Network File System Version 4
Internet-Draft
Intended status: Standards Track
Expires: 26 November 2026

T. Haynes
Hammerspace

25 May 2026

Parallel NFS (pNFS) Flexible File Layout Version 2
draft-haynes-nfsv4-flexfiles-v2-06

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 Version 2 Layout Type is defined in this document as an extension to pNFS that allows the use of storage devices that require only a limited degree of interaction with the metadata server and use already-existing protocols. Data protection is also added to provide integrity. Both Client-side mirroring and the 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 8
2. Requirements Language 10
3. Motivation 10
4. Use Cases 13
5. Definitions 14
6. Coupling of Storage Devices 19
6.1. LAYOUTCOMMIT 20
6.2. Fencing Clients from the Storage Device 20
6.2.1. Implementation Notes for Synthetic uids/gids 21
6.2.2. Example of using Synthetic uids/gids 22
6.3. State and Locking Models 23
6.3.1. Loosely Coupled Locking Model 24
6.3.2. Tightly Coupled Locking Model 26
6.4. Tight Coupling Control Protocol 28
6.4.1. Capability Discovery 29
6.4.2. Control Session 30
6.4.3. Flow at LAYOUTGET 31
6.4.4. Principal Binding and the Kerberos Gap 32
6.4.5. Client-Detected Trust Gap 34
6.4.6. Lease and Renewal 34
6.4.7. Storage Device Crash Recovery 35
6.4.8. Metadata Server Crash Recovery 35
6.4.9. Backward Compatibility 37
7. Device Addressing and Discovery 37
7.1. ff_device_addr4 37
7.2. Storage Device Multipathing 40
8. Flexible File Version 2 Layout Type 41
8.1. ffv2_coding_type4 42
8.1.1. Heterogeneous Mirror Sets 42
8.1.2. FFV2_ENCODING_PASSTHROUGH 43
8.1.3. FFV2_ENCODING_MIRRORED 44
8.1.4. Encoding Type Interoperability 46

8.2. ffv2_layout4 46
8.2.1. ffv2_flags4 46
8.3. ffv2_file_info4 48
8.4. ffv2_ds_flags4 48
8.5. ffv2_data_server4 49
8.6. ffv2_coding_type_data4 49
8.7. ffv2_stripes4 50
8.8. ffv2_mirror4 51
8.9. ffv2_layout4 53
8.10. ffv2_data_protection4 55
8.11. ffv2_layouthint4 55
8.11.1. Codec Negotiation 57
8.11.2. Error Codes from LAYOUTGET 61
8.11.3. Client Interactions with FF_FLAGS_NO_IO_THRU_MDS . . 61
8.12. LAYOUTCOMMIT 61
8.13. Interactions between Devices and Layouts 62
8.14. Handling Version Errors 62
9. Striping 63
10. Recovering from Client I/O Errors 64
11. Client-Side Protection Modes 65
11.1. Client-Side Mirroring 65
11.1.1. Selecting a Mirror 66
11.1.2. Writing to Mirrors 66
11.1.3. Metadata Server Resilvering of the File 68
11.2. Erasure Coding 68
11.2.1. Encoding a Data Block 70
11.2.2. Decoding a Data Block 73
11.2.3. Write Modes 75
11.2.4. Selecting the Repair Client 76
11.2.5. Repair Protocol: Normative vs. Informative 78
11.2.6. Carrying Out the Repair 80
11.2.7. Reading Chunks 85
11.2.8. Whole File Repair 85
11.3. Mixing of Coding Types 86
11.4. Reed-Solomon Vandermonde Encoding (FFV2_ENCODING_RS_VANDERMONDE) 89
11.4.1. Overview 89
11.4.2. Galois Field Arithmetic 89
11.4.3. Encoding Matrix 89
11.4.4. Encoding 90
11.4.5. Decoding 90
11.4.6. RS Interoperability Requirements 91
11.4.7. RS Shard Sizes 91
11.5. Mojette Transform Encoding (FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC) 91
11.5.1. Overview 92
11.5.2. Grid Structure 92

11.5.3. Directions 92
11.5.4. Forward Transform (Encoding) 92
11.5.5. Katz Reconstruction Criterion 93
11.5.6. Inverse Transform (Decoding) 93
11.5.7. Systematic Mojette 94
11.5.8. Non-Systematic Mojette 94
11.5.9. Mojette Shard Sizes 94
11.6. Comparison of Encoding Types 95
11.7. First-Line Substitution to a Spare 96
11.8. Handling write holes 97
12. System Model and Correctness 97
12.1. Wire Semantics vs Implementation 98
12.2. Chunks Are Not Blocks 99
12.3. Actors and Roles 100
12.4. Failure Model 101
12.5. Chunk State Machine 102
12.6. Consistency Guarantees 105
12.7. Ownership and Scope of Retained Prior Content 108
12.8. Progress and Termination 109
12.9. Relation to Classical Consensus 111
12.10. Non-Goals 112
13. NFSv4.2 Operations Allowed to Data Files 112
13.1. Control Plane: Metadata Server to Data Server 113
13.2. Data Path: Client to Data Server 114
13.2.1. Session and Identity Plumbing 114
13.2.2. Stateid Model on the Data Server 114
13.2.3. GETATTR on a Data File 115
13.2.4. SETATTR on a Data File 116
13.2.5. MDS-Driven Truncate on Erasure-Coded Files 116
13.2.6. PASSTHROUGH Data Files (FFV2_ENCODING_PASSTHROUGH) 117
13.2.7. Chunked Data Files (FFV2_ENCODING_MIRRORED, FFV2_ENCODING_MOJETTE_*, FFV2_ENCODING_RS_VANDERMONDE) 118
13.2.8. Operations That MUST NOT Be Sent to a Data File . . 119
13.3. Callback Path: Data Server to Client 120
13.4. Summary Table 120
14. Flexible File Version 2 Layout Type Return 122
14.1. I/O Error Reporting 123
14.1.1. ffv2_ioerr4 124
14.2. Layout Usage Statistics 124
14.2.1. ff_io_latency4 124
14.2.2. ff_layoutupdate4 125
14.2.3. ff_iostats4 126
14.3. ffv2_layoutreturn4 127
15. Flexible File Version 2 Layout Type LAYOUTERROR 128
16. Flexible File Version 2 Layout Type LAYOUTSTATS 128
17. Flexible File Version 2 Layout Type Creation Hint 128

