLOAD_LOST on any failure that exceeds per-range repair's reach, including the multi-data- server failure scenarios the Data Mover mechanism is intended to handle.

11.3. 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 32. 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 25.6) calls to the other indexes.

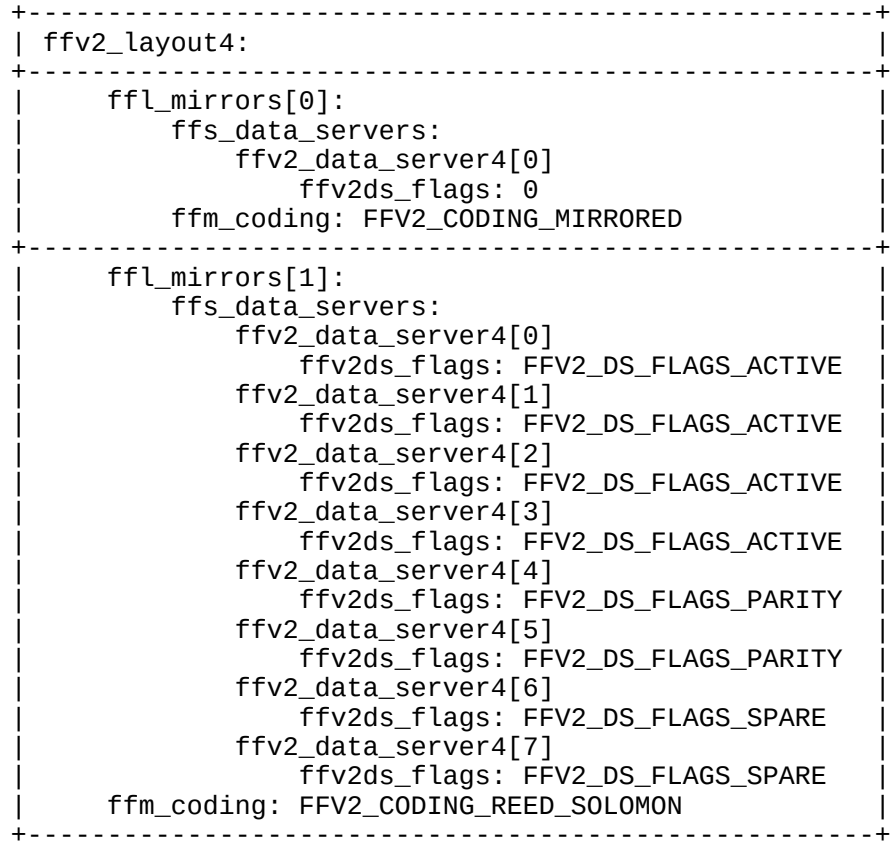


Figure 32: Example of Mixed Coding Types in a Layout

When performing I/O via a FFV2_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 70), 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.

11.4. Reed-Solomon Vandermonde Encoding (FFV2_ENCODING_RS_VANDERMONDE)

11.4.1. 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. [Plank97] is a tutorial treatment of RS coding in RAID-like systems and is the recommended background reading for implementers unfamiliar with the construction used here.

11.4.2. 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.

11.4.3. Encoding Matrix

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

1. Construct a (k+m) x k Vandermonde matrix V where V[i][j] = j^i in GF(2^8).

2. Extract the top k x 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.

11.4.4. 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, 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.

11.4.5. 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 x k sub-matrix S of the encoding matrix E by selecting the rows corresponding to the available shards.

3. Compute S_inv = S^(-1) using Gaussian elimination in 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 GF(2^8) multiplications.

11.4.6. 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 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.

11.4.7. 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

11.5. Mojette Transform Encoding (FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC)

11.5.1. 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.

11.5.2. Grid Structure

Data is arranged as a P x 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       (columns per row)
Q = k           (rows = data shards)
```

11.5.3. 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}

11.5.4. 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 = abs(p) * (Q - 1) + P

The byte size of the projection is B * W.

11.5.5. Katz Reconstruction Criterion

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

SUM(i=1..n) |q_i| >= Q OR SUM(i=1..n) |p_i| >= P

When all q_i = 1, the q-sum simplifies to n >= Q.

11.5.6. 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).

11.5.7. 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 proceeds via the corner-peeling algorithm of [NORMAND]:

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 >= 8).

11.5.8. 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.

11.5.9. 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.

11.6. 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	Zero	Zero	Full decode

cost				
GF operations	Yes (GF(2^8))	No	No	
Recommended k	k <= 6	k >= 4	Archive only	

Table 4: Comparison of erasure encoding types

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

11.7. First-Line Substitution to a Spare

When a client's CHUNK_WRITE to an FFV2_DS_FLAGS_ACTIVE data server fails with a transport-level error, NFS4ERR_IO, NFS4ERR_NOSPC, or any other code that indicates the data server cannot accept the shard, and the layout includes a data server flagged FFV2_DS_FLAGS_SPARE (Section 8.4) that is not already holding a shard for the affected payload, the client MAY substitute the spare for the failing active data server for this write.

Substitution avoids the full metadata-server repair flow. The client issues CHUNK_WRITE to the spare in place of the failing ACTIVE and, if successful, proceeds with CHUNK_FINALIZE and CHUNK_COMMIT against the full set of data servers the payload now resides on (the k-1 healthy ACTIVE plus the substituted SPARE). The spare becomes the i-th shard holder for the affected payload.

The client MUST inform the metadata server of the substitution before returning the layout. This is done via LAYOUTERROR on the failing ACTIVE (reporting the error code the client encountered) in the same compound as, or before, any LAYOUTSTATS reporting of the substitution. The metadata server uses the LAYOUTERROR to decide whether to update the layout in place -- promoting the spare to ACTIVE and demoting the failing ACTIVE to a stale-or-unreachable state -- or to push new layouts via CB_RECALL_ANY to other clients so readers do not continue to consult the failing ACTIVE.

Substitution is optional. A client that does not implement it, or does not have a suitable spare in the layout, falls through to the normal write-hole handling below. Substitution is also not available to clients writing with cwa_stable == FILE_SYNC unless the client is prepared to drive FILE_SYNC semantics on the spare as well; otherwise the substitution silently downgrades the durability contract.

Substitution MUST NOT be used when the existing PENDING state on any shard of the affected payload carries a different chunk_guard4 than the current transaction (the range has been adopted by a repair client already -- the normal repair flow applies and substitution would collide).

11.8. 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.

A single-shard CHUNK_WRITE failure MAY also be handled without CHUNK_ROLLBACK by substituting the failing data server with an FFV2_DS_FLAGS_SPARE, per Section 11.7. This avoids engaging the metadata server's repair flow and is the preferred path on transient single-DS failures when the layout exposes a suitable spare.

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.

When substitution and CHUNK_ROLLBACK are both unavailable, and the payload cannot be reconstructed because too many shards have been lost (for example, a catastrophic multi-DS failure with no spares provisioned), the repair flow ultimately terminates with NFS4ERR_PAYLOAD_LOST; see Section 21.1.5.

12. System Model and Correctness

The design decisions in this document -- centralized coordination through the metadata server, CAS semantics via chunk_guard4, pessimistic lock escrow during repair, and erasure-coded reads from any sufficient subset -- depart visibly from a classical distributed-consensus protocol such as Paxos or Raft. This section states the system model those decisions rest on, the consistency and progress guarantees the protocol provides under that model, and how the protocol relates to (and when it relies on) classical consensus. It is intended as the correctness framing for implementers and reviewers; the normative wire behavior is defined in the preceding sections.

12.1. Wire Semantics vs Implementation

The protocol defines wire semantics, not data-server implementation. The operations introduced in Section 25 (CHUNK_WRITE, CHUNK_FINALIZE, CHUNK_COMMIT, CHUNK_ROLLBACK, CHUNK_LOCK / CHUNK_UNLOCK, CHUNK_READ, CHUNK_REPAIRED, CHUNK_ERROR, CHUNK_HEADER_READ, CHUNK_WRITE_REPAIR) together with the per-chunk state machine (Section 12.4) and the chunk_guard4 CAS (Section 24.1) are the entire surface a peer observes. The data server's internal representation of persistent state is not exposed on the wire, and two data-server implementations that satisfy the same wire semantics MAY differ arbitrarily in their internal structure.

In particular, the protocol does NOT exchange:

* which on-disk layout (log-structured, append-only, in-place-overwrite, external object store, key-value store, or any other) a data server uses to persist chunks;

* whether a data server holds PENDING and FINALIZED chunks in a single blob or in distinct regions;

* how a data server represents the CHUNK_LOCK table, the guard epoch, or the escrow owner;

* whether a data server's chunk retention beyond COMMIT is implemented via shadow blocks, journals, reference counts, or copy-on-write.

This decoupling is deliberate. It lets the protocol accommodate future smart-DS designs -- including designs that integrate more closely with storage back-ends that already provide atomic replace, multi-version concurrency, or internal erasure coding -- without protocol revisions, provided the wire semantics are preserved.

Conversely, a data server implementer is free to pick the representation that best fits the underlying storage stack without fear that some less common implementation choice is disallowed.

The counterpart of this rule is that the wire is the entire contract. Any behavior a client relies on MUST be observable via the operations listed above; any behavior that is not observable (cache state, background scrubbing cadence, internal retry ordering, on-disk layout) is implementation detail and MUST NOT be depended upon.

12.2. Actors and Roles

Three actors participate on behalf of any given file:

pNFS client: Issues CHUNK operations to data servers over the data path; issues LAYOUTGET, LAYOUTRETURN, LAYOUTERROR, and SEQUENCE to the metadata server on the control path. Authenticates to the metadata server via AUTH_SYS, RPCSEC_GSS, or TLS. MAY be selected as a repair client via CB_CHUNK_REPAIR.

Metadata server (MDS): Is the sole coordinator for the file. Grants, renews, and revokes layouts; issues TRUST_STATEID / REVOKE_STATEID / BULK_REVOKE_STATEID to each tight-coupled data server; selects the repair client under the rules in Section 11.2.4; owns the reserved CHUNK_GUARD_CLIENT_ID_MDS escrow identity for in-flight repair.

Data server (DS): Persists chunks and enforces the per-file trust table, the per-chunk guard CAS (chunk_guard4), the per-chunk lock state (including the MDS-escrow owner), and the chunk state machine (EMPTY / PENDING / FINALIZED / COMMITTED). Has no coordinator role. Has no knowledge of the erasure coding type in use for any file: the erasure transform is performed entirely at the client, and the data server stores the resulting chunks without interpreting their contents.

The protocol does NOT mandate how a data server implements the chunk state machine or stores PENDING chunks. An implementation MAY use per-client staging files, a single append-only instance file with an index, a separate metadata-header file paired with a blocks file, a log-structured store, or any other representation that preserves the normative semantics (the EMPTY / PENDING / FINALIZED / COMMITTED transitions, the chunk_guard4 CAS, lock continuity across revocation, and the integrity checks). The choice is a data-server implementation concern and is transparent to clients and the metadata server.

Each file is owned by exactly one metadata server at any given instant. Ownership transfer between metadata servers (for example, during MDS failover) is implementation-defined and out of scope for this document; see Section 12.8.

12.3. Failure Model

The protocol assumes:

Crash-stop: Clients, metadata servers, and data servers fail by stopping. A restarted component rejoins the protocol with a fresh epoch and participates in the grace / reclaim path already defined in [RFC8881]. Correct components do not exhibit arbitrary (Byzantine) behavior.

Fail-silent data servers: Data servers report honestly about the state of the data they hold. The protocol detects on-disk bit rot via CRC32 (see Section 25.10) but does not defend against a data server that deliberately lies about whether a chunk is COMMITTED or what its contents are. Byzantine data servers are explicitly outside the trust model; see Section 12.9.

Authenticated writers and their own data: An authenticated client may write arbitrary (even semantically-invalid) bytes into chunks it owns. The CRC32 check detects transport corruption, not adversarial content. This matches the existing NFSv4 authorization model: once you have write access, you may write anything.

Network partitions: The protocol is partition-tolerant at the cost of availability during the partition window. A client partitioned from a data server recovers via LAYOUTERROR and may be issued a new layout (possibly against a spare, see Section 11.7). An MDS partitioned from a data server eventually renews trust entries on reconnection; in the interim, the data server returns NFS4ERR_DELAY for affected stateids (see Section 6.4.8). Message loss is bounded by RPC retransmit; eventual delivery is assumed once the partition heals.

Split-brain scenarios (in which a partitioned minority of the data servers in a mirror set attempts to make progress independently of the majority) cannot drive inconsistent writes to COMMITTED state. The chunk_guard4 CAS on each write requires the guard value from a successor chunk to strictly advance the guard value of its predecessor; on partition heal, any writes attempted on the minority side are detected by the majority because their guard values do not satisfy the CAS precondition, and those writes are discarded. When reconciliation is impossible -- for example, the

erasure code has lost too many shards across both sides of the partition to reconstruct any single consistent generation -- the repair flow terminates with NFS4ERR_PAYLOAD_LOST (see Section 21.1.5), which is terminal for the affected ranges.

Lease bound: All state held by a data server on behalf of a metadata server is bounded by the TRUST_STATEID expiry (see Section 6.4.6). An orphaned entry will eventually expire even if the metadata server never returns.

12.4. Chunk State Machine

Each chunk on a data server occupies exactly one of four states. The transitions below are the complete set; any implementation of the data server’s chunk state table MUST admit these transitions and no others.

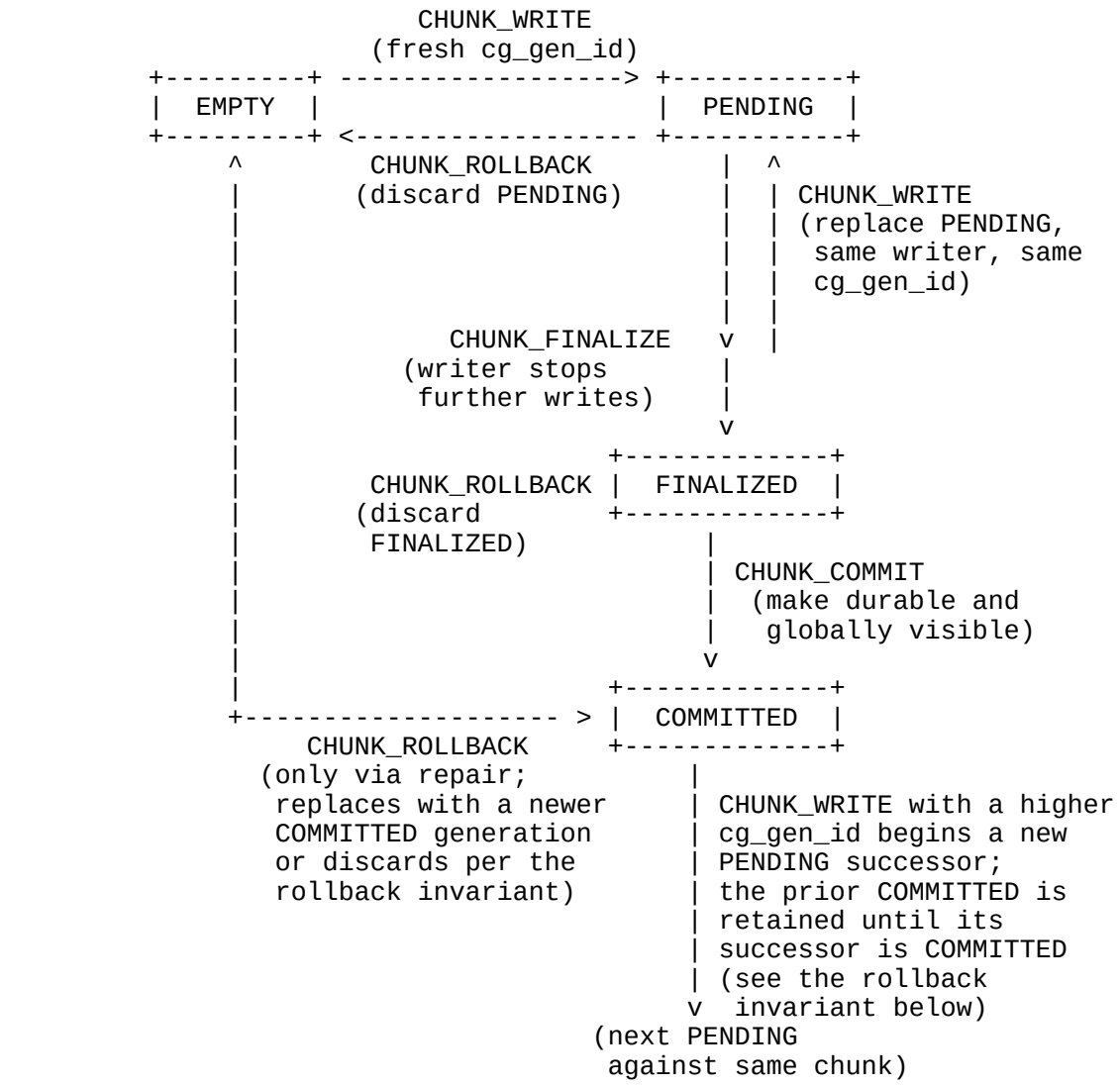


Figure 33: Chunk lifecycle on the data server

States:

EMPTY: The chunk has no payload. CHUNK_READ returns a zero-filled result; CHUNK_WRITE against an EMPTY chunk is the first write.

PENDING: The chunk has payload accepted by CHUNK_WRITE but not yet

finalized. Not visible to CHUNK_READ (see Section 12.5). Further CHUNK_WRITEs from the same writer MAY replace the payload in place (same cg_gen_id).

FINALIZED: The writer has signalled via CHUNK_FINALIZE that it will send no more CHUNK_WRITEs for this generation. Still not visible to CHUNK_READ, but a candidate for CHUNK_COMMIT.

COMMITTED: The chunk is durable and globally visible. Subsequent CHUNK_READs return this content until a newer COMMITTED generation replaces it. A higher-generation PENDING successor MAY exist concurrently; the rollback invariant in Section 12.5 requires the data server to retain the COMMITTED content while that successor exists.

Transitions are driven by the operations named on the arrows. CHUNK_ROLLBACK against a COMMITTED chunk is used only on the repair path (see Section 25.8) and replaces the chunk with a newer COMMITTED generation chosen by the repair client, rather than returning the chunk to EMPTY.

12.5. Consistency Guarantees

The protocol provides *per-chunk linearizability on COMMITTED state*:

1. Once CHUNK_COMMIT returns success to a writer for a given chunk, every subsequent CHUNK_READ whose stateid postdates the COMMIT observes either that writer's data or the data of a later committed write. A reader MUST NOT observe a rolled-back write as if it had committed.

2. Concurrent writers on the same chunk in multi-writer mode serialize via chunk_guard4. On guard conflict one writer succeeds; the other receives NFS4ERR_CHUNK_GUARDED and MUST either abandon the write or re-read and retry. At most one generation becomes COMMITTED per serialized decision.

3. During repair, the chunk's lock is held continuously -- first by the original writer, then transferred to the MDS-escrow owner on REVOKE_STATEID, and finally adopted by the repair client via CHUNK_LOCK_FLAGS_ADOPT. No writer that did not hold the lock may observe or mutate the chunk. The invariant "a chunk with a live lock has exactly one logical owner at any instant" is preserved across revocation.

Across multiple chunks the protocol makes *no multi-chunk atomicity or ordering guarantee*. A reader that reads chunk A at one offset and chunk B at another MAY observe A's new value and B's old value

simultaneously. Applications that require multi-chunk atomicity MUST layer it above this protocol -- for example, via file-level checksums, application-level generation fields, or external transaction managers.

The chunk is the unit of atomicity. Two properties follow:

1. Chunk-aligned writes do not interfere. Two concurrent writers whose writes cover disjoint chunks -- even writes that cover adjacent chunks -- never race. Each write terminates independently at COMMITTED per the per-chunk linearizability rule above.

2. Sub-chunk overlapping writes from different writers produce chunk-resolution-granularity contention. When two concurrent writers target overlapping byte ranges within a single chunk, chunk_guard4 resolves them: one writer's entire chunk-generation wins and becomes COMMITTED; the other writer sees NFS4ERR_CHUNK_GUARDED and is expected to re-read and retry if it wishes to apply its change on top of the winning generation (see Section 21.1.4). The protocol does NOT produce byte-level merges of overlapping sub-chunk writes: the losing writer's bytes are not preserved as a partial update within the winning generation.

Applications that require byte-level write merging or sub-chunk ordering guarantees MUST serialize such writes externally, for example via NFSv4 byte-range locks ([RFC8881], Section 12). The chunk size that bounds the atomicity unit for a given file is the product of ffm_striping_unit_size and the stripe width W in Figure 22; applications can query fattr4_coding_block_size (see Section 23.1) to learn the effective chunk size and align their writes accordingly.

This choice -- chunk-boundary atomicity rather than stripe- or block-boundary atomicity -- is load-bearing for the rest of the consistency story: the chunk_guard4 CAS evaluates at the chunk level, the PENDING / FINALIZED / COMMITTED state machine is per chunk, CHUNK_LOCK is per chunk, and repair via CB_CHUNK_REPAIR operates on chunks. A different atomicity boundary would require redefining those primitives, which this revision does not.

Erasure-coded reads: A reader of an erasure-coded file reconstructs the plaintext from any sufficient subset of k shards of the (k+m)-shard stripe; the guard values on those shards MUST agree. Shards with stale guards are ignored. This is not a quorum read in the Paxos sense -- there is no voting on a value; there is only reconstruction of the single value identified by the current guard.

Rollback invariant: The data server MUST retain the prior FINALIZED or COMMITTED content of a chunk while any successor PENDING chunk exists. A corollary of this rule is the *lowest-guard-recoverable* property: as long as at least k data servers in the mirror set retain the chunk at some generation G or lower, the payload that was COMMITTED at generation G (or earlier) can be reconstructed. This is the correctness basis for CHUNK_ROLLBACK (see Section 25.8): rollback does not synthesize data, it simply selects the lowest-generation chunks whose guards agree across the mirror set and discards the higher-generation PENDING or FINALIZED chunks that triggered the rollback. The protocol never relies on locating or reconstructing data from outside the mirror set.

Visibility of non-committed state: PENDING and FINALIZED chunks MUST NOT be globally visible. CHUNK_READ returns only COMMITTED content; a CHUNK_READ whose target chunk is currently PENDING or FINALIZED sees the predecessor COMMITTED content (or an EMPTY chunk if none exists), not the in-progress successor. A writer observing its own PENDING or FINALIZED chunk MAY receive the in-progress content on the same stateid that produced it, but no other stateid -- on the same or a different client -- sees it. The retention window that makes the prior COMMITTED content available to CHUNK_READ and to CHUNK_ROLLBACK is itself bounded; see Section 12.6 for the normative scoping rule.

12.6. Ownership and Scope of Retained Prior Content

The rollback invariant in Section 12.5 requires a data server to retain the prior FINALIZED or COMMITTED content of a chunk while any successor PENDING chunk exists. That retained content -- sometimes informally called the "safe buffer" -- is not global state. It is scoped to the stateid that wrote the PENDING successor, and its retention and visibility are governed by that owning stateid’s lease.

Owner: The data server MUST record, alongside each PENDING chunk, the owning stateid (the stateid presented on the CHUNK_WRITE that produced the PENDING). This is the owning writer’s stateid; it identifies the client and openowner/lockowner that the data server will release the PENDING to on CHUNK_FINALIZE or CHUNK_COMMIT, and that the MDS will treat as the authoritative owner for purposes of Section 12.7.

Visibility: Before transition to COMMITTED, the PENDING content is visible only on the owning stateid. A CHUNK_READ presenting any other stateid (from the same client or a different client) MUST observe the predecessor COMMITTED or EMPTY state, not the PENDING successor. This is the normative form of the "non-committed data MUST NOT be globally visible" rule in the Visibility bullet above.

Retention window: The data server MUST retain the predecessor COMMITTED (or FINALIZED) content that the PENDING is superseding for as long as the owning stateid's lease is valid. If the owning stateid's lease expires without the PENDING reaching COMMITTED, the retention obligation for that PENDING ends (see Section 12.7 for the scavenger rule that drives demotion). If the PENDING does reach COMMITTED, the new COMMITTED generation supersedes the prior one under the standard rollback invariant and its own retention is governed by any newer PENDING successor.

The practical effect is that the "safe buffer" for a chunk is not an unbounded chunk-global state but a per-writer window bounded by that writer's lease. The data server always has a rule for discarding retained prior content -- it is the owning stateid's lease expiry -- so a chunk cannot accumulate indefinitely many retained generations even in the presence of dropped or partitioned writers.

12.7. Progress and Termination

Under the failure model above, the protocol guarantees the following progress properties:

Data-path progress: If all mirrors are reachable and none are failed, a CHUNK_WRITE followed by CHUNK_FINALIZE followed by CHUNK_COMMIT completes in O(1) round trips independent of cluster size. In particular, there is no consensus round, no leader election, and no quorum voting on the write itself. The three operations MAY be amortized across compounds: a steady-state writer sending a series of CHUNK_WRITEs can piggyback the CHUNK_FINALIZE of the previous write on the compound that carries the next write (for example, SEQUENCE + PUTFH + CHUNK_FINALIZE + CHUNK_WRITE), reducing the data-path happy case to a single round trip per CHUNK_WRITE rather than three. The CHUNK_COMMIT for the final write in a sequence MAY similarly ride on the CLOSE compound. These compound-packing optimizations are permitted by the normal NFSv4.2 compound rules and require no protocol extensions.

Repair termination: Every CB_CHUNK_REPAIR completes in bounded time. The client selected as the repair client either:

1. returns NFS4_OK for every range in ccra_ranges (repair succeeded), or

2. returns NFS4ERR_PAYLOAD_LOST for one or more ranges (the erasure code lost too many shards to reconstruct; the data is permanently unrecoverable), or

3. fails to respond within the ccra_deadline, in which case the metadata server MUST re-select under the rules in Section 11.2.4 or MUST declare the ranges lost.

NFS4ERR_PAYLOAD_LOST is terminal for the affected ranges. The protocol makes no further attempt to recover them.

Eventual trust-table convergence: After a metadata server restart, each data server's trust table converges to the metadata server's view within one metadata-server lease period. Entries that the metadata server does not re-issue expire naturally via tsa_expire; entries that the metadata server does re-issue transition from pending-revalidation back to active on the next TRUST_STATEID (see Section 6.4.8).

Orphaned PENDING scavenger: A PENDING chunk whose owning stateid (see Section 12.6) has expired without transition to FINALIZED or COMMITTED is an orphan. The metadata server MUST drive demotion of orphaned PENDINGs so that no chunk remains in a non-terminal state indefinitely:

1. When an owning stateid's lease expires, the metadata server identifies every PENDING chunk owned by that stateid (either from its own bookkeeping or by query against the data server) and issues the control-plane operations needed to demote each PENDING.

2. Demotion replaces the PENDING with the predecessor COMMITTED (or EMPTY) content that the data server has been retaining under Section 12.6. The data server MUST NOT wait for a separate client action before performing the demotion.

3. Any CHUNK_LOCK held in escrow on behalf of the expired stateid (see Section 24.1.3) is released after an MDS-defined grace period. The grace period exists to let a recovering client reclaim its lock via the grace / reclaim path defined in [RFC8881]; on expiry of the grace period without reclaim, the lock becomes available for new CHUNK_LOCK_FLAGS_ADOPT acquirers.

The scavenger timeout (the delay between lease expiry and demotion) is implementation-defined but SHOULD be tied to the metadata server lease period so that it composes naturally with existing NFSv4 grace / reclaim semantics. A scavenger timeout shorter than the lease risks racing an in-progress client reclaim; a timeout substantially longer than the lease extends the retention budget without a commensurate benefit.

The protocol does NOT guarantee progress if the metadata server is unavailable for longer than its lease period -- this is the standard NFSv4 lease assumption and is inherited unchanged.

12.8. Relation to Classical Consensus

Classical consensus protocols (Paxos, Raft, Viewstamped Replication) solve the problem of reaching agreement among mutually-distrusting replicas in the absence of a trusted coordinator. They typically cost two or three round trips per decision, require a majority of replicas to be live and reachable for progress, and impose the overhead of leader election and log replication.

This protocol is not a consensus protocol and does not attempt to be. Its approach instead is:

1. *Designated coordinator.* The metadata server is the coordinator for a file. Clients accept the MDS's authority for layout grants, stateid registration, repair client selection, and revocation. This assumption is the same one made by [RFC8434] and all pNFS layout types to date.

2. *Per-chunk CAS, not per-chunk voting.* Concurrent writes on the same chunk serialize via chunk_guard4 as a CAS primitive (see Section 24.1). No replica vote is required; the data server that owns the chunk evaluates the guard locally and rejects stale writes with NFS4ERR_CHUNK_GUARDED.

3. *Pessimistic locks off the critical path.* CHUNK_LOCK is used only during repair, never on the normal write path. Lock escrow (see Section 24.1.3) preserves the "exactly one owner" invariant across stateid revocation without requiring a consensus round to elect the next owner.

4. *Erasure-coded reads replace quorum reads.* A reader reconstructs from any k of k+m shards with matching guards. No voting is needed because there is no disagreement to resolve: the guard identifies the single generation that was committed.

The result is a data path with O(1) round-trip cost independent of the number of replicas, and a repair path whose cost is bounded by the number of affected chunks rather than by the cluster size.

Metadata-server high availability is orthogonal. Deployments that require a highly-available metadata server MAY replicate metadata-server state across multiple metadata server instances using classical consensus (Raft, Paxos, or equivalent). Such replication is implementation-defined; from a pNFS client's perspective a highly-

available metadata server looks like a single metadata server that occasionally resets its session and triggers grace-period reclaim, and the client's behavior is already specified by [RFC8881]. This protocol neither requires nor precludes such an implementation.

12.9. Non-Goals

For clarity, the protocol explicitly does not provide:

* *Byzantine fault tolerance.* A data server that deliberately misreports its state, or a client that bypasses its own authentication, is outside the trust model. Deployments requiring Byzantine tolerance MUST add it in a layer above or below this protocol.

* *Metadata server high availability.* Single-MDS-per-file is the protocol model. MDS HA, if deployed, is implemented below the wire protocol and transparent to clients.

* *Cross-file atomicity.* Writes to multiple files are not atomic at the protocol level. File-system-level transactions are not defined.

* *Multi-chunk atomicity within a single file.* COMMITs on distinct chunks are independent. A reader may observe a partial write across chunks; applications must layer their own consistency if they need otherwise.

* *Global linearizability across unrelated files.* Each file's COMMITTED state is linearizable in isolation; no total order is defined across files.

* *Authenticated malicious client protection.* An authenticated client may write garbage into its own chunks with a correctly computed CRC32; see Section 27.1. The CRC32 check is a transport-integrity check, not an adversarial-integrity check.

* *General-purpose intent primitive.* Christoph Hellwig observed at IETF 121 (November 2024) that the intent-based pattern used here (CHUNK_WRITE -> CHUNK_FINALIZE -> CHUNK_COMMIT with CHUNK_ROLLBACK as the abort path) has potential applicability beyond erasure coding -- for example, as a general multi-target atomic-ish write primitive. This document scopes the mechanism to erasure coding: the on-wire operations carry erasure-coding-specific semantics (chunk_guard4, mirror-set repair, per-codec geometry), and generalising the primitive is explicit future work. Protocol extensions that reuse the intent / finalize / commit pattern in other contexts are not precluded by this document but are not defined by it.

13. NFSv4.2 Operations Allowed to Data Files

In the Flex Files Version 1 Layout Type ([RFC8435]), the data path between client and data server was NFSv3 ([RFC1813]); the operations a client sent to a data file were limited to READ, WRITE, and COMMIT, and the operations the metadata server sent on its control plane to the data server were limited to GETATTR, SETATTR, CREATE, and REMOVE. An NFSv4.2 data server, as used by the Flex Files Version 2 Layout Type, exposes a much larger operation set. This section defines which operations a client MAY send to a data file, which operations the metadata server MAY send, and which operations a data server MUST reject.

The restrictions below apply only to operations directed at a data file on a data server. Clients retain the full NFSv4.2 operation set for files visible through the metadata server, including the operations prohibited below (RENAME, LINK, CLONE, COPY, ACL-scoped SETATTR, and so on). The metadata server MAY internally use operations on data files that clients MUST NOT send, as part of its control-plane duties for the file (see Section 12.2).

13.1. Control Plane: Metadata Server to Data Server

When the metadata server acts as a client to a data server, it is managing the data file on behalf of the metadata file’s namespace. A data server MUST support the following operations on data files when issued by the metadata server:

* SEQUENCE, PUTFH, PUTROOTFH, GETFH ([RFC8881] Sections 18.46, 18.19, 18.21, 18.8): session and filehandle plumbing.

* LOOKUP ([RFC8881] Section 18.15): runway pool directory traversal.

* GETATTR ([RFC8881] Section 18.7): reflected GETATTR after a write layout is returned, and any other attribute queries the metadata server needs to reconcile its cached view.

* SETATTR ([RFC8881] Section 18.30): data file truncate for MDS-level SETATTR(size) fan-out, synthetic uid/gid rotation for fencing, and mode-bit initialisation on runway assignment.

* CREATE ([RFC8881] Section 18.4): runway pool file creation.

* REMOVE ([RFC8881] Section 18.25): cleanup on MDS file unlink.

* OPEN, CLOSE ([RFC8881] Sections 18.16, 18.2): used by the metadata server when it acts as a client to the data server for InBand or proxy I/O.

* EXCHANGE_ID, CREATE_SESSION, DESTROY_SESSION, BIND_CONN_TO_SESSION, DESTROY_CLIENTID ([RFC8881] Sections 18.35, 18.36, 18.37, 18.34, 18.50): control-session management. The metadata server sets EXCHGID4_FLAG_USE_PNFS_MDS in its EXCHANGE_ID. A data server that supports the tight-coupling control protocol (see Section 6.4.2) identifies the metadata server's session by EXCHGID4_FLAG_USE_PNFS_MDS and accepts TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID on that session.

* TRUST_STATEID (Section 25.12), REVOKE_STATEID (Section 25.13), BULK_REVOKE_STATEID (Section 25.14): the MDS-to-DS tight-coupling trust-table control operations.

The metadata server MAY also use other NFSv4.2 operations on data files as implementation-defined control-plane actions (for example, COPY or CLONE to migrate a data file between data servers during a data mover operation). The list above is the minimum set a Flex Files v2 data server MUST support for the metadata server's use.

13.2. Data Path: Client to Data Server

A pNFS client with an active Flex Files v2 layout MUST restrict the operations it issues against data files to the operations defined below. A data server MUST reject any other operation on a data file with NFS4ERR_NOTSUPP.

13.2.1. Session and Identity Plumbing

Required for all protection modes:

* SEQUENCE, PUTFH, GETFH, PUTROOTFH ([RFC8881] Sections 18.46, 18.19, 18.8, 18.21).

* EXCHANGE_ID, CREATE_SESSION, DESTROY_SESSION, BIND_CONN_TO_SESSION, DESTROY_CLIENTID ([RFC8881] Sections 18.35, 18.36, 18.37, 18.34, 18.50).

* RECLAIM_COMPLETE ([RFC8881] Section 18.51).

* SECINFO, SECINFO_NO_NAME ([RFC8881] Sections 18.29, 18.45): discovery of acceptable security flavours on the data server.

These operations are baseline NFSv4.2 session plumbing and are supported on data files as on any NFSv4.2 file.

13.2.2. GETATTR on a Data File

GETATTR MAY be issued by a client against a data file. The primary use case is repair: a repair client selected by CB_CHUNK_REPAIR (Section 26.1) may need to query the per-server file size or allocation state when reconstructing a payload, and the data mover described informally in Section 12.2 similarly benefits from attribute queries on surviving mirrors. Diagnostic use is also permitted.

Clients MUST NOT treat GETATTR values returned by a data server as authoritative for any file attribute (size, timestamps, owner, mode, ACL, and so on). The metadata server is the sole authority for file attributes. Values returned by a data server reflect the per-server data file instance only and MAY diverge from the metadata server's view, particularly during a write layout's lifetime or during a Data Mover transition. A client that uses a data-server GETATTR result to determine the file's visible size will observe inconsistencies.

13.2.3. SETATTR on a Data File

Clients MUST NOT issue SETATTR against a data file. A data server MUST reject a client SETATTR with NFS4ERR_NOTSUPP.

Attribute changes on data files MUST be reconciled with the metadata server's view and cannot be applied unilaterally by a client. A client that wants to truncate, change the mode, change ownership, or otherwise modify attributes on a file MUST issue SETATTR to the metadata server for the file's MDS handle; the metadata server fans the change out to the data files as a control-plane operation.

This rule explicitly covers truncate (SETATTR with size in the bitmap): a client MUST NOT truncate a data file directly. Similarly, a client MUST NOT issue DEALLOCATE against a data file; see the next subsection.

13.2.4. Mirrored Data Files (FFV2_CODING_MIRRORED)

For a mirror whose ffm_coding_type_data is FFV2_CODING_MIRRORED (see Section 8.8), client operations on the data file follow the same pattern as the File Layout Type in [RFC8881] Section 13.6 and the Flex Files v1 Layout Type in [RFC8435]:

Required:

* READ ([RFC8881] Section 18.22).

* WRITE ([RFC8881] Section 18.32).

* COMMIT ([RFC8881] Section 18.3).

Optional (the client MAY send, and the data server MAY support):

* READ_PLUS ([RFC7862] Section 15.10): hole-aware reads.

* SEEK ([RFC7862] Section 15.11): hole and data detection.

* ALLOCATE ([RFC7862] Section 15.1): space reservation hint.

The client MUST NOT send:

* DEALLOCATE ([RFC7862] Section 15.4): hole punching is a metadata-server responsibility; the client issues DEALLOCATE on the metadata-server filehandle, and the metadata server fans out to the data servers as a control-plane operation.

13.2.5. Erasure-Coded Data Files (FFV2_ENCODING_*)

For a mirror whose ffm_coding_type_data is any of the erasure- coding types defined in this document (FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC, FFV2_ENCODING_RS_VANDERMONDE), client operations use the CHUNK_* operations rather than READ / WRITE / COMMIT.

Required for all erasure-coded clients:

* CHUNK_WRITE (Section 25.10).

* CHUNK_READ (Section 25.6).

* CHUNK_FINALIZE (Section 25.3).

* CHUNK_COMMIT (Section 25.1).

* CHUNK_HEADER_READ (Section 25.4).

* CHUNK_LOCK (Section 25.5) and CHUNK_UNLOCK (Section 25.9).

* CHUNK_ROLLBACK (Section 25.8).

Required for clients that participate in repair:

* CHUNK_ERROR (Section 25.2).

* CHUNK_REPAIRED (Section 25.7).

* CHUNK_WRITE_REPAIR (Section 25.11).

Clients MUST NOT send:

* READ, WRITE, COMMIT against an erasure-coded data file. A data server MUST reject these with NFS4ERR_NOTSUPP and MAY log the client for operator attention; this case is almost always a client bug in which the client did not inspect the mirror's ffm_coding_type_data before issuing I/O.

* READ_PLUS, SEEK, ALLOCATE, DEALLOCATE against an erasure- coded data file. Chunk-level allocation is a metadata-server responsibility.

13.2.6. Operations That MUST NOT Be Sent to a Data File

Clients MUST NOT send the following operations to a data server on a data file, regardless of protection mode. A data server MUST return NFS4ERR_NOTSUPP:

* OPEN, CLOSE, OPEN_DOWNGRADE, OPEN_CONFIRM ([RFC8881] Sections 18.16, 18.2, 18.18, 18.20). Opens occur on the metadata server; the stateid obtained there is used on the data path.

* LOCK, LOCKU, LOCKT, RELEASE_LOCKOWNER ([RFC8881] Sections 18.10, 18.11, 18.13, 18.24). Byte-range locks on data files are not supported; erasure-coded files use CHUNK_LOCK, and mirrored files rely on metadata-server coordination.

* DELEGPURGE, DELEGRETURN, WANT_DELEGATION ([RFC8881] Sections 18.5, 18.6 and [RFC7862] Section 15.3). Delegations are issued by the metadata server.

* Any operation whose purpose is to manipulate the file's namespace: RENAME, LINK, SYMLINK, CREATE (at the file- creation use, not MDS runway creation), REMOVE. Namespace operations belong on the metadata server.

* Any ACL-scoped SETATTR or GETATTR bit (FATTR4_ACL, FATTR4_DACL, FATTR4_SACL). Access control on data files is delegated to the metadata server.

* CLONE, COPY, COPY_NOTIFY, OFFLOAD_CANCEL, OFFLOAD_STATUS ([RFC7862] Sections 15.13, 15.2, 15.3, 15.8, 15.9). File-level data migration is a metadata-server responsibility.

* LAYOUTGET, LAYOUTCOMMIT, LAYOUTRETURN, LAYOUTSTATS, LAYOUTERROR, GETDEVICEINFO, GETDEVICELIST ([RFC8881] Sections 18.43, 18.42, 18.44, [RFC7862] Sections 15.7, 15.6, [RFC8881] Sections 18.40, 18.41). Layout operations belong on the metadata server.

* TRUST_STATEID, REVOKE_STATEID, BULK_REVOKE_STATEID (Section 25.12, Section 25.13, Section 25.14). These are MDS-to-DS control-plane operations; a data server rejects them with NFS4ERR_PERM when received on a client session (see Section 6.4.2).

13.3. Callback Path: Data Server to Client

A data server does not call back directly to pNFS clients. Recall notifications and repair coordination flow through the metadata server's backchannel session with the client. The callbacks a client will observe that affect its data files are:

* CB_LAYOUTRECALL ([RFC8881] Section 20.3).

* CB_NOTIFY_DEVICEID ([RFC8881] Section 20.12).

* CB_RECALL_ANY ([RFC8881] Section 20.6).

* CB_CHUNK_REPAIR (Section 26.1).

A data server influences these callbacks only indirectly, via LAYOUTERROR reports the client issues to the metadata server or by returning error codes that prompt the client to report. A data server MUST NOT attempt to send CB_* operations to clients directly.

13.4. Summary Table

Table 5 lists each relevant NFSv4.2 operation and its applicability on a data file in each direction. "required" means the data server MUST support the operation when received on the indicated path; "OPT" means the data server MAY support it and the client MUST tolerate the absence of support; "MUST NOT" means the client MUST NOT send the operation and the data server MUST reject it with NFS4ERR_NOTSUPP; "MAY" means the metadata server MAY use the operation as an implementation-defined control-plane action.

Operation	Client -> DS	MDS -> DS
SEQUENCE, PUTFH, GETFH, PUTROOTFH	required	required
EXCHANGE_ID, CREATE_SESSION, DESTROY_SESSION, BIND_CONN_TO_SESSION, DESTROY_CLIENTID	required	required
RECLAIM_COMPLETE	required	required
SECINFO, SECINFO_NO_NAME	required	MAY
GETATTR	OPT (non-authoritative)	required
SETATTR	MUST NOT	required
LOOKUP, CREATE, REMOVE	MUST NOT	required
READ, WRITE, COMMIT	required (mirrored); MUST NOT (erasure-coded)	MAY
READ_PLUS, SEEK, ALLOCATE	OPT (mirrored); MUST NOT (erasure-coded)	MAY
DEALLOCATE	MUST NOT	MAY
CHUNK_WRITE, CHUNK_READ, CHUNK_FINALIZE, CHUNK_COMMIT, CHUNK_HEADER_READ, CHUNK_LOCK, CHUNK_UNLOCK, CHUNK_ROLLBACK	required (erasure-coded); MUST NOT (mirrored)	not used

CHUNK_ERROR, CHUNK_REPAIRED, CHUNK_WRITE_REPAIR	required (erasure-coded repair clients); MUST NOT (mirrored)	not used
OPEN, CLOSE, OPEN_DOWNGRADE, OPEN_CONFIRM	MUST NOT	OPT (proxy I/O)
LOCK, LOCKU, LOCKT, RELEASE_LOCKOWNER	MUST NOT	MUST NOT
DELEGPURGE, DELEGRETURN, WANT_DELEGATION	MUST NOT	MUST NOT
RENAME, LINK, SYMLINK	MUST NOT	MUST NOT
CLONE, COPY, COPY_NOTIFY, OFFLOAD_CANCEL, OFFLOAD_STATUS	MUST NOT	MAY (data migration)
LAYOUTGET, LAYOUTCOMMIT, LAYOUTRETURN, LAYOUTSTATS, LAYOUTERROR, GETDEVICEINFO, GETDEVICELIST	MUST NOT	MUST NOT
ACL-scoped GETATTR/SETATTR bits	MUST NOT	MAY
TRUST_STATEID, REVOKE_STATEID, BULK_REVOKE_STATEID	MUST NOT	required (tight coupling)

Table 5: NFSv4.2 operations allowed on data files

14. 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 34).