18. ff_layouthint4 128
19. Recalling a Layout 129
19.1. CB_RECALL_ANY 129
20. Layout Revocation and Fencing 130
21. New NFSv4.2 Error Values 131
21.1. Error Definitions 132
21.1.1. NFS4ERR_CODING_NOT_SUPPORTED (Error Code 10097) . . 132
21.1.2. NFS4ERR_PAYLOAD_NOT_ATOMIC (Error Code 10098) . . . 132
21.1.3. NFS4ERR_CHUNK_LOCKED (Error Code 10099) 132
21.1.4. NFS4ERR_CHUNK_GUARDED (Error Code 10100) 133
21.1.5. NFS4ERR_PAYLOAD_LOST (Error Code 10101) 133
21.1.6. NFS4ERR_LAYOUT_CHECKSUM_NOT_SUPPORTED (Error Code 10102) 133
21.2. Operations and Their Valid Errors 134
21.3. Callback Operations and Their Valid Errors 136
21.4. Errors and the Operations That Use Them 136
22. EXCHGID4_FLAG_USE_ERASURE_DS 136
23. New NFSv4.2 Attributes 137
23.1. Attribute 89: fattr4_coding_block_size 137
24. New NFSv4.2 Common Data Structures 137
24.1. chunk_guard4 137
24.1.1. Metadata-Server Assignment Rules for cg_client_id . 140
24.1.2. Data-Server Collision Handling 140
24.1.3. Reserved cg_client_id Value: CHUNK_GUARD_CLIENT_ID_NONE 141
24.1.4. Reserved cg_client_id Value: CHUNK_GUARD_CLIENT_ID_MDS 141
24.2. chunk_owner4 141
24.3. checksum4 142
25. New NFSv4.2 Operations 143
25.1. Operation 78: CHUNK_COMMIT - Activate Cached Chunk Data 147
25.1.1. ARGUMENTS 147
25.1.2. RESULTS 147
25.1.3. DESCRIPTION 148
25.1.4. RESPONSE CODES 152
25.2. Operation 79: CHUNK_ERROR - Report Error on Cached Chunk Data 152
25.2.1. ARGUMENTS 153
25.2.2. RESULTS 153
25.2.3. DESCRIPTION 153
25.2.4. RESPONSE CODES 154
25.3. Operation 80: CHUNK_FINALIZE - Transition Chunks from Pending to Finalized 155
25.3.1. ARGUMENTS 155
25.3.2. RESULTS 155
25.3.3. DESCRIPTION 155
25.3.4. RESPONSE CODES 158

25.4. Operation 81: CHUNK_HEADER_READ - Read Chunk Header from File 158
25.4.1. ARGUMENTS 158
25.4.2. RESULTS 158
25.4.3. DESCRIPTION 159
25.4.4. RESPONSE CODES 161
25.5. Operation 82: CHUNK_LOCK - Lock Cached Chunk Data . . . 162
25.5.1. ARGUMENTS 162
25.5.2. RESULTS 162
25.5.3. DESCRIPTION 163
25.5.4. RESPONSE CODES 165
25.6. Operation 83: CHUNK_READ - Read Chunks from File 166
25.6.1. ARGUMENTS 166
25.6.2. RESULTS 166
25.6.3. DESCRIPTION 167
25.6.4. RESPONSE CODES 170
25.7. Operation 84: CHUNK_REPAIRED - Confirm Repair of Errored Chunk Data 171
25.7.1. ARGUMENTS 171
25.7.2. RESULTS 171
25.7.3. DESCRIPTION 172
25.7.4. RESPONSE CODES 173
25.8. Operation 85: CHUNK_ROLLBACK - Rollback Changes on Cached Chunk Data 173
25.8.1. ARGUMENTS 173
25.8.2. RESULTS 174
25.8.3. DESCRIPTION 174
25.8.4. RESPONSE CODES 176
25.9. Operation 86: CHUNK_UNLOCK - Unlock Cached Chunk Data . 177
25.9.1. ARGUMENTS 177
25.9.2. RESULTS 177
25.9.3. DESCRIPTION 177
25.9.4. RESPONSE CODES 178
25.10. Operation 87: CHUNK_WRITE - Write Chunks to File 179
25.10.1. ARGUMENTS 179
25.10.2. RESULTS 179
25.10.3. DESCRIPTION 180
25.10.4. RESPONSE CODES 185
25.11. Operation 88: CHUNK_WRITE_REPAIR - Write Repaired Cached Chunk Data 186
25.11.1. ARGUMENTS 186
25.11.2. RESULTS 186
25.11.3. DESCRIPTION 187
25.11.4. RESPONSE CODES 189
25.12. Operation 89: TRUST_STATEID - Register Layout Stateid on Data Server 190
25.12.1. ARGUMENTS 190
25.12.2. RESULTS 190

25.12.3. DESCRIPTION 190
25.12.4. RESPONSE CODES 192
25.13. Operation 90: REVOKE_STATEID - Revoke Registered Stateid on Data Server 192
25.13.1. ARGUMENTS 192
25.13.2. RESULTS 193
25.13.3. DESCRIPTION 193
25.13.4. RESPONSE CODES 195
25.14. Operation 91: BULK_REVOKE_STATEID - Revoke All Stateids for a Client 195
25.14.1. ARGUMENTS 195
25.14.2. RESULTS 195
25.14.3. DESCRIPTION 196
25.14.4. RESPONSE CODES 197
26. New NFSv4.2 Callback Operations 197
26.1. Callback Operation 16: CB_CHUNK_REPAIR - Request Repair of Inconsistent Chunk Ranges 198
26.1.1. ARGUMENTS 198
26.1.2. RESULTS 199
26.1.3. DESCRIPTION 199
26.1.4. RESPONSE CODES 201
27. Security Considerations 201
27.1. Checksum Integrity Scope 203
27.2. Chunk Lock and Lease Expiry 204
27.3. Error Code Information Disclosure 204
27.4. Transport Layer Security 205
27.5. RPCSEC_GSS and Security Services 205
27.5.1. Loosely Coupled 205
27.5.2. Tightly Coupled 206
27.6. Trusted Stateids 206
27.6.1. Interaction with Kerberos and RPCSEC_GSS 206
27.6.2. Attack Surfaces and Mitigations 207
28. IANA Considerations 209
28.1. Checksum Algorithm Registry 211
29. XDR Description of the Flexible File Version 2 Layout Type . 213
30. References 213
30.1. Normative References 213
30.2. Informative References 215
Implementation Status 216
reffs (metadata server and data server) and ec_demo (Client) . 216
Interoperability and Benchmarks 217
Architectural Implication: Cost of Fault Tolerance 219
Design Rationale: Rejected Alternatives 220
Proprietary Projection Header Inside Opaque Payload 220
Per-Client Swap Files with metadata server MAPPING_RECALL . . . 221
Server-Side Byte-Range Lock Manager per File 221
Modified Two-Touch Paxos on Each Chunk 222
Automatic Commit of Empty Chunks 222

Global Clock or Wall-Clock-Based Generation Counter 223
Layout-Level Generation Counter 223
Declustered RAID with Dynamic Parity Mapping 224
Working Group Concern: Codec on Every Client 225
Source 225
The Question as Asked 225
What We Believe Is Being Asked 225
How the Proxy Server Addresses This 226
Working Group Concern: Coherent Multi-data server Writes Without Recall Storms 227
Source 227
The Question as Asked 227
What We Believe Is Being Asked 228
How TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID Address This 228
Combined Effect on the "Cluster Tax" 230
Acknowledgments 230
Author’s Address 231

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].

A metadata server that supports the Flexible File Version 2 Layout Type MUST be an NFSv4.2 server. The new operations defined by this document for the metadata server (the TRUST_STATEID family on the metadata server / storage device control session, and the CB_CHUNK_REPAIR back-channel callback to clients) are NFSv4.2 operations and have no representation in NFSv4.1 or earlier minor versions. Storage devices can speak NFSv3, NFSv4.1, or NFSv4.2, but some encoding types and coupling configurations narrow that choice; see Section 7.1 for the exact rules.

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 Version 1 Layout Type to version 2 by providing error-detection integrity (checksum) 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 Version 1 Layout Type (LAYOUT4_FLEX_FILES, [RFC8435]). The two layout types are NOT backward compatible: a flexible file v2 layout cannot be parsed as a flexible file v1 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

Workloads that need both the throughput of parallel pNFS data servers and the durability of erasure coding have driven the work in this draft. The deployments span scientific-instrumentation pipelines that produce petabytes of detector data per year, training-checkpoint files written hot from machine-learning jobs, archive workloads in which a single file may hold the only copy of an experiment’s result, and ordinary production filesystems where read patterns evolve across a file’s lifetime. These deployments are documented in detail in Section 4; the protocol shape that follows is the result of looking at them and asking what a pNFS layout type would have to provide.