```
   /* 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 34: 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 14.3). 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.

14.1. I/O Error Reporting

14.1.1. ff_ioerr4

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

Figure 35: ff_ioerr4

Recall that [RFC7862] defines device_error4 as in Figure 36:

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

Figure 36: 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.

14.2. Layout Usage Statistics

14.2.1. 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 37: ff_io_latency4

Both operation counts and bytes transferred are kept in the ff_io_latency4 (see Figure 37. As seen in ff_layoutupdate4 (see Section 14.2.2), 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.

14.2.2. 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 38: 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.

14.2.3. 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 39: ff_iostats4

[RFC7862] defines io_info4 as in Figure 39.

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

Figure 40: 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.

14.3. ff_layoutreturn4

```
/// struct ffv2_layoutreturn4 {
///         ffv2_ioerr4     fflr_ioerr_report<>;
///         ffv2_iostats4   fflr_iostats_report<>;
/// };
///
```

Figure 41: 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.

15. 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 14.1), 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.

16. 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 14.2), 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.

17. Flexible File Layout Type Creation Hint

The layouthint4 type is defined in the [RFC8881] as in Figure 42.

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

Figure 42: 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.

18. 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 43: 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).

19. 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 11.1.3.

19.1. 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 44: RCA4 masks for v2

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

Figure 45: 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 46.

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

Figure 46: 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.

20. 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 6.2.

21. 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;
/// const NFS4ERR_PAYLOAD_LOST           = 10101;
///
```

Figure 47: Errors XDR

The new error codes are shown in Figure 47.

21.1. Error Definitions

Error	Number	Description
NFS4ERR_CODING_NOT_SUPPORTED	10097	Section 21.1.1
NFS4ERR_PAYLOAD_NOT_CONSISTENT	10098	Section 21.1.2
NFS4ERR_CHUNK_LOCKED	10099	Section 21.1.3
NFS4ERR_CHUNK_GUARDED	10100	Section 21.1.4
NFS4ERR_PAYLOAD_LOST	10101	Section 21.1.5

Table 6: Error Definitions

21.1.1. 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.

21.1.2. 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.

21.1.3. 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.

21.1.4. 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.

21.1.5. NFS4ERR_PAYLOAD_LOST (Error Code 10101)

Returned by a repair client on the CB_CHUNK_REPAIR response (ccrr_status) to indicate that the identified ranges cannot be repaired and the underlying data is no longer recoverable. Causes include: too few surviving shards to meet the reconstruction threshold (Katz criterion for Mojette, any k-of-(k+m) subset for Reed-Solomon Vandermonde), inability to roll back to a previously committed payload because that payload is also lost, or exhaustion of all FFV2_DS_FLAGS_SPARE and FFV2_DS_FLAGS_REPAIR data servers available in the layout.

On receipt, the metadata server MUST NOT retry the repair by selecting a different client -- the payload is damaged and the metadata server transitions the affected file or byte range into an implementation-defined damaged state. Operator notification and restore-from-snapshot are out of scope for this specification.

NFS4ERR_PAYLOAD_LOST is distinct from NFS4ERR_DELAY (transient; metadata server MAY extend the deadline or re-select) and from NFS4ERR_IO (per-operation failure; metadata server MAY retry or re-select). Only NFS4ERR_PAYLOAD_LOST is terminal.

21.2. Operations and Their Valid Errors

The 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].

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_INVAL, 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_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_ROLLBACK		NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_UNLOCK		NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_INVAL, 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
TRUST_STATEID		NFS4_OK, NFS4ERR_BADXDR,
		NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,
		NFS4ERR_INVAL, NFS4ERR_NOFILEHANDLE,
		NFS4ERR_NOTSUPP, NFS4ERR_PERM,
		NFS4ERR_SERVERFAULT
REVOKE_STATEID		NFS4_OK, NFS4ERR_BADXDR,
		NFS4ERR_BAD_STATEID, NFS4ERR_DELAY,
		NFS4ERR_INVAL, NFS4ERR_NOFILEHANDLE,
		NFS4ERR_NOTSUPP, NFS4ERR_PERM,
		NFS4ERR_SERVERFAULT
BULK_REVOKE_STATEID		NFS4_OK, NFS4ERR_BADXDR, NFS4ERR_DELAY,
		NFS4ERR_NOTSUPP, NFS4ERR_PERM,
		NFS4ERR_SERVERFAULT

Table 7: Operations and Their Valid Errors

21.3. Callback Operations and Their Valid Errors

The callback operations and their valid errors are presented in Table 8. 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	NFS4_OK, 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_PAYLOAD_LOST,
	NFS4ERR_SERVERFAULT, NFS4ERR_STALE

Table 8: Callback Operations and Their Valid Errors

21.4. Errors and the Operations That Use Them

The operations and their valid errors are presented in Table 9. 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
NFS4ERR_PAYLOAD_LOST	CB_CHUNK_REPAIR

Table 9: Errors and the Operations That Use Them

22. EXCHGID4_FLAG_USE_ERASURE_DS

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

Figure 48: The EXCHGID4_FLAG_USE_ERASURE_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 48) 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 10. Further, this support is orthogonal to the Erasure Coding Type selected. The data server is unaware of which type is driving the I/O.

23. New NFSv4.2 Attributes

23.1. Attribute 89: fattr4_coding_block_size

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

Figure 49: XDR for fattr4_coding_block_size

The new attribute fattr4_coding_block_size (see Figure 49) 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.

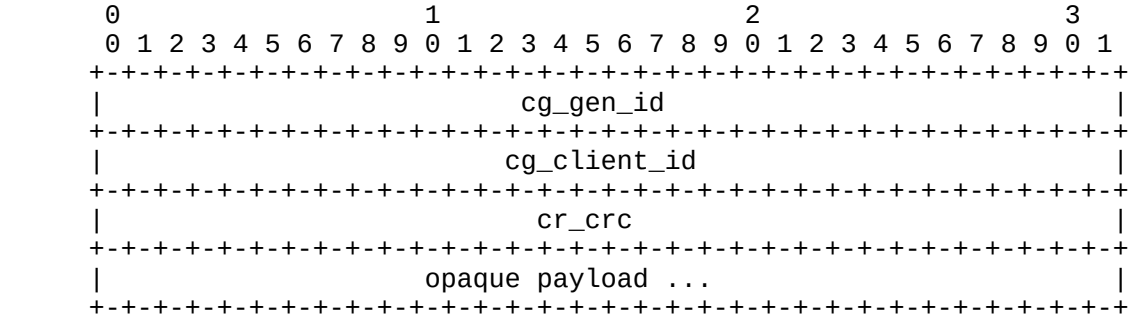
24. New NFSv4.2 Common Data Structures

24.1. chunk_guard4

```
/// const CHUNK_GUARD_CLIENT_ID_MDS  = 0xFFFFFFFF;
///
/// struct chunk_guard4 {
///     uint32_t   cg_gen_id;
///     uint32_t   cg_client_id;
/// };
```

Figure 50: XDR for chunk_guard4

On the wire, a single CHUNK_WRITE carries the 8-byte header followed by the opaque payload, as shown in Figure 51. The payload length is carried separately in the CHUNK_WRITE4args cwa_chunks<> slot; the diagram shows the per-chunk framing only.



Bytes 0-3: cg_gen_id (per-chunk generation counter)
Bytes 4-7: cg_client_id (owning-client short id)
Bytes 8-11: cr_crc (CRC32 over the opaque payload)
Bytes 12-N: opaque payload (encoded shard; variable length)

Figure 51: Per-chunk wire layout

The chunk_guard4 (see Figure 50) is effectively a 64-bit value identifying a specific write transaction on a specific chunk. It has two fields:

cg_gen_id: A per-chunk monotonic generation counter. Each chunk’s

gen_id starts at 0 when the chunk is first written and is incremented on each successful write by any client. cg_gen_id is NOT a timestamp -- the protocol does not rely on a global clock, and no interpretation of cg_gen_id as a wall-clock value is supported. cg_gen_id values are NOT comparable across distinct chunks; a given cg_gen_id is only meaningful within the scope of a single chunk on a single file.

cg_client_id: A 32-bit value established by the metadata server at the time the client's layout is granted (see Section 8.8 and ffm_client_id). The metadata server MUST assign distinct cg_client_id values to distinct clients that hold concurrent write layouts on the same file. cg_client_id is opaque with respect to client identity -- a data server MUST NOT interpret its bits as naming or ordering clients in any external sense. The value supports two operations only: equality comparison (to detect whether two chunks were written by the same transaction) and numeric comparison (to implement the tiebreaker rule below).

Uniqueness contract: The pair (cg_gen_id, cg_client_id) uniquely identifies a write transaction on a chunk. Neither field alone is globally unique; two clients MAY independently write with the same cg_gen_id on the same chunk (in particular, both may write with cg_gen_id equal to some prior value + 1), and the cg_client_id is what makes the resulting transactions distinguishable.

Deterministic tiebreaker for concurrent writers: When two or more clients race on the same chunk in the multi-writer mode, the client whose cg_client_id compares numerically lowest wins the race. A data server enforces this by accepting the first CHUNK_WRITE whose guard check succeeds and rejecting later writers with NFS4ERR_CHUNK_GUARDED; across the mirror set, the subset of data servers on which each client wins will vary, but the deterministic tiebreaker ensures all clients agree on which client's write ultimately becomes COMMITTED. A client that lost the race on at least one data server MUST re-read the chunk and MAY retry its write with a refreshed cg_gen_id. A client that detects no forward progress after a bounded number of retries MUST escalate via LAYOUTERROR and the repair coordination flow in Section 11.2.4.

The numeric ordering of cg_client_id values is arbitrary with respect to the clients' external identities -- it is a deterministic total order over the opaque 32-bit values, not a preference ordering over the clients themselves. A deployment that requires a specific client to win a race MUST arrange cg_client_id assignment at the metadata server; the protocol does not provide a preference mechanism at layout-grant time.

24.1.1. Metadata-Server Assignment Rules for cg_client_id

To uphold the uniqueness contract, the metadata server MUST follow these rules when assigning cg_client_id (that is, when populating ffm_client_id at layout-grant time):

* Two clients holding concurrent write layouts on the same file MUST receive distinct cg_client_id values. A client that holds only a read layout need not be assigned a distinct value.

* The reserved sentinel CHUNK_GUARD_CLIENT_ID_MDS (0xFFFFFFFF) MUST NOT be assigned to any client.

* A cg_client_id MAY be reused by the metadata server after the prior holder's layout has been fully returned (via LAYOUTRETURN or revocation). The metadata server SHOULD avoid reusing a cg_client_id within a single lease period to simplify diagnosis of stale writes.

* cg_client_id values do not persist across metadata-server restart. Clients reclaiming layouts during the grace period receive freshly assigned values; the protocol does not rely on any pre-restart assignment surviving.

24.1.2. Data-Server Collision Handling

A (cg_gen_id, cg_client_id) pair that the uniqueness contract would otherwise render unique can nonetheless collide if a client and the metadata server disagree about which cg_client_id the client currently holds, or if a client presents a spoofed cg_client_id. The data server enforces the contract locally:

* If the data server receives a CHUNK_WRITE whose chunk_guard4 has the same (cg_gen_id, cg_client_id) as a chunk already in PENDING, FINALIZED, or COMMITTED state AND the presented payload differs from the retained payload, the data server MUST reject the write with NFS4ERR_CHUNK_GUARDED and SHOULD report the collision to the metadata server via LAYOUTERROR. This situation is a protocol violation on one side of the conversation; the metadata server resolves it by revoking the offending client's layout and selecting a repair client under Section 11.2.4.

* If a client presents CHUNK_GUARD_CLIENT_ID_MDS as cg_client_id in any client-originated operation, the data server MUST reject the operation with NFS4ERR_INVAL (see Section 24.1.3).

* A cg_client_id that does not match any layout the data server has been told about (via TRUST_STATEID) MUST be rejected. Unknown cg_client_id values are treated as stale layouts; the data server returns the error specified in Section 6.4 for unknown stateids.

24.1.3. Reserved cg_client_id Value: CHUNK_GUARD_CLIENT_ID_MDS

The value CHUNK_GUARD_CLIENT_ID_MDS (0xFFFFFFFF) is reserved. It denotes that the chunk lock is held by the metadata server itself, in escrow during a repair coordination sequence (see Section 11.2.4). The data server produces a chunk_guard4 with this cg_client_id when the metadata server revokes the prior holder's stateid while that holder still holds chunk locks; the locks MUST NOT be dropped and are transferred to the MDS-escrow owner instead.

The metadata server does not originate CHUNK_LOCK or CHUNK_WRITE traffic on its own session. Clients MUST NOT present CHUNK_GUARD_CLIENT_ID_MDS as the cg_client_id of any client-originated chunk_guard4 or chunk_owner4. A data server that receives such a value from a client MUST reject the operation with NFS4ERR_INVAL.

The MDS-escrow owner is released only by a CHUNK_LOCK from the client selected via CB_CHUNK_REPAIR, carrying CHUNK_LOCK_FLAGS_ADOPT. See Section 25.5.

24.2. chunk_owner4

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

Figure 52: XDR for chunk_owner4

The chunk_owner4 (see Figure 52) is used to determine when and by whom a block was written. The co_chunk_id is used to identify the chunk 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 24.1), 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 an unique co_guard. I.e., it can be determined which transaction across all data files that a chunk corresponds.

25. New NFSv4.2 Operations

```
///
/// /* New operations for Erasure Coding start here */
///
///  OP_CHUNK_COMMIT        = 78,
///  OP_CHUNK_ERROR         = 79,
///  OP_CHUNK_FINALIZE      = 80,
///  OP_CHUNK_HEADER_READ   = 81,
///  OP_CHUNK_LOCK          = 82,
///  OP_CHUNK_READ          = 83,
///  OP_CHUNK_REPAIRED      = 84,
///  OP_CHUNK_ROLLBACK      = 85,
///  OP_CHUNK_UNLOCK        = 86,
///  OP_CHUNK_WRITE         = 87,
///  OP_CHUNK_WRITE_REPAIR  = 88,
///
/// /* MDS-to-DS control-plane operations for tight coupling */
///
///  OP_TRUST_STATEID       = 90,
///  OP_REVOKE_STATEID      = 91,
///  OP_BULK_REVOKE_STATEID = 92,
///
```

Figure 53: Operations XDR

The following amendment blocks extend the nfs_argop4 and nfs_resop4 dispatch unions defined in [RFC7863] with arms for each of the new operations defined in this document. A consumer that combines this document's extracted XDR with the RFC 7863 XDR applies these amendments at the union's extension point.

```
/// /* nfs_argop4 amendment block */
///
/// case OP_CHUNK_COMMIT: CHUNK_COMMIT4args opchunkcommit;
/// case OP_CHUNK_ERROR: CHUNK_ERROR4args opchunkerror;
/// case OP_CHUNK_FINALIZE: CHUNK_FINALIZE4args opchunkfinalize;
/// case OP_CHUNK_HEADER_READ:
///     CHUNK_HEADER_READ4args opchunkheaderread;
/// case OP_CHUNK_LOCK: CHUNK_LOCK4args opchunklock;
/// case OP_CHUNK_READ: CHUNK_READ4args opchunkread;
/// case OP_CHUNK_REPAIRED: CHUNK_REPAIRED4args opchunkrepaired;
/// case OP_CHUNK_ROLLBACK: CHUNK_ROLLBACK4args opchunkrollback;
/// case OP_CHUNK_UNLOCK: CHUNK_UNLOCK4args opchunkunlock;
/// case OP_CHUNK_WRITE: CHUNK_WRITE4args opchunkwrite;
/// case OP_CHUNK_WRITE_REPAIR:
///     CHUNK_WRITE_REPAIR4args opchunkwriterepair;
/// case OP_TRUST_STATEID: TRUST_STATEID4args optruststateid;
/// case OP_REVOKE_STATEID: REVOKE_STATEID4args oprevokestateid;
/// case OP_BULK_REVOKE_STATEID:
///     BULK_REVOKE_STATEID4args opbulkrevokestateid;
```

Figure 54: nfs_argop4 amendment block