The first thing the deployments share is that erasure coding moves work off the data servers that are already the bottleneck in the write-heavy parallel case. Server-side erasure coding makes each data server compute its share of the parity transform on every write, multiplying the per-write CPU cost by (k + m) across the storage tier and serialising on the data server’s limited compute. Client-side erasure coding shifts that compute to the writers, which scale horizontally with the workload, and lets the data servers stay close to their strength -- storing and serving bytes. Flexible file v1 layout ([RFC8435]) already chose client-side compute by placing replication at the writer; this draft extends that choice to client-side erasure coding. Benchmark measurements summarised in Appendix "Implementation Status" confirm that the resulting overhead is competitive with server-side encoding on realistic workloads and that the encoding compute scales with the writer population rather than with the data-server count.

Client-side erasure coding has a corollary that protocol designers cannot avoid: when a client fans a stripe out across multiple data servers and fails mid-write, no single data server has whole-transaction visibility. The state left behind on each data server is a partial fragment of a write that may or may not have completed elsewhere. A server-side coordinator that holds the whole stripe -- the flexible file v1 case -- can resilver from a surviving copy without any client involvement. In the v2 case there is no such coordinator, and the on-wire protocol must specify how the partial state is reconciled. This is the load-bearing constraint that shapes the rest of the design.

A natural-looking answer is to add a distributed-consensus protocol between the data servers and have them agree on which write committed. That answer is rejected here. Distributed consensus is operationally expensive, introduces a synchronisation cost on every write, and makes the data servers themselves stateful peers in a way that closes off the simpler implementations the protocol should accommodate. Instead, this draft uses two narrowly-scoped primitives that together provide just enough on-wire reconciliation: the chunk_guard4 compare-and-swap (CAS) and the CB_CHUNK_REPAIR callback.

Every CHUNK_WRITE carries a chunk_guard4 -- a 32-bit per-chunk generation counter and a 32-bit owning-client short-id -- and the data server performs a per-chunk CAS on receipt. If two writers race for the same chunk, exactly one wins on each data server, the loser receives NFS4ERR_CHUNK_GUARDED for that chunk, and the chunks the loser intended to write are left unchanged. No data server needs to consult its peers; the CAS is local. The cost on the metadata server is bounded by the 8-byte chunk_guard4 header per chunk plus a 32-bit per-layout client identifier (Section 24.1, Section 8.8). Independent collisions on different chunks resolve independently; there is no file-wide lock and no global ordering across writes.

When the per-chunk CAS detects that a stripe ended up non-atomic -- some shards under writer A’s guard, others under writer B’s, or a writer crashed mid-fan-out -- the metadata server selects a repair client via CB_CHUNK_REPAIR (Section 26.1) and that client drives the repair: it acquires CHUNK_LOCK on the affected range, reads the surviving shards, decodes through the erasure transform, writes the reconstructed shards via CHUNK_WRITE_REPAIR, and clears the errored state via CHUNK_REPAIRED. The repair client is exactly one actor holding a chunk-range lock; the data servers still do not coordinate among themselves. The repair-client selection rule is given in Section 11.2.4.

These two primitives -- the per-chunk CAS and the callback-driven repair -- replace what would otherwise require a distributed-consensus protocol. The CAS handles the common case at local cost, where independent writers racing for different chunks resolve independently and concurrent writers on the same chunk get a clean win/loss decision. CB_CHUNK_REPAIR handles the rare case of partial-failure non-atomicity with a single coordinator selected per repair episode. The cost model is asymmetric on purpose: the hot path pays for an 8-byte header and a local CAS; the cold path pays for a selected actor and a small number of round-trips.

The CHUNK_* operation set in this draft is the minimum sufficient to drive the chunk state machine (Section 12.5) plus the repair flow above. CHUNK_WRITE places PENDING content; CHUNK_FINALIZE signals that the writer is done with a generation; CHUNK_COMMIT promotes that generation to durable, globally visible state; CHUNK_READ retrieves it; CHUNK_HEADER_READ provides the fast probe that lets repair coordinators and recovering writers inspect chunk metadata without reading payloads. CHUNK_LOCK, CHUNK_UNLOCK, CHUNK_ERROR, CHUNK_REPAIRED, CHUNK_WRITE_REPAIR, and CHUNK_ROLLBACK together drive the repair flow. Each operation does one well-scoped job; the complexity is in the state machine the operations drive, not in the operation set itself. Each of these primitives closes a specific gap in the lifecycle or the repair path. The detailed treatment of the operation set is in Section 25.

The same design discipline shapes the rest of the specification. The protocol describes wire format and server obligations; it does not pin a data-server backend, a control protocol between metadata server and data server, a checksum algorithm, or a file-attribute representation on the data server. Different implementations resolve these choices differently and remain conformant. TRUST_STATEID (Section 6.4) is one such control protocol that this draft defines; storage devices with their own established control protocols are conformant without implementing it. The tagged checksum4 (Section 24.3) lets the metadata server pick any registered checksum algorithm per file. The authorization-outcome parity rule (Section 6.3) lets data servers that do not expose a POSIX file namespace satisfy the tight-coupling requirements without materialising POSIX uid/gid bits.

A protocol-level consequence of placing erasure coding at the client is that the layout must be able to describe a file’s storage shape over its full lifetime -- including the transition windows when the file is being assimilated from a non-erasure-coded source, re-encoded from one codec to another, or recovered from a correlated codec failure through a mirror under a different encoding. This draft allows a single file’s layout to contain mirrors under different

encodings. The heterogeneous-mirror capability is not a steady-state expectation; most files have one encoding most of the time. It is the protocol shape that lets transitions happen while the file remains readable. The deployment cases that drive the allowance are catalogued in Section 4.

Scope note: the consistency goal of flexible file v2 layout is RAID consistency across the shards 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 shards comprising a given stripe agree on which write produced them -- expressed on the wire as agreement on the chunk_guard4 value of every chunk that carries those shards -- 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 flexible file v2 layout 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 12.8.

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, disjoint regions: 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

block: the application's view of file data. A block is a unit of file content as observed by an NFS client at the POSIX layer or by the local file system. A chunk's payload, after any decoding the client performs, is presented to the application as one or more blocks.

shard: the codec's view. A shard is a single piece of an encoded stripe produced by an erasure-coding (or replication) transformation. A stripe of k data shards plus m parity shards is the unit a codec encodes and decodes. The word "shard" only has meaning while the codec is reasoning about a stripe; once a shard is at rest on a data server it is, by virtue of having been transmitted, the payload of a chunk.

chunk: the protocol's unit of file data on the wire, carrying an envelope that distinguishes it from a block: a compare-and-swap guard (chunk_guard4 -- atomicity, see Section 24.1), a checksum (per-chunk integrity), a provenance identifier (chunk_owner4, see Section 24.2), a lifecycle state (PENDING / FINALIZED / COMMITTED via the chunk state machine, see Section 12.5), and per-chunk locking that survives stateid revocation through lock escrow. A chunk is the addressable unit named in the CHUNK_* operations defined in this document and durably persisted by a data server. A chunk's payload may be a block (mirrored layout) or a shard (erasure-coded layout); the wire protocol does not distinguish. The chunk size MAY differ from the size of the block or shard it carries. See Section 12.2 for the load-bearing role each envelope property plays in the protocol's consistency story.