```
/// /* nfs_resop4 amendment block */
///
/// case OP_CHUNK_COMMIT: CHUNK_COMMIT4res opchunkcommit;
/// case OP_CHUNK_ERROR: CHUNK_ERROR4res opchunkerror;
/// case OP_CHUNK_FINALIZE: CHUNK_FINALIZE4res opchunkfinalize;
/// case OP_CHUNK_HEADER_READ:
///     CHUNK_HEADER_READ4res opchunkheaderread;
/// case OP_CHUNK_LOCK: CHUNK_LOCK4res opchunklock;
/// case OP_CHUNK_READ: CHUNK_READ4res opchunkread;
/// case OP_CHUNK_REPAIRED: CHUNK_REPAIRED4res opchunkrepaired;
/// case OP_CHUNK_ROLLBACK: CHUNK_ROLLBACK4res opchunkrollback;
/// case OP_CHUNK_UNLOCK: CHUNK_UNLOCK4res opchunkunlock;
/// case OP_CHUNK_WRITE: CHUNK_WRITE4res opchunkwrite;
/// case OP_CHUNK_WRITE_REPAIR:
///     CHUNK_WRITE_REPAIR4res opchunkwriterepair;
/// case OP_TRUST_STATEID: TRUST_STATEID4res optruststateid;
/// case OP_REVOKE_STATEID: REVOKE_STATEID4res oprevokestateid;
/// case OP_BULK_REVOKE_STATEID:
///     BULK_REVOKE_STATEID4res opbulkrevokestateid;
```

Figure 55: nfs_resop4 amendment block

Operations 78 through 88 (the CHUNK_* operations) are sent by clients to storage devices on the data path. Operations 90 through 92 (TRUST_STATEID, REVOKE_STATEID, BULK_REVOKE_STATEID) are sent by the metadata server to storage devices on the MDS-to-DS control session (see Section 6.4.2); they MUST NOT be sent by pNFS clients.

Operation	Number	Target Server	Description
CHUNK_COMMIT	78	DS (client)	Section 25.1
CHUNK_ERROR	79	DS (client)	Section 25.2
CHUNK_FINALIZE	80	DS (client)	Section 25.3
CHUNK_HEADER_READ	81	DS (client)	Section 25.4
CHUNK_LOCK	82	DS (client)	Section 25.5
CHUNK_READ	83	DS (client)	Section 25.6
CHUNK_REPAIRED	84	DS (client)	Section 25.7
CHUNK_ROLLBACK	85	DS (client)	Section 25.8
CHUNK_UNLOCK	86	DS (client)	Section 25.9
CHUNK_WRITE	87	DS (client)	Section 25.10
CHUNK_WRITE_REPAIR	88	DS (client)	Section 25.11
TRUST_STATEID	90	DS (MDS control)	Section 25.12
REVOKE_STATEID	91	DS (MDS control)	Section 25.13
BULK_REVOKE_STATEID	92	DS (MDS control)	Section 25.14

Table 10: Protocol OPs

25.1. Operation 78: CHUNK_COMMIT - Activate Cached Chunk Data

25.1.1. ARGUMENTS

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

Figure 56: XDR for CHUNK_COMMIT4args

25.1.2. RESULTS

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

Figure 57: XDR for CHUNK_COMMIT4resok

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

Figure 58: XDR for CHUNK_COMMIT4res

25.1.3. 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.

25.1.3.1. Interaction with CHUNK_FINALIZE

CHUNK_COMMIT transitions a chunk from FINALIZED to COMMITTED (see Section 12.4). A chunk MUST have previously been transitioned from PENDING to FINALIZED via CHUNK_FINALIZE before CHUNK_COMMIT is accepted:

* If the target chunk is PENDING (i.e., the writer never issued CHUNK_FINALIZE), the data server MUST reject the CHUNK_COMMIT entry for that chunk with NFS4ERR_PAYLOAD_NOT_CONSISTENT in the corresponding ccr_status slot. The writer is expected to either issue CHUNK_FINALIZE to advance the state or CHUNK_ROLLBACK to abandon the PENDING generation.

* If the target chunk is EMPTY (no generation to commit), the data server MUST reject with NFS4ERR_PAYLOAD_NOT_CONSISTENT for that chunk.

* If the target chunk is already COMMITTED at the generation identified by the cca_chunks entry's cg_gen_id, the CHUNK_COMMIT is idempotent and MUST succeed. Idempotence preserves the NFSv4 COMMIT contract for duplicate-request retransmission.

* If the target chunk is FINALIZED at a different generation than the one named in the cca_chunks entry, the data server MUST reject with NFS4ERR_CHUNK_GUARDED. A client that sees this has lost a race and SHOULD re-read the chunk (see Section 24.1).

The three-step CHUNK_WRITE -> CHUNK_FINALIZE -> CHUNK_COMMIT sequence MAY be pipelined within a single NFSv4.2 compound (see Section 12.7); each operation evaluates the current state of the target chunks independently.

25.1.3.2. Interaction with a Locked Chunk

When a chunk is locked via CHUNK_LOCK (see Section 25.5), CHUNK_COMMIT is permitted only when the submitter owns the lock -- that is, when the stateid carried on the compound matches the lock holder's stateid (or is an CHUNK_LOCK_FLAGS_ADOPT-transferred continuation):

* The owning writer MAY issue CHUNK_COMMIT; the chunk transitions from FINALIZED to COMMITTED normally.

* A non-owning client MUST receive NFS4ERR_CHUNK_LOCKED in the corresponding ccr_status slot. The chunk's state is not changed.

* During repair, the MDS-escrow owner (CHUNK_GUARD_CLIENT_ID_MDS, see Section 24.1.3) holds the lock while the repair client adopts it via CHUNK_LOCK_FLAGS_ADOPT. CHUNK_COMMIT during the escrow window is permitted only to the holder of the adopted lock.

This rule is what Section 12.5 calls "lock continuity across revocation": the COMMIT privilege follows the lock without gaps in which a non-owner could race.

25.2. Operation 79: CHUNK_ERROR - Report Error on Cached Chunk Data

25.2.1. ARGUMENTS

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

Figure 59: XDR for CHUNK_ERROR4args

25.2.2. RESULTS

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

Figure 60: XDR for CHUNK_ERROR4res

25.2.3. 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 25.11) and the repair is confirmed via CHUNK_REPAIRED (Section 25.7).

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.

25.3. Operation 80: CHUNK_FINALIZE - Transition Chunks from Pending to Finalized

25.3.1. ARGUMENTS

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

Figure 61: XDR for CHUNK_FINALIZE4args

25.3.2. RESULTS

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

Figure 62: XDR for CHUNK_FINALIZE4resok

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

Figure 63: XDR for CHUNK_FINALIZE4res

25.3.3. 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 25.8).

25.4. Operation 81: CHUNK_HEADER_READ - Read Chunk Header from File

25.4.1. ARGUMENTS

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

Figure 64: XDR for CHUNK_HEADER_READ4args

25.4.2. RESULTS

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

Figure 65: XDR for CHUNK_HEADER_READ4resok

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

Figure 66: XDR for CHUNK_HEADER_READ4resok

25.4.3. DESCRIPTION

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

25.5. Operation 82: CHUNK_LOCK - Lock Cached Chunk Data

25.5.1. ARGUMENTS

```
/// const CHUNK_LOCK_FLAGS_ADOPT  = 0x00000001;
///
/// struct CHUNK_LOCK4args {
///     /* CURRENT_FH: file */
///     stateid4        cla_stateid;
///     offset4         cla_offset;
///     count4          cla_count;
///     uint32_t        cla_flags;
///     chunk_owner4    cla_owner;
/// };
```

Figure 67: XDR for CHUNK_LOCK4args

25.5.2. RESULTS

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

Figure 68: XDR for CHUNK_LOCK4res

25.5.3. 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 and cla_flags does not include CHUNK_LOCK_FLAGS_ADOPT, 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 11.2.6.3) to coordinate concurrent access to the same block range, and in the repair flow (Section 11.2.4) to transfer lock ownership to a repair client.

The lock is released by CHUNK_UNLOCK (Section 25.9) or implicitly when the client's lease expires.

25.5.3.1. Lock Transfer via CHUNK_LOCK_FLAGS_ADOPT

The CHUNK_LOCK_FLAGS_ADOPT flag in cla_flags requests an atomic transfer of lock ownership to cla_owner for every chunk in [cla_offset, cla_offset+cla_count). The data server MUST perform the transfer as a single atomic step per chunk: there is no window in which the chunk is unlocked. After a successful ADOPT, subsequent CHUNK_WRITE, CHUNK_WRITE_REPAIR, CHUNK_ROLLBACK, and CHUNK_UNLOCK operations MUST present cla_owner as their chunk_owner4.

CHUNK_LOCK_FLAGS_ADOPT is the sole mechanism by which a chunk lock can change hands without first being released. The lock ordering invariant -- that every chunk in a payload transitioning through repair is held by exactly one owner continuously from failure detection to repair completion -- depends on it.

CHUNK_LOCK_FLAGS_ADOPT is valid only when the caller has been selected as the repair client for the range by the metadata server, typically via CB_CHUNK_REPAIR (Section 26.1). A data server that receives CHUNK_LOCK with the ADOPT flag from a client that has not been so designated MAY reject the operation with NFS4ERR_ACCESS. The mechanism by which the data server determines designation is coupling-model dependent:

* In a tightly coupled deployment, the metadata server notifies the data server via the control protocol (e.g., TRUST_STATEID with the new client's stateid or a similar facility).

* In a loosely coupled deployment, the data server MAY rely on the metadata server's authentication of the client and accept ADOPT from any authenticated client holding a current layout that includes the range. The write-hole exposure cost is that a misbehaving client can trigger spurious ownership transfers; the write-hole exposure is bounded by the chunk_guard4 checks that subsequent CHUNK_WRITEs from displaced writers experience.

The current lock holder at the moment of ADOPT MAY be:

1. Another client whose stateid remains valid (for example, a client that has stopped making progress but has not yet lost its lease). The prior owner's PENDING or FINALIZED shards remain on disk until the new owner issues CHUNK_WRITE_REPAIR, CHUNK_ROLLBACK, or CHUNK_COMMIT.

2. The metadata server itself, acting through the CHUNK_GUARD_CLIENT_ID_MDS escrow owner (Section 24.1.3). This occurs when the metadata server has revoked the prior holder's stateid in a tightly coupled deployment.

In either case, ADOPT's effect from the repair client's perspective is the same: after the successful return the caller holds the lock and may drive the range to consistency.

The data server MUST reject CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT if cla_owner's cg_client_id equals CHUNK_GUARD_CLIENT_ID_MDS -- that value is reserved for server production and MUST NOT be presented by a client. The operation returns NFS4ERR_INVAL in that case.

25.6. Operation 83: CHUNK_READ - Read Chunks from File

25.6.1. ARGUMENTS

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

Figure 69: XDR for CHUNK_READ4args

25.6.2. 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;
///     nfsstat4        cr_status;
///     opaque          cr_chunk<>;
/// };
```

Figure 70: XDR for read_chunk4

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

Figure 71: XDR for CHUNK_READ4resok

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

Figure 72: XDR for CHUNK_READ4res

25.6.3. 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 or it returns an empty block for that block.

For example, in Figure 73, the client asks for 4 blocks starting with the 3rd block in the file. The second data server responds as in Figure 74. 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.

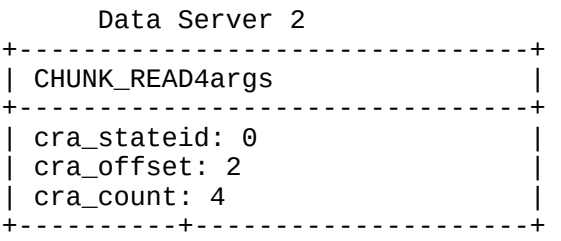


Figure 73: Example: CHUNK_READ4args parameters

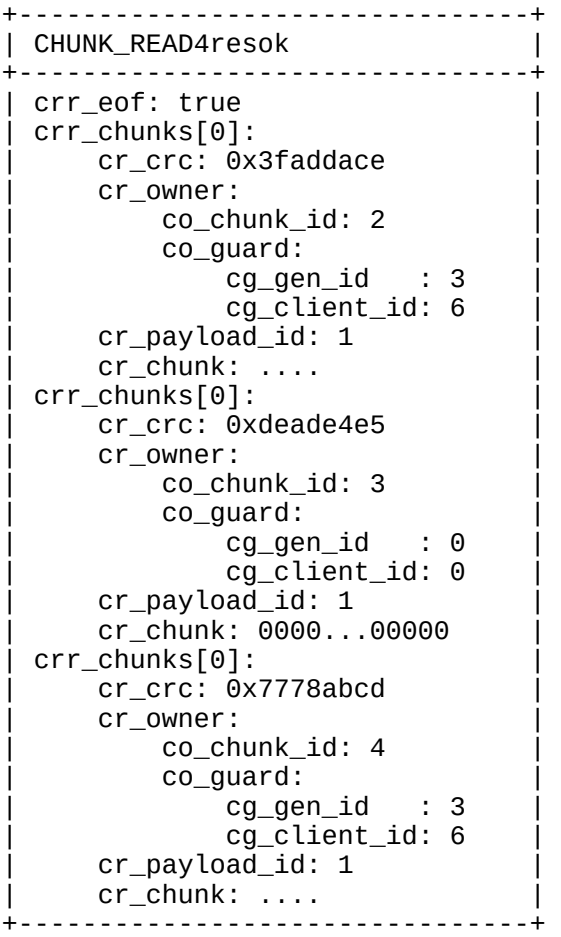


Figure 74: Example: Resulting CHUNK_READ4resok reply

25.7. Operation 84: CHUNK_REPAIRED - Confirm Repair of Errored Chunk Data

25.7.1. ARGUMENTS

```
/// struct CHUNK_REPAIRED4args {
///     /* CURRENT_FH: file */
///     stateid4        cra_stateid;
///     offset4         cra_offset;
///     count4          cra_count;
///     chunk_owner4    cra_owner;
/// };
```

Figure 75: XDR for CHUNK_REPAIRED4args

25.7.2. RESULTS

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

Figure 76: XDR for CHUNK_REPAIRED4res

25.7.3. DESCRIPTION

CHUNK_REPAIRED signals that blocks previously marked as errored (via CHUNK_ERROR, Section 25.2) have been repaired. The repair client writes replacement data via CHUNK_WRITE_REPAIR (Section 25.11), then calls CHUNK_REPAIRED to clear the error state and make the blocks available for normal reads.

The cra_offset and cra_count identify the repaired block range. The cra_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_INVAL.

25.8. Operation 85: CHUNK_ROLLBACK - Rollback Changes on Cached Chunk Data

25.8.1. ARGUMENTS

```
/// struct CHUNK_ROLLBACK4args {
///     /* CURRENT_FH: file */
///     offset4         cra_offset;
///     count4          cra_count;
///     chunk_owner4    cra_chunks<>;
/// };
```

Figure 77: XDR for CHUNK_ROLLBACK4args

25.8.2. RESULTS

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

Figure 78: XDR for CHUNK_ROLLBACK4resok

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

Figure 79: XDR for CHUNK_ROLLBACK4res

25.8.3. 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 cra_offset is the starting block offset and cra_count is the number of blocks to roll back. The cra_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.

25.9. Operation 86: CHUNK_UNLOCK - Unlock Cached Chunk Data

25.9.1. ARGUMENTS

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

Figure 80: XDR for CHUNK_UNLOCK4args

25.9.2. RESULTS

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

Figure 81: XDR for CHUNK_UNLOCK4res

25.9.3. DESCRIPTION

CHUNK_UNLOCK releases the exclusive lock on the block range previously acquired by CHUNK_LOCK (Section 25.5). The cua_owner MUST match the owner that acquired the lock; otherwise the operation returns NFS4ERR_INVAL.

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.

25.10. Operation 87: CHUNK_WRITE - Write Chunks to File

25.10.1. ARGUMENTS

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

Figure 82: 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 83: XDR for CHUNK_WRITE4args

25.10.2. 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 84: XDR for CHUNK_WRITE4resok

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

Figure 85: XDR for CHUNK_WRITE4res

25.10.3. 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.

25.10.3.1. 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 50 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.

25.10.3.2. Per-Block Acceptance Semantics

A CHUNK_WRITE targets a contiguous range of blocks on a single data server. The data server evaluates each block independently and reports the outcome per block in cwr_block_status (see Figure 84):

* Each block is subjected to the guard check (when cwa_guard.cwg_check is TRUE), the cg_client_id validation (see Section 24.1), and any other local preconditions (storage-space limits, tight-coupling trust-table state, etc.).

* Blocks that pass their preconditions are written and their cwr_block_status entry is NFS4_OK. Blocks that fail produce the appropriate error code (NFS4ERR_CHUNK_GUARDED, NFS4ERR_NOSPC, etc.) in the corresponding cwr_block_status slot, and their data is NOT persisted.