The three terms describe the same data at three different layers and should be used accordingly. The codec transforms blocks into shards; the wire protocol transmits shards as chunk payloads; the data server persists chunks. On read the path reverses.

A protocol-internal note: the chunk state machine (Section 12.5) and several CHUNK_* operations refer to the per-chunk-offset state records as "blocks" (PENDING / FINALIZED / COMMITTED / errored). This is a finer-grained use of the word, internal to the data

server’s chunk metadata, and should not be confused with the application-layer "block" defined above. Where ambiguity matters, this document writes "chunk-state block" or relies on context (operation names, state names) to disambiguate.

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 (as defined above) 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 stripe of data is encoded into shards (k data shards and m parity shards) so that the original content can be reconstructed from any sufficient subset of the shards. Shards are transmitted as the payload of CHUNK operations and stored on different data servers.

client-side erasure coding: A file based integrity method where copies are maintained in parallel.

compare-and-swap (CAS): an atomic primitive from concurrent programming in which an update is conditional on a prior observed value: the operation succeeds only if the current value matches an expected prior value, and otherwise fails so the caller can retry. In this document, the chunk_guard4 mechanism (see Section 24.1) implements CAS at the chunk level; the "expected prior value" is the chunk_guard4 the writer observed at read time, and the "fail" outcome is NFS4ERR_CHUNK_GUARDED.

(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.

escrow (lock escrow, MDS-escrow): a state in which a chunk lock is held by the metadata server on behalf of an as-yet-unselected future owner. When the metadata server revokes a client's stateid while the client still holds chunk locks, the locks are not dropped (which would expose the chunks to concurrent writers) but are transferred to the metadata server itself, marked by the reserved cg_client_id value CHUNK_GUARD_CLIENT_ID_MDS (see Section 24.1.4). The metadata server holds the locks in escrow until a repair client adopts them via CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT (driven by CB_CHUNK_REPAIR). An "MDS-escrow owner" is the metadata server acting in this placeholder role; "in escrow" describes a lock in this state. Escrow preserves the lock-continuity invariant across stateid revocation: at no point during the revocation sequence is a chunk simultaneously locked and unowned.

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.

proxy server (PS): a peer of the metadata server, defined in [I-D.haynes-nfsv4-flexfiles-v2-proxy-server], that admits client I/O on the metadata server’s behalf -- either as a translator for clients that cannot speak the file’s native codec, or as a proxy-mediated data path during whole-file move and repair operations. A proxy server may additionally act as a data server.

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. The specific tight-coupling variant defined by this document, in which the control protocol is the TRUST_STATEID family, is referred to as trusted-stateid tight coupling (see Section 6.4).

trusted-stateid tight coupling: the specific tight-coupling control

protocol defined in this document, consisting of the operations TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID. Within the scope of this document, unqualified references to "tight coupling" or "tightly coupled" refer to trusted-stateid tight coupling unless the context explicitly discusses the general concept. Other tight-coupling control protocols (proprietary or future Standards Track) may exist but are not covered by this specification.

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 v2 layout addresses these issues. To implement the tightly coupled model, a control protocol has to be defined. As the flexible file v2 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 and the storage device agree on the authorization decision for each client access: a client allowed by the metadata server to read or write a file is allowed the same access at the storage device, and a client denied at the metadata server is denied at the storage device. How the storage device reaches that decision is not constrained by this specification. Some storage devices replicate the user, group, mode bits, and ACL of the metadata file onto a POSIX-shaped local representation of the data file and let their native filesystem enforce the decision; others (such as storage devices backed by an object store, a control-protocol-driven backend, or a backend with no exposed file namespace) consult the control protocol directly without ever materializing a POSIX file representation. Both approaches are conformant; the specification’s requirement is the authorization-outcome parity, not the mechanism that produces it.

The client authenticates with the storage device and receives the same authorization outcome it would have received 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. 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 receives 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

The coupling model in effect for a given metadata-server / storage-device pair is not negotiated over the NFS protocol. The metadata server determines the coupling model from out-of-band signals: administrative configuration, the choice and capabilities of the control protocol between the metadata server and the storage device, the storage device's data-path protocol version, and the storage device's backend architecture. At the NFS protocol level, the metadata server's expectations of the storage device follow these classifications:

* Storage devices implementing the NFSv3 or NFSv4.0 protocols on the data path are treated as loosely coupled.

* NFSv4.1+ storage devices that do not return the EXCHGID4_FLAG_USE_PNFS_DS flag in EXCHANGE_ID indicate 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 identify themselves with the EXCHGID4_FLAG_USE_PNFS_DS flag set in EXCHANGE_ID can potentially be tightly coupled. They use a back-end control protocol to implement the global stateid model described in [RFC8881].

Tight coupling additionally requires a control protocol between the metadata server and the storage device, discovered or advertised out-of-band as described above.

Some storage devices cannot operate under the loosely coupled model at all. The loose-coupling model in this specification relies on the storage device authorizing client access against synthetic uid and gid values (Section 6.2), which presupposes that the data file has a local representation on the storage device against which POSIX-style ownership checks can be applied. Storage devices whose backend has no exposed file namespace -- for example, object-store-backed data servers, or data servers driven entirely through a control protocol against a non-POSIX backend -- do not have that local representation and MUST operate in the tightly coupled model with a control protocol that conveys the authorization decision directly. A metadata server deploying with such a storage device cannot fall back to loose coupling.

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. This document defines one such control protocol -- the TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID operations in Section 6.4 -- for deployments in which the storage devices are NFSv4.2 servers willing to implement the new operations. A storage device with its own established back-end control protocol that provides the equivalent functional capabilities is conformant under this specification without implementing the TRUST_STATEID family; see Section 6.4 for the conformance framing.

When using the TRUST_STATEID control protocol defined in Section 6.4, the metadata server and a storage device establish that they can use it 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:

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.

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 one such control protocol, designated _trusted-stateid tight coupling_, 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.

Other tight-coupling control protocols may exist or be defined elsewhere. Existing pNFS server implementations with established back-end control protocols -- for example, dCache, which has its own control protocol between its metadata service and its data servers -- satisfy the tightly-coupled locking model (Section 6.3) through their own mechanisms and are conformant under this specification provided they meet the functional capabilities described there. Such implementations need not adopt the TRUST_STATEID family, and their interoperability with the TRUST_STATEID family is outside the scope of this document.

A storage device that does not implement TRUST_STATEID is treated as not supporting trusted-stateid tight coupling specifically; the capability probe in Section 6.4.1 detects this and the metadata server falls back to loose coupling (Section 6.4.9) or, if the storage device's own control protocol is in use, that protocol governs. Within the remainder of Section 6.4 and its subsections, unqualified references to "tight coupling" or "tightly coupled" refer to the trusted-stateid variant defined here.

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 (data server). The same mechanism applies to a proxy server [I-D.haynes-nfsv4-flexfiles-v2-proxy-server] -- a proxy server may or may not additionally act as a data server, 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 [I-D.haynes-nfsv4-flexfiles-v2-proxy-server]"; the flag check and the three operations are identical for both roles.

6.4.1. Capability Discovery

A storage device indicates support for trusted-stateid tight coupling implicitly, by processing TRUST_STATEID rather than returning NFS4ERR_NOTSUPP. (A storage device that supports a non-TRUST_STATEID form of tight coupling but not the trusted-stateid variant defined here will return NFS4ERR_NOTSUPP on this probe; from this specification's perspective it is treated the same as a storage device that does not support tight coupling at all.) 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: trusted-stateid 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: trusted-stateid 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 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

The flexible file v1 layout 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 flexible file v1 layouts carry only ffds_user / ffds_group (POSIX uid/gid for AUTH_SYS). A strict Kerberos deployment on the flexible file v1 layout must either allow AUTH_SYS from the metadata server's subnet or accept that the flexible file v1 layout's 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 flexible file v1 layout 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 -- 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 [I-D.haynes-nfsv4-flexfiles-v2-proxy-server] -- the storage device observes CHUNK operations arriving from the proxy server'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 [I-D.haynes-nfsv4-flexfiles-v2-proxy-server]'s credential-forwarding rules (in particular rule 1, "Credential pass-through") require the proxy server 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 proxy server, 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 proxy server's service identity.

* For an AUTH_SYS client whose I/O is proxied through a proxy server, the metadata server MUST set tsa_principal to the empty string (identical to the non-proxied case). The proxy server 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 proxy server's own service principal. Doing so would require the proxy server to authenticate to the storage device as itself (bypassing credential forwarding) which is explicitly prohibited by rule 4 of [I-D.haynes-nfsv4-flexfiles-v2-proxy-server] ("proxy server 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

A storage device MAY persist its trust table across restarts. An implementation that does so MUST also persist its server-instance identity, returning the same eir_server_owner.so_minor_id on EXCHANGE_ID after the restart (per [RFC8881] S18.35), so that clients and the metadata server observe the device as continuously available and the persisted trust entries remain valid against the layout stateids that were issued before the restart.

A storage device that does NOT persist its trust table empties the table on restart and MUST present a new server instance (incremented so_minor_id) so that clients detect the restart. The remainder of this section describes the recovery path for the volatile case.

The client detects a volatile 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

A metadata server MAY persist all its trust-management state across restarts. An implementation that does so MUST also persist its server-instance identity, returning the same eir_server_owner.so_minor_id on EXCHANGE_ID after the restart (per [RFC8881] S18.35), so that storage devices observe the metadata server as continuously available and accept incoming TRUST_STATEID and REVOKE_STATEID operations against the existing trust entries without revalidation. No grace period is required.

A metadata server that presents a new server instance (incremented so_minor_id) on restart follows the recovery path in the remainder of this section.

When the metadata server restarts as a new instance, 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 1 or 2, and the client MUST access the storage device using NFSv4 with the specified minor version.

Two additional constraints narrow the valid set of (ffdv_version, ffdv_minorversion) tuples in specific cases:

* When a mirror’s encoding type uses CHUNK_* operations (that is, any FFV2_ENCODING_* value other than FFV2_ENCODING_PASSTHROUGH), the corresponding storage device MUST be advertised with ffdv_version = 4 and ffdv_minorversion = 2. CHUNK_* operations are NFSv4.2 ops defined in this document; NFSv3 and NFSv4.1 storage devices cannot serve a non-PASSTHROUGH mirror.

* When ffdv_tightly_coupled is true (indicating trusted-stateid tight coupling), the storage device MUST be advertised with ffdv_version = 4 and ffdv_minorversion = 2. The TRUST_STATEID family of operations is defined as NFSv4.2; NFSv4.1 storage devices cannot participate in trusted-stateid tight coupling.

PASSTHROUGH mirrors with loose coupling are the only configuration for which (3, 0) or (4, 1) remain valid; for all other configurations the storage device MUST be NFSv4.2.

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. The flag was defined by [RFC8435] as a general tight-coupling indicator; in flexible file v2 layouts the flag specifically indicates trusted-stateid tight coupling (Section 6.4). 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, regardless of whether it implements some other (non-TRUST_STATEID) tight-coupling control protocol; from this specification’s perspective, only trusted-stateid tight coupling is interoperable.