* cwr_count reflects only the blocks that were written successfully; failed blocks do not contribute.

* The top-level cwr_status is NFS4_OK when the call itself was structurally valid and the data server could evaluate each block. Per-block failures are reported in cwr_block_status, not by failing the whole operation. The data server returns a top-level error only if it could not evaluate the request at all (for example, NFS4ERR_BADXDR, NFS4ERR_SERVERFAULT).

This is the "continue and report" discipline. It is intentionally not all-or-none: atomicity is already per-chunk (see Section 12.5), so there is no file-level correctness reason to reject the entire compound because of a single chunk guard failure. Per-block reporting gives the client the information it needs to construct a targeted CHUNK_ROLLBACK or CHUNK_WRITE retry that covers only the blocks that failed.

The data server does not hold a file-wide lock across the per-block evaluation. The chunk_guard4 CAS is evaluated atomically per chunk at the point the data server updates that chunk's state, so an interleaving CHUNK_WRITE from a different client that arrives mid-compound will either win its own CAS race (and the losing client sees NFS4ERR_CHUNK_GUARDED for the contested block) or be rejected itself, without introducing data-server-level locking beyond the per-chunk scope.

25.11. Operation 88: CHUNK_WRITE_REPAIR - Write Repaired Cached Chunk Data

25.11.1. 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 86: XDR for CHUNK_WRITE_REPAIR4args

25.11.2. RESULTS

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

Figure 87: XDR for CHUNK_WRITE_REPAIR4resok

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

Figure 88: XDR for CHUNK_WRITE_REPAIR4res

25.11.3. DESCRIPTION

CHUNK_WRITE_REPAIR has the same semantics as CHUNK_WRITE (Section 25.10) 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 25.5) 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 25.7) to clear the error state.

7. The client releases the lock via CHUNK_UNLOCK (Section 25.9).

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.

25.12. Operation 90: TRUST_STATEID - Register Layout Stateid on Data Server

25.12.1. ARGUMENTS

```
/// struct TRUST_STATEID4args {
///     /* CURRENT_FH: file */
///     stateid4        tsa_layout_stateid;
///     layoutiomode4   tsa_iomode;
///     nfstime4        tsa_expire;
///     utf8str_cs      tsa_principal;
/// };
```

Figure 89: XDR for TRUST_STATEID4args

25.12.2. RESULTS

```
/// union TRUST_STATEID4res switch (nfsstat4 tsr_status) {
///     case NFS4_OK:
///         void;
///     default:
///         void;
/// };
```

Figure 90: XDR for TRUST_STATEID4res

25.12.3. DESCRIPTION

TRUST_STATEID registers a layout stateid with the data server so that subsequent CHUNK operations presenting that stateid can be validated against the data server’s per-file trust table. It is the mechanism by which tight coupling (see Section 6.4) is established between the metadata server and the data server for a particular layout.

TRUST_STATEID operates on the current filehandle; a PUTFH naming the data server’s file MUST precede it in the same compound.

tsa_layout_stateid is the stateid the metadata server issued in the LAYOUTGET that produced this layout. It MUST NOT be a special stateid (anonymous, invalid, read-bypass, or current). The sole exception is the capability probe described in Section 6.4.1: when the metadata server sends TRUST_STATEID with tsa_layout_stateid set to the anonymous stateid against the root filehandle, the data server MUST reject the request with NFS4ERR_INVAL. That rejection is the positive response to the probe.

tsa_iomode is the iomode of the layout (LAYOUTIOMODE4_READ or LAYOUTIOMODE4_RW). The data server MAY enforce this against the CHUNK operation presented: a READ-iomode trust entry does not authorize CHUNK_WRITE.

tsa_expire is the absolute wall-clock time at which the trust entry becomes invalid if not renewed. See Section 6.4.6. The data server MUST reject a TRUST_STATEID whose tsa_expire has tv_nseconds >= 10^9 with NFS4ERR_INVAL.

tsa_principal is the client's authenticated identity as verified by the metadata server at LAYOUTGET time. For RPCSEC_GSS clients this is the GSS display name (e.g., "alice@REALM"). For AUTH_SYS and TLS clients, tsa_principal MUST be the empty string, indicating that no principal binding is enforced on subsequent CHUNK operations. See Section 6.4.4.

If the data server receives TRUST_STATEID on a session whose owning client did not present EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID, the data server MUST return NFS4ERR_PERM. The data server MUST NOT process TRUST_STATEID on a regular client session.

If a trust entry already exists for the same tsa_layout_stateid on the same current filehandle, TRUST_STATEID atomically updates tsa_expire and tsa_principal; this is the renewal path (see Section 6.4.6).

At registration time, the data server tags the new trust entry with the identity of the metadata server -- derived from the clientid of the owning client of the control session on which TRUST_STATEID arrived. This tag is consulted by REVOKE_STATEID and BULK_REVOKE_STATEID to ensure that revocation only affects entries registered by the same metadata server (see Section 25.14). In a multi-metadata-server deployment sharing a single data server, each metadata server registers and revokes only its own entries; the tag is opaque to pNFS clients and is not carried on the wire.

25.12.4. RESPONSE CODES

* NFS4_OK: the trust entry is registered (or updated).

* NFS4ERR_BADXDR: arguments could not be decoded.

* NFS4ERR_BAD_STATEID: tsa_layout_stateid was a special stateid other than the anonymous stateid on the root filehandle.

* NFS4ERR_DELAY: the data server is temporarily unable to process the request; the metadata server SHOULD retry.

* NFS4ERR_INVAL: tsa_layout_stateid was the anonymous stateid and the current filehandle is not the root filehandle; tsa_expire is malformed; or the current filehandle is a directory (except in the capability-probe case).

* NFS4ERR_NOFILEHANDLE: no current filehandle is set.

* NFS4ERR_NOTSUPP: the data server does not implement TRUST_STATEID. This is the capability-probe response (see Section 6.4.1).

* NFS4ERR_PERM: the request arrived on a session whose owning client did not present EXCHGID4_FLAG_USE_PNFS_MDS.

* NFS4ERR_SERVERFAULT: the data server failed while processing the request.

25.13. Operation 91: REVOKE_STATEID - Revoke Registered Stateid on Data Server

25.13.1. ARGUMENTS

```
/// struct REVOKE_STATEID4args {
///     /* CURRENT_FH: file */
///     stateid4        rsa_layout_stateid;
/// };
```

Figure 91: XDR for REVOKE_STATEID4args

25.13.2. RESULTS

```
/// union REVOKE_STATEID4res switch (nfsstat4 rsr_status) {
///     case NFS4_OK:
///         void;
///     default:
///         void;
/// };
```

Figure 92: XDR for REVOKE_STATEID4res

25.13.3. DESCRIPTION

REVOKE_STATEID invalidates a single trust entry on the data server. Subsequent CHUNK operations that present the revoked stateid MUST fail with NFS4ERR_BAD_STATEID.

The metadata server calls REVOKE_STATEID in any of the following situations:

* CB_LAYOUTRECALL timeout: the client did not return the layout within the recall timeout. REVOKE_STATEID terminates the client's ability to issue further I/O to the data server without waiting for tsa_expire.

* LAYOUTERROR with NFS4ERR_ACCESS or NFS4ERR_PERM: the data server rejected the client's I/O; the trust entry is stale and must be removed. This mirrors the fencing case in the loose-coupled model.

* Explicit LAYOUTRETURN: the client returned the layout cleanly. The metadata server MAY issue REVOKE_STATEID at this time or MAY rely on tsa_expire; either is correct.

REVOKE_STATEID operates on the current filehandle; a PUTFH naming the data server's file MUST precede it in the same compound. The filehandle and rsa_layout_stateid together identify the trust entry to revoke.

In-flight CHUNK operations that arrived before REVOKE_STATEID completes MAY be allowed to finish. The data server MUST NOT process new CHUNK operations presenting rsa_layout_stateid after REVOKE_STATEID returns.

Lock state (see Section 25.5) held by the revoked stateid is NOT released as part of REVOKE_STATEID; the data server MUST transfer each held lock to the MDS-escrow owner (see Section 24.1.3). Dropping a chunk lock during revocation would permit a write hole and is prohibited; the repair coordination sequence in Section 11.2.4 assumes that locks held by a revoked writer remain held until a repair client adopts them via CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT.

If the data server receives REVOKE_STATEID on a session whose owning client did not present EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID, the data server MUST return NFS4ERR_PERM.

REVOKE_STATEID is scoped to the issuing metadata server's entries (see the tagging rule in Section 25.12). The data server MUST NOT remove an entry that was registered by a different metadata server, even if rsa_layout_stateid happens to match. In a multi-metadata-server deployment, one metadata server therefore cannot revoke another metadata server's entries.

REVOKE_STATEID is idempotent: revoking a stateid that has no matching trust entry (either no entry exists, or the entry was registered by a different metadata server) returns NFS4_OK. The metadata server therefore does not need to track precisely which entries are currently live on which data server in order to revoke safely.

25.13.4. RESPONSE CODES

* NFS4_OK: the trust entry was removed, or no matching entry existed (idempotent).
* NFS4ERR_BADXDR: arguments could not be decoded.
* NFS4ERR_BAD_STATEID: rsa_layout_stateid was a special stateid.
* NFS4ERR_DELAY: the data server is temporarily unable to process the request.
* NFS4ERR_INVAL: rsa_layout_stateid was the anonymous stateid.
* NFS4ERR_NOFILEHANDLE: no current filehandle is set.
* NFS4ERR_NOTSUPP: the data server does not implement REVOKE_STATEID.
* NFS4ERR_PERM: the request arrived on a session whose owning client did not present EXCHGID4_FLAG_USE_PNFS_MDS.
* NFS4ERR_SERVERFAULT: the data server failed while processing the request.

25.14. Operation 92: BULK_REVOKE_STATEID - Revoke All Stateids for a Client

25.14.1. ARGUMENTS

```
/// struct BULK_REVOKE_STATEID4args {
///     clientid4       brsa_clientid;
/// };
```

Figure 93: XDR for BULK_REVOKE_STATEID4args

25.14.2. RESULTS

```
/// union BULK_REVOKE_STATEID4res switch (nfsstat4 brsr_status) {
///     case NFS4_OK:
///         void;
///     default:
///         void;
/// };
```

Figure 94: XDR for BULK_REVOKE_STATEID4res

25.14.3. DESCRIPTION

BULK_REVOKE_STATEID removes every trust entry on the data server that was registered on behalf of the named client. The data server applies this as a scan over its trust table.

The metadata server calls BULK_REVOKE_STATEID in any of the following situations:

* Client lease expiry: when a client's lease on the metadata server expires, the metadata server revokes all of that client's layouts. A single BULK_REVOKE_STATEID replaces the N per-file REVOKE_STATEID compounds that per-entry revocation would require.

* CB_LAYOUTRECALL with LAYOUTRECALL4_ALL: the metadata server is recalling all layouts for a client. BULK_REVOKE_STATEID is the data-server-side complement.

* Metadata server restart cleanup: after the metadata server reconnects to a data server, it MAY issue BULK_REVOKE_STATEID(brsa_clientid = all-zeros) to clear the prior trust table before re-issuing TRUST_STATEID as clients reclaim. See Section 6.4.8.

BULK_REVOKE_STATEID is scoped to the issuing metadata server's entries (see the tagging rule in Section 25.12). The data server MUST NOT affect entries registered by a different metadata server. Consequently, in a multi-metadata-server deployment sharing a single data server, one metadata server cannot clear another metadata server's entries via BULK_REVOKE_STATEID.

The special value with all fields of brsa_clientid set to zero means "revoke every entry owned by the issuing metadata server, regardless of which pNFS client registered it". The data server MUST interpret this value as a clear of the issuing metadata server's entries only, and MUST NOT treat it either as "the pNFS client whose clientid happens to be zero" or as a global table clear across metadata servers.

BULK_REVOKE_STATEID does not operate on the current filehandle; no PUTFH is required in the compound.

If the data server receives BULK_REVOKE_STATEID on a session whose owning client did not present EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID, the data server MUST return NFS4ERR_PERM.

Like REVOKE_STATEID, BULK_REVOKE_STATEID is idempotent (no error is returned if there are no matching entries) and preserves chunk locks held under any revoked stateid by transferring them to the MDS-escrow owner (see Section 24.1.3), rather than dropping them.

25.14.4. RESPONSE CODES

* NFS4_OK: the matching entries were removed, or there were none (idempotent).
* NFS4ERR_BADXDR: arguments could not be decoded.
* NFS4ERR_DELAY: the data server is temporarily unable to process the request.
* NFS4ERR_NOTSUPP: the data server does not implement BULK_REVOKE_STATEID.
* NFS4ERR_PERM: the request arrived on a session whose owning client did not present EXCHGID4_FLAG_USE_PNFS_MDS.
* NFS4ERR_SERVERFAULT: the data server failed while processing the request.

26. New NFSv4.2 Callback Operations

```
///
/// /* New callback operations for Erasure Coding start here */
///
///  OP_CB_CHUNK_REPAIR     = 16,
///
```

Figure 95: Callback Operations XDR

The following amendment blocks extend the nfs_cb_argop4 and nfs_cb_resop4 dispatch unions defined in [RFC7863] with arms for the new callback operation defined in this document.

```
/// /* nfs_cb_argop4 amendment block */
///
/// case OP_CB_CHUNK_REPAIR: CB_CHUNK_REPAIR4args opcbchunkrepair;
```

Figure 96: nfs_cb_argop4 amendment block

```
/// /* nfs_cb_resop4 amendment block */
///
/// case OP_CB_CHUNK_REPAIR: CB_CHUNK_REPAIR4res opcbchunkrepair;
```

Figure 97: nfs_cb_resop4 amendment block

26.1. Callback Operation 16: CB_CHUNK_REPAIR - Request Repair of Inconsistent Chunk Ranges

26.1.1. ARGUMENTS

```
/// enum cb_chunk_repair_reason4 {
///     CB_REPAIR_REASON_RACE  = 1,
///     CB_REPAIR_REASON_SCRUB = 2
/// };
///
/// struct cb_chunk_range4 {
///     offset4         ccr_offset;
///     count4          ccr_count;
///     nfsstat4        ccr_error;
/// };
///
/// struct CB_CHUNK_REPAIR4args {
///     nfs_fh4                     ccra_fh;
///     stateid4                    ccra_layout_stateid;
///     nfstime4                    ccra_deadline;
///     cb_chunk_repair_reason4     ccra_reason;
///     cb_chunk_range4             ccra_ranges<>;
/// };
```

Figure 98: XDR for CB_CHUNK_REPAIR4args

26.1.2. RESULTS

```
/// struct CB_CHUNK_REPAIR4res {
///     nfsstat4           ccrr_status;
/// };
```

Figure 99: XDR for CB_CHUNK_REPAIR4res

26.1.3. DESCRIPTION

CB_CHUNK_REPAIR is sent by the metadata server to request that a selected client repair one or more inconsistent chunk ranges. Selection follows the rules in Section 11.2.4; those rules are normative for how the client MUST respond on receipt of this callback.

The ccra_fh identifies the file whose chunks are inconsistent. The callback compound carries the filehandle directly; there is no preceding PUTFH in callback compounds.

The ccra_layout_stateid carries the recipient client's current layout stateid for the file if one is held. A client that does not hold a layout on ccra_fh MUST ignore ccra_layout_stateid (it will be the anonymous stateid) and MUST acquire one via LAYOUTGET before issuing any CHUNK operation on the ranges.