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 v2 layout 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_ENCODING_PASSTHROUGH             = 1,
///     FFV2_ENCODING_MOJETTE_SYSTEMATIC      = 2,
///     FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC  = 3,
///     FFV2_ENCODING_RS_VANDERMONDE          = 4,
///     FFV2_ENCODING_MIRRORED                = 5
/// };
```

Figure 9: The coding type

The ffv2_coding_type4 (see Figure 9) encompasses a new IANA registry for 'Flexible File Version 2 Layout Type Erasure Coding Type Registry'. I.e., instead of defining a new Layout Type for each erasure coding, we define a new Erasure Coding Type. The encoding types this document defines fall into two groups:

* FFV2_ENCODING_PASSTHROUGH is the non-chunked, non-integrity on-ramp from flexible file v1 layout. It uses NFSv3 WRITE / READ directly against each replica's data server. No CHUNK_WRITE, no CHUNK_READ, no per-chunk CRC. See Section 8.1.2.

* FFV2_ENCODING_MIRRORED, FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC, and FFV2_ENCODING_RS_VANDERMONDE all use the new operations defined here: in particular CHUNK_WRITE (Section 25.10) and CHUNK_READ (Section 25.6), which carry the per-chunk checksum this version of the layout type relies on for end-to-end integrity. The encoding type selects how chunks are produced from application data (mirrored verbatim, Reed-Solomon shards, Mojette projections); the wire and the storage device are the same in every case.

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.

8.1.1. Heterogeneous Mirror Sets

A single flexible file v2 layout's ffv2l_mirrors array MAY carry mirror entries of different encoding types. The protocol does not require the entries to agree -- one mirror can be FFV2_ENCODING_PASSTHROUGH, another can be FFV2_ENCODING_RS_VANDERMONDE, both describing the same file's data range. This combination is the structural primitive for three

operations that motivate keeping PASSTHROUGH in the layout type's vocabulary:

Assimilate: A file that exists today as a plain copy on storage outside flexible file v2 layout -- no chunk envelope, no per-chunk CRC -- enters the namespace as a PASSTHROUGH mirror against the source bytes as they are. The metadata server then adds one or more encoded mirrors (MIRRORED, RS, Mojette) to the same layout and synchronizes them from the PASSTHROUGH source. Clients can read the file via the PASSTHROUGH mirror immediately; the encoded mirrors become available as they are populated. No "rewrite before serve" step is required.

Copy / migrate between encodings: Changing a file from one encoding to another is "add a mirror in the target encoding to the same layout, let it sync from any healthy source mirror, retire the source mirror." PASSTHROUGH is the special case where one endpoint of that migration is "no encoding."

Repair across encodings: When an encoded mirror has a chunk whose CRC fails and whose parity cannot reconstruct, a PASSTHROUGH mirror in the same layout is an authoritative source: CHUNK_READ a peer encoded mirror or read the PASSTHROUGH byte range, then CHUNK_WRITE the repaired chunk. The reverse is also true: a byte range on the PASSTHROUGH copy whose contents the metadata server suspects has drifted can be repaired by reconstructing from the verified-CRC chunks of an encoded peer. Two encodings of the same file are two independent recovery paths.

The metadata server is responsible for keeping the entries in a heterogeneous mirror set in sync; the protocol does not require client awareness of which encoding produced which mirror beyond what the layout already states.

8.1.2. FFV2_ENCODING_PASSTHROUGH

FFV2_ENCODING_PASSTHROUGH is the on-ramp from flexible file v1 layout ([RFC8435]) into the flexible file v2 layout type. A PASSTHROUGH mirror points at the file's bytes as they exist on the data server, without the chunk envelope, checksum header, or chunk_guard4 fields that the encoded types use. Client I/O against a PASSTHROUGH mirror uses NFSv3 WRITE / READ ([RFC1813]) or NFSv4 READ / WRITE ([RFC8881]) directly -- not CHUNK_WRITE / CHUNK_READ.

PASSTHROUGH provides:

* Replication of data across N data servers, exactly as flexible file v1 layout does. Clients write to every replica; clients read from any one. N-way redundancy tolerates up to N-1 replica losses.

* Zero codec compute at the client and zero chunk-metadata overhead at the server. The on-disk format is the file itself.

* Compatibility with files that already exist outside flexible file v2 layout. A PASSTHROUGH mirror can be created over an existing file without rewriting it.

PASSTHROUGH does NOT provide:

* Per-chunk integrity. There is no checksum on the data path. Silent corruption is undetectable without out-of-band tooling (e.g., comparing checksums across replicas).

* Chunk-grained repair. The repair unit is the whole file: resilvering picks a trusted replica and replicates it end to end to the affected replica(s).

* The concurrent-writer disambiguation that chunk_guard4 provides for encoded types.

PASSTHROUGH is RECOMMENDED for the assimilation, migration, and heterogeneous-mirror use cases described in Section 8.1.1. New deployments that do not need a flexible file v1 layout on-ramp SHOULD use FFV2_ENCODING_MIRRORED for the integrity guarantees described in Section 8.1.3.

8.1.3. FFV2_ENCODING_MIRRORED

FFV2_ENCODING_MIRRORED is the chunked-with-integrity peer of PASSTHROUGH. The chunk produced for each replica is the application data verbatim -- no transform, no parity shards -- but it travels on the wire and is stored on the data server through CHUNK_WRITE / CHUNK_READ and so carries every integrity property the encoded coding types carry.

What FFV2_ENCODING_MIRRORED keeps from the mirror model:

* Zero codec compute at the client. Each replica’s chunk is the input bytes; there is no transform to apply on write and nothing to decode on read.

* Storage cost of N x payload, where N is the replica count. Mirroring trades storage for redundancy without the reconstruction machinery that erasure coding requires.

* Reading any one intact replica is sufficient. If a replica fails to verify (see below), the client tries another.

What FFV2_ENCODING_MIRRORED adds beyond PASSTHROUGH, by virtue of using CHUNK_WRITE and CHUNK_READ:

* Per-chunk checksum on write and on read. The CRC is computed by the client over the chunk payload, sent on the wire with the chunk, recomputed by the data server before storing, and recomputed again from disk by the data server on every CHUNK_READ. Wire-level bit flips are caught before the chunk is stored; on-disk bit rot is caught the next time the chunk is read.

* Per-chunk repair granularity. When one replica's CRC fails to verify and another replica's verifies, the repair unit is the chunk, not the file: CHUNK_READ the good replica, CHUNK_WRITE to the bad replica, done. No whole-file resilvering is required.

* Per-chunk concurrent-writer disambiguation. Mirrored writes carry the same chunk_guard4 (Section 24.1) the erasure coding types do. Two clients racing to write the same offset of the same file fan out to every replica with a guard pair (generation, owning-client short-id) per chunk; the CHUNK_FINALIZE step resolves which writer's chunk wins and the other writer observes a deterministic loss instead of an unresolved split-mirror.

What FFV2_ENCODING_MIRRORED is for: files where the deployment wants integrity and replication without the storage savings or the reconstruction story of erasure coding. Small files that do not exceed a single stripe, files whose access pattern is read-mostly and where the N x storage cost is acceptable, and files where the operator prefers the simplicity of "any one replica is the file" over "k of (k+m) shards reconstruct the file." The coding choice is per-file; a deployment can mix mirrored and erasure-coded files in the same namespace and pick whichever fits each file's profile.

What FFV2_ENCODING_MIRRORED is not: a substitute for erasure coding when storage efficiency or multi-replica fault tolerance matters. An N-way mirror tolerates up to N-1 replica losses but costs N x the payload; a (k, m) erasure coding tolerates m losses at (k+m)/k x the payload. Both have per-chunk integrity under this document; the choice is the cost-vs-tolerance one.

8.1.4. 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                ffv2fi_stateid;
///     nfs_fh4                 ffv2fi_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 Version 2 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_ENCODING_PASSTHROUGH:
///         ffv2_data_protection4   fctd_protection;
///     case FFV2_ENCODING_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 19) 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 every arm of the union currently carries 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.

The (data, parity) tuple is interpreted per encoding type:

* FFV2_ENCODING_PASSTHROUGH preserves the flexible file v1 layout-style notation for backward compatibility: fdp_data is 1 and fdp_parity is the number of additional copies (e.g., fdp_parity=2 for 3-way mirroring). The "1" data carrier is the file as stored; the fdp_parity additional copies are the flexible file v1 layout mirror replicas.

* FFV2_ENCODING_MIRRORED uses the N+0 notation: fdp_data is the number of replicas (e.g., fdp_data=3 for 3-way mirroring) and fdp_parity MUST be 0. Every replica is a full, independent data carrier; mirroring carries no parity reconstruction.

* Erasure coding types (FFV2_ENCODING_RS_VANDERMONDE, FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC, and any future types subsequently registered in the IANA registry established by this document) use fdp_data >= 2 and fdp_parity >= 1.

8.7. ffv2_stripes4

```
/// enum ffv2_striping {
///     FFV2_STRIPING_NONE = 0,
///     FFV2_STRIPING_SPARSE = 1,
///     FFV2_STRIPING_DENSE = 2
/// };
///
/// struct ffv2_stripes4 {
///         ffv2_data_server4       ffv2s_data_servers<>;
/// };
```

Figure 15: The ffv2_stripes4 structure

Each stripe contains a set of data servers in ffv2s_data_servers. If the stripe is part of a ffv2_coding_type_data4 of FFV2_ENCODING_PASSTHROUGH or FFV2_ENCODING_MIRRORED, then the length of ffv2s_data_servers MUST be 1: under both encoding types each stripe's data lives on a single data server, with replica multiplicity expressed in ffv2l_mirrors rather than in ffv2s_data_servers.

8.8. ffv2_mirror4

```
/// struct ffv2_mirror4 {
///         ffv2_coding_type_data4  ffv2m_coding_type_data;
///         ffv2_striping           ffv2m_striping;
///         uint32_t                ffv2m_striping_unit_size;
///         uint32_t                ffv2m_client_id;
///         checksum_algorithm4     ffv2m_checksum_algorithm;
///         ffv2_stripes4           ffv2m_stripes<>;
/// };
```

Figure 16: The ffv2_mirror4

The ffv2_mirror4 (in Figure 16) describes the Flexible File Layout Version 2 specific fields.

The ffv2m_checksum_algorithm field names the checksum algorithm the client MUST use when computing cwa_checksums on CHUNK_WRITE and cwra_checksums on CHUNK_WRITE_REPAIR, and the algorithm the client MUST expect in cr_checksum on CHUNK_READ responses, for chunks in this mirror. The metadata server picks the algorithm at LAYOUTGET time; the value is one of the registered checksum_algorithm4 codes (see Section 24.3). Different mirrors of the same file MAY name different checksum algorithms, supporting transition cases where one mirror is being migrated to a stronger algorithm while others retain the previous algorithm.

A client that does not implement the algorithm named in ffv2m_checksum_algorithm MUST return the layout with NFS4ERR_LAYOUT_CHECKSUM_NOT_SUPPORTED; the metadata server may then issue a new layout naming a different algorithm the client supports, or deny the layout request.

The ffv2m_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 identifier; [RFC8881] does not constrain how a server populates its bits, and the bit-layout choices made by any particular metadata server implementation are not visible to the client and MUST NOT be assumed by the client. Folding clientid4 to 32 bits locally at the client therefore risks colliding with another client's folded value, which would 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 ffv2m_client_id that is guaranteed distinct from the ffv2m_client_ids assigned to other clients holding concurrent write layouts on the same file. The metadata server MUST assign ffv2m_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 ffv2m_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 ffv2m_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 ffv2m_client_id. ffv2m_client_id does NOT survive a metadata server restart: the metadata server reassigns values as clients reclaim layouts during the grace period.

The ffv2m_coding_type_data is which encoding type is used by the mirror.

The ffv2m_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 ffv2m_striping_unit_size is the stripe unit size used by the mirror. The minimum stripe unit size is 64 bytes. If the value of ffv2m_striping is FFV2_STRIPING_NONE, then the value of ffv2m_striping_unit_size MUST be 1.

The ffv2m_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 ffv2m_coding_type_data is FFV2_ENCODING_PASSTHROUGH, then the length of ffv2m_stripes MUST be 1. Under FFV2_ENCODING_MIRRORED the file MAY be striped within each replica; the constraint that ffv2s_data_servers length is 1 still applies, but ffv2m_stripes may carry multiple stripes.

8.9. ffv2_layout4

```
/// struct ffv2_layout4 {
///     ffv2_mirror4            ffv2l_mirrors<>;
///     ffv2_flags4             ffv2l_flags;
///     uint32_t                ffv2l_stats_collect_hint;
/// };
```

Figure 17: The ffv2_layout4

The ffv2_layout4 (in Figure 17) describes the Flexible File Layout Version 2.

The ffv2l_mirrors field is the array of mirrored storage devices that provide the storage for the current stripe; see Figure 18.

The ffv2l_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.

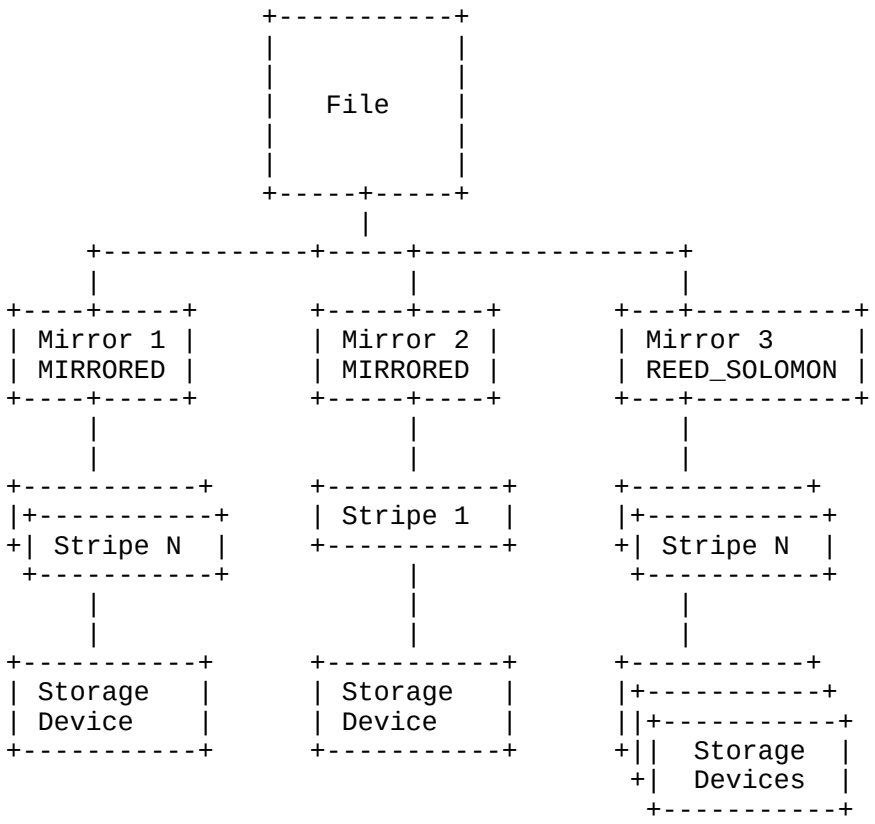


Figure 18: The Relationship between MDS and DSes

As shown in Figure 18 if the ffv2m_coding_type_data is FFV2_ENCODING_PASSTHROUGH or FFV2_ENCODING_MIRRORED, then each of the stripes MUST only have 1 storage device. I.e., the length of ffv2s_data_servers MUST be 1. The erasure-coding encoding types (FFV2_ENCODING_MOJETTE_SYSTEMATIC, FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC, FFV2_ENCODING_RS_VANDERMONDE) distribute shards across multiple storage devices and so carry multiple entries in ffv2s_data_servers.

The abstraction here is that for FFV2_ENCODING_PASSTHROUGH and FFV2_ENCODING_MIRRORED, each stripe describes exactly one data server. And for the erasure-coded encoding types, each of the stripes describes a set of data servers to which the shards 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 19: The ffv2_data_protection4

The ffv2_data_protection4 (in Figure 19) 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       ffv2lh_supported_types<>;
///     ffv2_data_protection4   ffv2lh_preferred_protection;
/// };
```

Figure 20: The ffv2_layouthint4

The ffv2_layouthint4 (in Figure 20) 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:

ffv2lh_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.