The ccra_deadline is a wall-clock nfstime4 (seconds and nanoseconds since the epoch, as defined in Section 3.3.1 of [RFC8881]) by which the client is expected to have driven every range to completion (CHUNK_REPAIRED on the reconstruction path, or CHUNK_UNLOCK on the rollback path). Missing the deadline does not corrupt state -- the metadata server MAY re-select another repair client after the deadline elapses -- but a client that has missed the deadline MUST re-verify its layout and the chunk lock state before continuing any repair-related CHUNK operation.

The ccra_reason distinguishes the two flows that cause the metadata server to issue a repair callback:

CB_REPAIR_REASON_RACE: A live-race repair. A client (not necessarily the recipient of this callback) detected a chunk-level inconsistency at write or read time and reported it via LAYOUTERROR. The metadata server is driving repair synchronously because the affected chunk is on the critical path of some I/O. The recipient SHOULD prioritise the callback over background work.

CB_REPAIR_REASON_SCRUB: A background scrub. The metadata server has detected stale or inconsistent payloads during a scheduled integrity sweep and is opportunistically driving repair. No client is currently blocked on these ranges. The recipient MAY schedule the callback at lower priority than CB_REPAIR_REASON_RACE, and MAY return NFS4ERR_DELAY to defer repair to a more convenient time; the metadata server will retry.

The two reasons share all other semantics: the same ccra_ranges encoding, the same response codes, the same deadline contract. Only the priority / retry behaviour differs.

The ccra_ranges array lists every chunk range the metadata server requests the client to repair. Each entry carries its own ccr_error describing the failure mode the client is being asked to remedy. The repair strategy depends on the error code; see Section 11.2.4 for the normative and guidance split.

The metadata server SHOULD keep each CB_CHUNK_REPAIR compound within the back-channel maximum (ca_maxrequestsize) negotiated in CREATE_SESSION (see Section 18.36.3 of [RFC8881]). If the set of affected ranges would exceed that maximum, the metadata server MAY

issue multiple CB_CHUNK_REPAIR callbacks to the same client. Each callback is independent; the client drives each to completion before the deadline on that callback's ranges.

The fact that a range appears in ccra_ranges implies the data server holds a chunk lock on the range (the failure occurred in or around a PENDING or FINALIZED state that established the lock). The repair client MUST use CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT (Section 25.5) to take ownership of the lock before issuing CHUNK_WRITE_REPAIR, CHUNK_ROLLBACK, or CHUNK_WRITE on any chunk in a requested range.

26.1.4. Response Codes

The ccrr_status value returned by the client has the following normative meanings to the metadata server:

NFS4_OK The client has accepted the request and driven every range in this callback to completion (CHUNK_REPAIRED or CHUNK_UNLOCK on every affected chunk). The metadata server clears the repair queue entry.

NFS4ERR_DELAY The client has accepted the request but requires more time. The metadata server MAY extend the deadline by issuing a new CB_CHUNK_REPAIR with a later ccra_deadline, or MAY re-select another client. The client continues to hold any locks it has adopted until the original or extended deadline.

NFS4ERR_CODING_NOT_SUPPORTED The client does not implement the encoding type of the layout and cannot reconstruct. The metadata server MUST NOT retry with the same client and SHOULD select a different client.

NFS4ERR_PAYLOAD_LOST The client has concluded that the identified ranges cannot be repaired -- there are not enough surviving shards to reconstruct and rollback is also impossible. The metadata server MUST NOT retry the repair and transitions the affected ranges into an implementation-defined damaged state. See Section 21.1.5.

All other error codes listed in Table 8 are treated by the metadata server as retriable: the metadata server MAY issue a subsequent CB_CHUNK_REPAIR to the same or a different client. If the client becomes unreachable (no response within the deadline), the metadata server re-selects per Section 11.2.4.

27. 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.

27.1. 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 27.4) or RPCSEC_GSS.

An authenticated client is in the "active attacker" role with respect to its own chunks, in a restricted sense. The data server validates the CRC32 against the bytes the client provided, so an authenticated client that chooses to send semantically-invalid bytes with a correctly computed CRC32 will have those bytes accepted. The residual surface differs per authentication model:

* Under AUTH_SYS with loose coupling, the residual surface is essentially the pre-existing attack surface of NFSv3 writes: any host that can reach the data server with a valid uid can write nonsense to chunks that uid owns. This is the Flex Files v1 authorization model, which Flex Files v2 inherits without modification for this path.

* Under RPCSEC_GSS or TLS with mutual authentication, the residual surface reduces to: only the authenticated client can write nonsense into chunks it owns. Cross-client corruption is prevented because the data server verifies the principal before accepting the write. The remaining attack surface is the client's own integrity: any deployment that relies on data integrity above the wire MUST apply application-level content validation.

Flex Files v2 does not attempt to defend against this authenticated-but-malicious case. The CRC32 mechanism is a transport-integrity check, not a content-integrity check; the system trust model assumes that an authenticated principal is entitled to destroy the content of chunks it owns.

27.2. 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.

27.3. 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 CHUNK operations only succeed after credential verification and return NFS4ERR_ACCESS for unverified callers rather than the more specific error codes.

27.4. 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 6.2. Servers that require transport security SHOULD advertise this via the SECINFO mechanism rather than silently dropping connections.

27.5. RPCSEC_GSS and Security Services

This document does not specify how RPCSEC_GSS [RFC7861] is used between the client and a storage device in the loosely coupled model, and the reasons differ between the two coupling models. Because the loosely coupled model uses synthetic credentials that are managed by the metadata server rather than shared with the storage device, a full RPCSEC_GSS integration would require protocol work (RPCSEC_GSSv3 structured privilege assertions, per [RFC7861]) on all three of the metadata server, the storage device, and the client. In the tightly coupled model the principal used to access the data file is the same as the one used to access the metadata file, so RPCSEC_GSS applies unchanged. The two subsections below treat each model in turn.

27.5.1. 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.

27.5.2. 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.

28. 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 11).

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

Table 11: 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 12).

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 12: 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 13).

Range	Purpose	Allocation Policy
0x0000-0x00FF	Standards Track	IETF Review
0x0100-0x0FFF	Experimental	Expert Review
0x1000-0x7FFF	Vendor (open)	First Come First Served
0x8000-0xFFFE	Private/proprietary	No registration required
0xFFFF	Reserved	--

Table 13: Erasure Coding Type Value Ranges

Standards Track (0x0000-0x00FF) Encoding types intended for broad interoperability. The specification MUST include a complete mathematical description sufficient for independent interoperable implementations (see Section 8.1.1). Allocated by IETF Review.

Experimental (0x0100-0x0FFF) 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) (0x1000-0x7FFF) 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 (0x8000-0xFFFE) Encoding types for use within a single vendor’s ecosystem. No IANA registration is required. Interoperability with other implementations is not expected. To reduce the likelihood of accidental codepoint collisions between independent vendors, implementations SHOULD derive the low-order 15 bits of any value in this range from that vendor’s Private Enterprise Number [IANA-PEN] (for example, by hashing the PEN into the 15-bit space and reserving one well-known offset per codec). The encoding type name SHOULD include an organizational identifier (e.g., FFV2_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 FFV2_CODING_MIRRORED type for Client-Side Mirroring (see Table 14).

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 14: Flexible File Version 2 Layout Type Erasure Coding Type Assignments

28.1. Flag-Word Allocation

This document defines three bitmap spaces -- ffv2_flags4 (see Section 8.2.1), ffv2_ds_flags4 (see Section 8.4), and cwa_flags (see Section 25.10) -- whose allocated bits are enumerated in this document. Following the precedent of ff_flags4 in [RFC8435], IANA does not maintain a registry for any of these bitmap spaces. Future bit allocations are made by a document that updates or obsoletes this one. Implementations MUST treat unknown bits as reserved and MUST NOT assign meaning to them locally.

29. 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 100 to produce the machine-readable XDR description of the flexible file layout type.

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

Figure 100: 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 101.

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

Figure 101: 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.

Implementation Status

Note to RFC Editor: please remove this section prior to publication, per [RFC7942].

This section records the implementation status of this specification at the time of writing. The purpose, per [RFC7942], is to help reviewers evaluate the protocol against running code and to document which parts have been validated end-to-end versus specified on paper.

reffs (MDS and DS) and ec_demo (Client)

Organization: Independent / open source.

License: AGPL-3.0-or-later.

Source: https://github.com/loghyr/reffs (https://github.com/loghyr/reffs).

Implementation: reffs is an NFSv4.2 server written in C that acts as both a metadata server (MDS) and a data server (DS) in a Flex Files v2 deployment. ec_demo is a client-side library with a demonstration driver that exercises the Flex Files v2 data path over NFSv4.2 with all three erasure-coding types defined in this document.

Coverage:

* CHUNK_WRITE, CHUNK_READ, CHUNK_FINALIZE, and CHUNK_COMMIT (the happy-path data-plane operations) are implemented end-to-end and have been exercised against the three codec families (Reed-Solomon Vandermonde, Mojette systematic, Mojette non-systematic).

* The chunk_guard4 CAS primitive, including the conflict-detection and deterministic-tiebreaker rules in Section 24.1, is implemented on both the client and the data server.

* Per-chunk CRC32 integrity checking (see Section 27.1) is implemented end-to-end.

* Per-inode persistent storage of chunk state (PENDING / FINALIZED / COMMITTED) is implemented using write-temp / fdatasync / rename for crash safety.

* The repair data path (CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT, CHUNK_WRITE_REPAIR, CHUNK_REPAIRED, CHUNK_ROLLBACK, and CB_CHUNK_REPAIR) is *specified but not yet implemented* in the prototype. The corresponding operations currently return NFS4ERR_NOTSUPP. A fault-injection test harness is in place to drive the repair path once it is implemented.

* The tight-coupling control protocol (TRUST_STATEID, REVOKE_STATEID, BULK_REVOKE_STATEID) is *specified but not yet implemented*. Data servers advertise loose coupling via ffdv_tightly_coupled = false, and synthetic AUTH_SYS credentials with fencing are used for access control.

Level of maturity: Research-quality prototype. The implementation demonstrates the protocol and has produced the benchmark data summarised below. It is not production-ready; in particular, it does not yet implement the repair path required to tolerate concurrent-writer races or multi-DS failure reconstruction.

Contact: loghyr@gmail.com.

Last update: April 2026.

Interoperability and Benchmarks

The reffs + ec_demo implementation has been benchmarked against itself (no second Flex Files v2 implementation is known to the authors at the time of writing). The benchmark suite exercises four I/O strategies -- plain mirroring, pure striping, Reed-Solomon Vandermonde, Mojette systematic, and Mojette non-systematic -- at five file sizes (4 KB, 16 KB, 64 KB, 256 KB, and 1 MB), at two parity geometries (4+2 and 8+2), and on two platforms (an Apple M4 host

running macOS with a Rocky Linux 8.10 Docker container, and a Fedora 43 native Linux host on aarch64). Each data point is the mean of five measured runs. Data servers run as Docker containers on a single-host bridge network, so absolute latency numbers reflect encoding and RPC fan-out cost with near-zero network latency; real deployments will see higher absolute values but similar overhead ratios.

Selected findings:

* *Erasure-coded write overhead is modest at small and mid sizes.* At 4 KB to 64 KB payloads, all three EC codecs add 14% to 21% write latency relative to plain mirroring. Above 64 KB the encoding cost begins to dominate; at 1 MB Reed-Solomon and Mojette systematic reach approximately +54%, Mojette non-systematic approximately +62%.

* *The dominant write cost is encoding, not fan-out.* A pure-striping variant (6 data shards, no parity) isolates the two costs. At 1 MB, plain mirroring writes in 64 ms, striping in 71 ms (+11%), Reed-Solomon in 103 ms (+60%). Of the 39 ms Reed-Solomon penalty, only 7 ms comes from parallel fan-out; the remaining 32 ms is encoding plus two additional parity RPCs.

* *Reconstruction of a missing data shard is essentially free for systematic codecs at 4+2.* Reed-Solomon and Mojette systematic add 1% to 6% to read latency in degraded-1 mode (one data shard missing, reconstructed from the remaining five). A client that discovers a failed DS at read time can reconstruct transparently with no user-visible latency impact.

* *At 8+2, systematic-codec reconstruction diverges.* Mojette systematic reconstruction overhead stays at approximately +4% at 1 MB, while Reed-Solomon grows to approximately +54% due to the O(k^2) cost of inverting a k x k matrix in GF(2^8). Mojette systematic's back-projection algorithm scales with m (parity count) rather than k (data count) and is therefore preferable at wider geometries.

* *Mojette non-systematic applies a full inverse transform on every read* regardless of whether any shard is missing. At 1 MB this produces approximately 4x read overhead at 4+2 and approximately 7x at 8+2. This codec is suitable only for write-once cold storage where reads are rare; it should not be the default for interactive workloads.

* *Results are platform-independent.* The largest absolute latency delta between macOS M4 and Fedora 43 at 1 MB is 20 ms on writes. Codec ordering, overhead percentages, and qualitative scaling behavior are reproducible across operating systems and Docker implementations.

The benchmarks confirm that the protocol’s central design claims hold in practice: client-side erasure coding is affordable at typical payload sizes; systematic codecs reconstruct missing shards cheaply; and the scaling properties of the three codec families follow directly from their published algorithmic complexities.

The benchmarks also identify two non-goals for deployment: Mojette non-systematic is not a viable general-purpose read codec, and Reed-Solomon at k greater than approximately 6 loses its "reconstruction is free" property. These observations inform the choice of default codec and geometry in implementations that consume this specification.

A full benchmark report with per-size tables, figures, and the platform comparison is available alongside the source code.

Architectural Implication: Cost of Fault Tolerance

The headline question every storage audience asks of an erasure-coding protocol is: "what does it cost when something goes wrong?" The benchmark answer for the recommended operating point is *essentially zero*. Mojette systematic at 4+2 reconstructs a missing data shard with read-latency overhead within run-to-run noise of healthy operation. Mojette systematic at 8+2 holds at approximately +4%.

This shifts the deployment conversation away from "is erasure coding cheap enough to enable" and toward "which codec and geometry minimise the compromise." The compromise that remains is not the cost of fault tolerance; it is the cost of write-time encoding, which is bounded (under 60% at 1 MB, under 25% at 64 KB), and the cost of crash-safe durability via the chunk state machine (see Section 12.5), which is +7% to +22% on writes and +2% to +10% on reads.

Wire-format performance objections raised earlier in the working group’s review of this work are addressed in Section "Design Rationale: Rejected Alternatives": the per-RPC byte-shuffling cost of the original Mojette-specific projection header has been replaced with XDR-encoded chunk metadata (see Section 24.1), so the remaining wire-format cost is the XDR-encoded chunk header itself, which is identical for every codec and is part of the +7% to +22% v2 write overhead measured above.

Design Rationale: Rejected Alternatives

The design of Flex Files v2 went through several iterations between 2024 and 2026 that are recorded here for the benefit of future reviewers and implementers. Each alternative below was considered and rejected, with the specific concern that led to its rejection. Understanding why these approaches were rejected may help reviewers evaluate the current design against a fuller space of possibilities and may guide future extensions or replacements.

Proprietary Projection Header Inside Opaque Payload

The earliest iteration placed a 16-byte Mojette-specific header at the start of the READ/WRITE opaque payload, interpreted in the endianness of the writer's host. This was rejected because:

* It embedded a specific erasure-coding type (Mojette) into the generic replication-method framework, preventing alternate codings from reusing the same wire format.

* The header bytes were not XDR-aligned, which required every implementation to handle endianness explicitly rather than relying on XDR's natural byte order.

* Carrying integrity and identification data inside an opaque disrespected the XDR self-description model that the rest of NFSv4 relies on.

The rejection of this approach at IETF 120 (July 2024) motivated the shift to explicit XDR-encoded chunk headers and the chunk_guard4 structure, both visible in the wire format.

Per-Client Swap Files with MDS MAPPING_RECALL

One proposal split logical and physical chunk addressing: the metadata server maintained a mapping from logical offset to physical location, and the client appended new chunks to a per-client staging file on each data server before asking the metadata server to atomically remap the file to the new chunks. This was rejected because:

* The MAPPING_RECALL operation required to atomically update the mapping would, in a multi-writer deployment, have to recall all outstanding read/write layouts on the file -- grinding the application to a halt during every remap.

* Each client required its own staging file on every data server, producing N clients * M data servers staging files that had to be reconciled on client restart.

* The approach was biased toward correctness at the expense of throughput, which inverted the expected workload mix where single-writer cases dominate.

Server-Side Byte-Range Lock Manager per File

Another proposal relied on byte-range locks obtained by clients before writing, with the lock manager state spread across the data servers. This was rejected because:

* A failed lock holder required a lock manager to arbitrate recovery, effectively reintroducing a centralized decision point for each chunk.

* The lock recall path for HPC checkpoint workloads (many ranks writing disjoint regions) would have required thousands of locks per file, with recall storms on every phase transition.

* The design did not specify how the lock manager itself would be replicated for high availability, deferring the hardest part of the problem.

The current design uses CHUNK_LOCK (see Section 25.5) but only on the repair path, not on the normal write path.

Modified Two-Touch Paxos on Each Chunk

A fully distributed-consensus proposal placed a lightweight (modified two-touch) Paxos round on each chunk write, reaching agreement among the data servers holding the mirror set. This was rejected because:

* The constant-factor cost per write (two or three round trips, leader election overhead, majority quorum requirement) was unacceptable for workloads where single-writer throughput dominates the deployment mix.

* The approach demanded that data servers be peers in a consensus protocol, which is a substantially heavier requirement than being independent chunk stores.

* A majority of (k+m) data servers must be reachable for any progress, which is a strictly stronger availability requirement than the k-of-(k+m) needed for erasure-coded reads.

Working-group feedback on this proposal was uniformly negative. The current design retains the option -- nothing in this specification prevents an implementation from running classical consensus internally among MDS replicas (see Section 12.8) -- but does not require it per write.

Automatic Commit of Empty Chunks

An earlier version included a WRITE_BLOCK_FLAGS_COMMIT_IF_EMPTY flag (later renamed CHUNK_WRITE_FLAGS_ACTIVATE_IF_EMPTY) that automatically committed a write to a previously-empty chunk without a separate CHUNK_COMMIT round trip. The flag is retained in the current design but its scope was narrowed: it is performant in the exclusive-writer case but produces blocks that cannot be rolled back if a racing writer appears concurrently, requiring either hole-punching or an extension of CHUNK_ROLLBACK to work on committed blocks. The narrow scope is documented in the flag's definition; a broader version was rejected because it created rollback liabilities that were disproportionate to the single-RTT savings.

Global Clock or Wall-Clock-Based Generation Counter

An early design used a wall-clock timestamp as the cg_gen_id. This was rejected because:

* No global clock exists among the many clients of a multi-rack deployment. Clock skew can cause a newer write to appear to have an earlier timestamp than an older one.

* Timestamps at millisecond or microsecond resolution are not fine-grained enough to disambiguate bursty writes from the same client.

* Mixing client identity bits into the low-order bits of a timestamp (to make it unique) reduces effective timestamp resolution without providing a useful total ordering.

The current design uses a per-chunk monotonic counter scoped to the chunk on the data server, with cg_client_id as the disambiguator across clients. See Section 24.1.

Layout-Level Generation Counter

Christoph Hellwig proposed at IETF 122 (March 2025) adding a generation counter to the layout itself, transmitted to the data servers alongside each I/O, so that the metadata server could redirect writes to new data servers without issuing a full CB_LAYOUTRECALL storm across every holder of the file. This is a natural extension of the per-chunk cg_gen_id: where cg_gen_id

disambiguates successive writes to the same chunk, a layout-level counter would disambiguate successive placements of the same data. This was rejected because:

* The use case is already covered. CB_CHUNK_REPAIR (see Section 26.1) and the Data Mover / Proxy-DS mechanism (see the companion Data Mover design) together handle mid-layout remap without requiring a layout-level epoch on the wire. CB_CHUNK_REPAIR reaches the specific chunks that need redirection; the Data Mover reaches the broader re-placement case; between them the full remap space is covered.

* Adding a layout-level counter introduces a second, potentially-conflicting epoch alongside cg_gen_id. The CAS semantics on the data server would have to compose the two generations (per-chunk and per-layout), which multiplies the states the data server must reason about without strengthening any guarantee the protocol offers today.

* The CB_LAYOUTRECALL storm that motivated the proposal is a worst-case cost that the current design pays only during a genuine data-server retirement or full re-placement. Partial remaps -- the common case -- already flow through CB_CHUNK_REPAIR + layout refresh on LAYOUTGET without disturbing other holders.

If a future revision determines that layout-level generation is needed, it can be added as a protocol extension: the on-wire surface is additive rather than a replacement, because cg_gen_id's semantics are independent of any outer layout epoch.

Declustered RAID with Dynamic Parity Mapping

Christoph Hellwig raised at IETF 121 (November 2024) the possibility of borrowing from declustered RAID designs: the metadata server maintains, for every fixed-size region of each file, a mapping from logical address to the specific data servers that currently hold that region's data and parity shards; writes do not update chunks in place but instead produce a new parity stripe on a freshly allocated set of data servers, and the mapping is atomically swapped on the metadata server once the new stripe is durable. The attraction is that overwrite is replaced by remap, eliminating the write-hole problem entirely at the cost of moving consistency into the mapping table. This was rejected because:

* The mapping load scales with the file's chunk count, not with the file count. A single large file with billions of chunks produces a billion-entry mapping that the metadata server must maintain with transactional semantics; the overhead is inverted from the usual "a few large files" regime that pNFS is designed for.

* Remapping storms during rebalancing, data-server addition, or data-server failure require atomic updates to many mapping entries at once. Providing those updates with the reasonable-latency bounds required by HPC checkpoint workloads is an open research problem, not a specifiable protocol.

* The approach reintroduces the metadata-server scale bottleneck that client-side erasure coding is designed to avoid: every write traverses the mapping table, and the mapping table is the hot-spot under concurrent writes.

* The mapping table becomes the single point of failure that the rest of the Flex Files architecture works hard to avoid; replicating it with strong consistency requires a consensus protocol on the metadata server, which the current design deliberately does not require (see Section 12.8).

The current design uses fixed per-file chunk placement decided at LAYOUTGET time plus chunk_guard4 CAS for writes, which localises consistency decisions to the chunks being written rather than to a global mapping table.

Working Group Concern: Codec on Every Client

Source

Christoph Hellwig, IETF 120, NFSv4 Working Group session, during the discussion of the original Flexible File Version 2 erasure-coding proposal.

The Question as Asked

Christoph stated that he was "very scared of the implications of having every client be a full participant in a distributed storage system." He pointed out that any erasure-coding or replication protocol that runs at the client requires every client implementation to understand the codec, and that codecs evolve over time as new algorithms appear in the storage research literature. He observed that the same problem appears with replication ("simple two-, three-, four-way replication"): a client power-failure event mid-write leaves the participating data servers in inconsistent states, and the recovery machinery (mirrored logs, write-ahead replay, partial-write

detection) is "a bit of overkill for simple replication."

David Black seconded the concern in the same session, stating that "it’s better to have the data protection algorithm be inside the boundary of what you think the storage system is than outside."

What We Believe Is Being Asked

Two coupled requirements:

1. Codec correctness and codec evolution must not be a per-client burden. An ecosystem in which every client must ship and update every supported codec does not interoperate at scale: an organisation cannot upgrade its storage system’s encoding without coordinating an upgrade across every client.

2. The expensive recovery paths (partial writes, durable shard placement, mirrored logging) must not live at the client either. A protocol that exposes those paths to the client forces every client implementation to carry the failure-recovery machinery, which is precisely what RAID controllers and distributed storage systems put behind a service boundary so that hosts do not have to reason about it.

In short: the data-protection algorithm and its recovery story belong inside a storage boundary, not at the client.

How the Proxy Server Addresses This

The Proxy Server (PS) role, defined in [I-D.haynes-nfsv4-flexfiles-v2-proxy-server], is the storage boundary that Christoph and David asked for.

A PS is a peer of the MDS and the data servers that:

* speaks the codec on behalf of clients that cannot;

* receives whole-stripe operations from a codec-ignorant client;

* encodes (or decodes) using whatever the layout’s Figure 9 demands;

* drives the CHUNK operations to the participating data servers;

* carries the partial-write / FINALIZE / COMMIT recovery machinery that the codec requires.

Three properties follow:

* A legacy NFSv4.2 (or even NFSv3) client gets erasure-coded durability without speaking erasure coding. The PS is where the codec lives; the client does not have to be upgraded when the codec is upgraded.

* Codec evolution is a server-side concern. Adding a new entry to Figure 9 requires updating the PSes and DSes, not every client in the deployment. This matches the operational pattern of every other distributed-storage protocol on the wire.

* The recovery machinery (PENDING -> FINALIZED -> COMMITTED, the chunk-state machine, partial-write detection via Section 24.1) executes on the PS, not the client. Clients see ordinary NFSv4.2 semantics; the PS is responsible for converting those semantics into the chunk state-machine the DSes implement.

A codec-aware NFSv4.2 client is still permitted (and is the fast path: no proxy hop, no double bandwidth on the proxy's link). The PS is the answer for clients that either cannot speak the codec or are too old to be upgraded. In Christoph's framing, the PS is the inside of the storage boundary; codec-aware clients are implementations that have been admitted into that boundary by design.

The PS does carry a data-plane cost: client bytes traverse the proxy on the way to the DSes, so the proxy's link sees roughly twice the bandwidth of a direct client-to-DS path, and the PS pays the encode/decode CPU. This is the price of admission for clients that do not speak the codec; it is the same store-and-forward cost any storage gateway pays. It does not affect codec-aware clients, which talk to the DSes directly.

Working Group Concern: Coherent Multi-DS Writes Without Recall Storms

Source

Christoph Hellwig, IETF 122, NFSv4 Working Group session, during the FFv2 erasure-coding discussion.

The Question as Asked

Christoph observed that performing erasure coding across a set of data servers, where clients need a coherent view of the encoded data while writes are in flight, is "just really complicated, especially without recalling layouts." He continued: "maybe we need a more efficient network operation that doesn't recall layout but updates layouts in a different way, and that might reduce the overhead. Basically any scheme would require either a fair amount of intelligence on the data servers or some form of updating outstanding

layouts to point to a new right-out-of-place location." He explicitly noted he was "leaning to updating the data servers to be smarter."

The same conversation introduced the idea of a "generation counter that gets sent over the wire to the data servers, which means the data server now needs to look for a new location for the same existing layout."

What We Believe Is Being Asked

Two coupled requirements:

1. The MDS must be able to mutate where data lives -- replace a failing data server, redirect to a spare, rebalance, repair -- without serialising every layout-holding client through a CB_LAYOUTRECALL round-trip. A recall is global with respect to the layout: every client holding it must drain in-flight I/O and DELEGRETURN before the MDS can mutate. In an erasure-coded workload with many concurrent clients, this turns a localised DS hiccup into a global stall.

2. The data servers must be smart enough to enforce per-client access on a finer grain than "the file is reachable from the network." Anonymous-stateid I/O combined with synthetic-uid fencing is a coarse instrument: fencing one client's access to a file affects every client's access to that file. The only way to selectively revoke is to teach the DS who is permitted, on which file, with which iomode -- which is the "smarter data server" Christoph was asking for.

How TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID Address This

Sections Section 25.12, Section 25.13, and Section 25.14 of this document define exactly the "smarter data server" the working group asked for.

The mechanism:

* At LAYOUTGET, the MDS issues a real layout stateid and fans out TRUST_STATEID to each DS in the mirror set, registering (stateid.other, fh, clientid, iomode, expire) in a per-DS trust table. CHUNK_WRITE and CHUNK_READ on the DS now validate against the trust table; an unknown, expired, or revoked stateid yields NFS4ERR_BAD_STATEID.

* When the MDS needs to mutate the layout for a particular client -- because that client misbehaved, because a DS the layout points at is being drained, because the file is being repaired -- it issues REVOKE_STATEID to the affected DS. Other clients' trust entries on the same file are untouched.

* When the MDS needs to mutate at client-scope (lease expiry, client eviction), it issues BULK_REVOKE_STATEID, which removes every trust entry the named client has on the DS without affecting other clients.

The control-plane cost reshapes accordingly:

* Layout mutation is no longer global. The MDS reroutes data to a spare DS, rebuilds shards from surviving copies, and revokes only the trust entries that pointed at the failing location. The other clients holding the layout are not contacted.

* The revoked client only learns of the mutation lazily, on its next CHUNK_WRITE or CHUNK_READ to the affected stripe. That operation returns NFS4ERR_BAD_STATEID; the client responds with LAYOUTERROR; the MDS replies with a refreshed layout pointing at the new location; the client re-trusts and resumes. A client that never touches the affected stripe never pays the cost at all.

* With warm spares known to the MDS, the entire repair can complete before any client notices. The MDS reconstructs onto a spare using server-to-server traffic, atomically swaps the layout slot in its in-memory state, and revokes only the trust entries on the now-evacuated DS. Reading clients see no interruption (any k of the surviving shards reconstructs); writing clients pay one round-trip to refresh the layout when they next write the affected stripe.

The combination of TRUST_STATEID and a warm-spare DS pool is the "more efficient network operation that updates layouts" Christoph asked for. It is not literally a layout update on the wire; it is a primitive that makes layout updates a local event the MDS can resolve before the client has to pay a recall round-trip.

The chunk state machine (PENDING -> FINALIZED -> COMMITTED) and Section 24.1 address the orthogonal concern of partial-write recovery, ensuring that even when the MDS reroutes mid-write the DSes can detect inconsistent stripes via per-chunk generation checks rather than via a global wall-clock or consensus protocol.

Combined Effect on the "Cluster Tax"

The Proxy Server addresses the codec-distribution cost; the trust stateid mechanism addresses the layout-mutation cost. Together, they confine the residual cluster overhead to:

* the store-and-forward bandwidth on the PS link, paid only by clients that route through a PS rather than going DS-direct; and

* one LAYOUTERROR/LAYOUTGET round-trip per client per affected stripe, paid only by clients that actually try to use a stripe whose backing has changed.

Neither cost scales with the number of layout-holding clients, which is the property the working group asked for.

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

David Black clarified at IETF 124 that the consistency goal of Flex Files v2 is RAID consistency across the chunks of a stripe rather than POSIX write ordering across application writes; that framing is reflected in Section 3 and in the Non-Goals of Section 12.5.

Chris Inacio, Brian Pawlowski, and Gorry 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

[I-D.haynes-nfsv4-flexfiles-v2-proxy-server] Haynes, T., "Proxy-Driven Server for Flexible Files Version 2", Work in Progress, Internet-Draft, draft-haynes-nfsv4-flexfiles-v2-proxy-server-00, 28 April 2026, <https://datatracker.ietf.org/doc/html/draft-haynes-nfsv4-flexfiles-v2-proxy-server-00>.

[IANA-PEN] IANA, "Private Enterprise Numbers", <https://www.iana.org/assignments/enterprise-numbers/>.

[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, DGCI 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>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/rfc/rfc7942>.

Author’s Address

Thomas Haynes
Hammerspace
Email: loghyr@gmail.com