ffv2lh_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:

```
ffv2lh_supported_types = { FFV2_ENCODING_PASSTHROUGH,
                           FFV2_ENCODING_MIRRORED,
                           FFV2_ENCODING_MOJETTE_SYSTEMATIC,
                           FFV2_ENCODING_RS_VANDERMONDE }
ffv2lh_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_ENCODING_MIRRORED with (fdp_data=3, fdp_parity=0) for 3-way mirroring with per-chunk integrity, or FFV2_ENCODING_PASSTHROUGH with (fdp_data=1, fdp_parity=2) for 3-way flexible file v1 layout-compatible mirroring without per-chunk integrity.

8.11.1. Codec Negotiation

Because the coding-type registry is expected to grow over time (new erasure coding types 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 ffv2lh_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 ffv2lh_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) ffv2lh_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 metadata server translating to the underlying data server 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 metadata server 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_ENCODING_MIRRORED for a minimal NFSv4.2 client, or FFV2_ENCODING_PASSTHROUGH / a flat NFSv3 surface for an NFSv3 client). The proxy encodes and decodes on the fly against the real data servers. 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 [I-D.haynes-nfsv4-flexfiles-v2-proxy-server]; 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 ffv2lh_supported_types on its next LAYOUTGET.

Note: In Figure 17 ffv2_coding_type_data4 is an enumerated union with the payload of each arm being defined by the protection type. ffv2m_client_id tells the client which id to use when interacting with the data servers.

The ffv2_layout4 structure (see Figure 17) 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 17, 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 ffv2m_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 ffv2s_data_servers within each ffv2_stripes4. If the number of stripes is one, then ffv2m_striping_unit_size MUST be zero. The mapping scheme (sparse or dense) is selected per mirror by ffv2m_striping and is detailed in Section 9.

Stripe unit size and stripe count MAY differ between mirrors in the same layout segment. In particular, mirrors of different encoding types (see Section 8.1.1) have stripe counts determined by their respective (fdp_data, fdp_parity) protection structures, and there is no requirement that those structures match across mirrors. Each mirror is self-consistent internally; cross-mirror coherence is at the byte level (every mirror represents the same file bytes), not at the stripe-geometry level.

The ffv2l_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 (ffv2fi_fh_vers) with a stateid (ffv2fi_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, ffv2fi_stateid provides the stateid to be used by the client to access the file. The metadata server registers ffv2fi_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 ffv2fi_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 ffv2fi_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 ffv2fi_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 ffv2fi_fh_vers with its own ffv2fi_stateid inside ffv2_file_info4, the flexible file v2 layout addresses a limitation in the flexible file v1 layout where a single stateid was shared across all filehandles.

Whether the ffv2fi_stateid values across an ffv2_file_info4 array are distinct depends on each entry's coupling mode per the rules above. Loose-coupling and NFSv3 entries MUST carry the anonymous stateid; those entries are therefore byte-identical by mandate. Tight-coupling entries carry stateids the metadata server assigned and registered via TRUST_STATEID; the metadata server MAY assign these distinctly per filehandle version or MAY reuse the same stateid across entries.

The client MUST treat each (ffv2fi_fh_vers, ffv2fi_stateid) pair as an opaque, independent authorization unit. The client MUST NOT compare ffv2fi_stateid values across entries in the array and MUST NOT infer any relationship between two entries whose stateid values are byte-identical. When the client selects an entry to use for I/O, it presents that entry's stateid with that entry's filehandle; other entries in the array are unused for that I/O.

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 v2 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 v2 layout, 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 ffv2fi_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 ffv2_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 v2 layout 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 ffv2m_striping field (see Section 8.8) and applies to every data server in that mirror's ffv2s_data_servers list. Three values are permitted:

FFV2_STRIPING_NONE: The mirror is not striped. ffv2m_striping_unit_size MUST be 1 and ffv2m_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 21. Common definitions apply to both.

```
   L: logical offset within the file (bytes)
   U: stripe-unit size in bytes  = ffv2m_striping_unit_size
   W: stripe width               = length of ffv2s_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 21: 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 ffv2_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 v2 layout 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 18).

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 ffv2l_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 ffv2l_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 ffv2l_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 ffv2_ioerr4 indicates that the client successfully replayed the I/O to all mirrors. Any LAYOUTRETURN with a ffv2_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 22). 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 checksum

carries a 32-bit CRC computed over the header and the chunk. Because the checksum field is itself part of the header, the computation treats the bytes of that field as zero so that the result is independent of the field's wire value; the writer then stores the computed CRC into the checksum field for transmission. To validate on the read path, the receiver saves the received checksum, treats those bytes as zero, recomputes the CRC over the header and chunk, and compares against the saved value. By combining the two parts of the payload in the CRC, integrity is ensured for both 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 27).

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

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

Integrity: A payload has *integrity* when it is atomic AND every contained chunk passes its checksum 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. Atomicity detects protocol-level failures (racing writers, partial writes, rollback windows); the checksum 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 one data server may well be the "first"

on another. 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 checksum 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

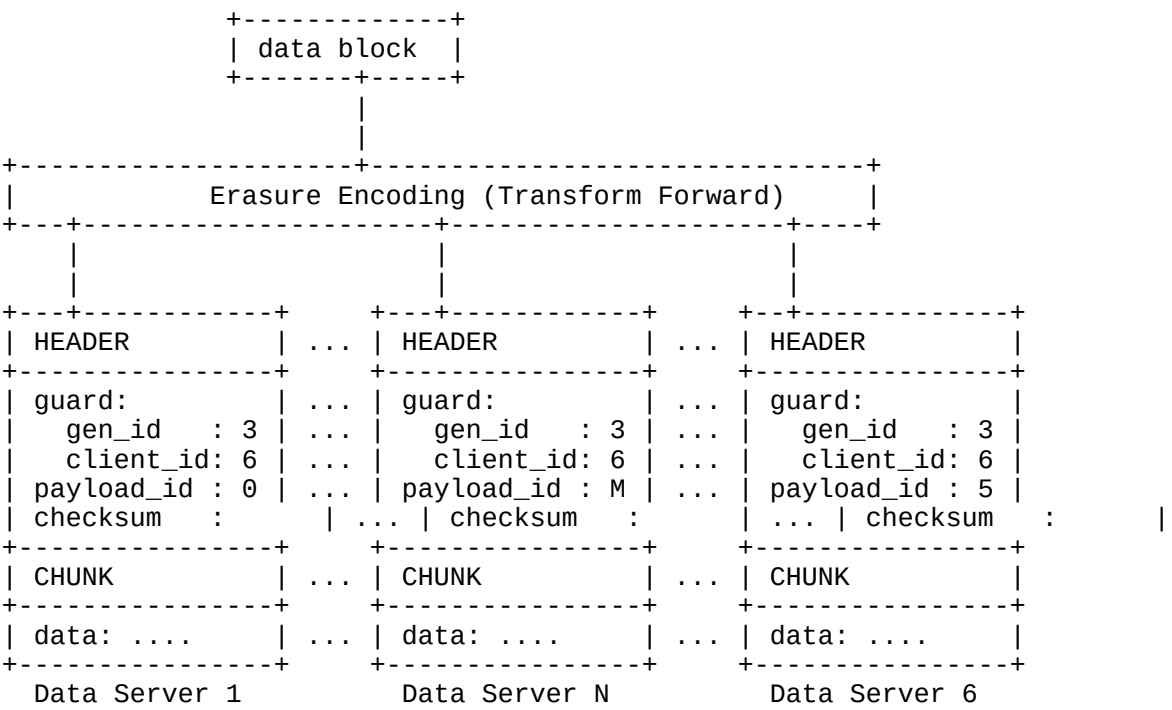


Figure 22: Encoding a Data Block

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 22. 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 23.

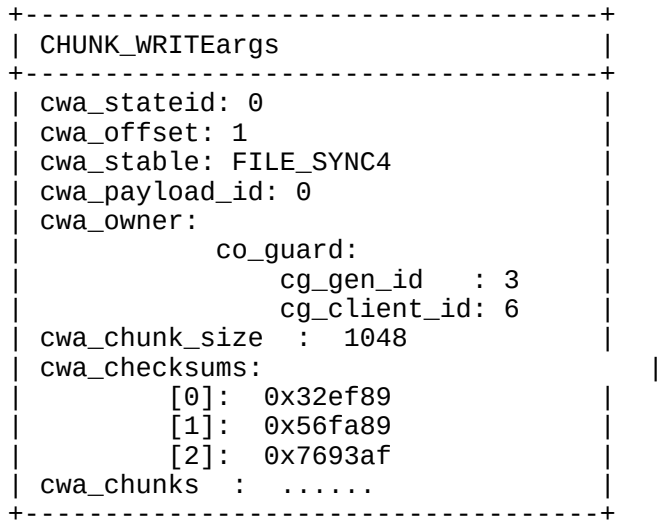


Figure 23: Example of CHUNK_WRITE_args

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 checksum from the cwa_checksums. The cwa_chunks are sent together as a byte stream to increase performance.

Assuming that there were no issues, Figure 24 illustrates the results. The payload sequence id is implicit in the CHUNK_WRITEargs.

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

Figure 24: Example of CHUNK_WRITE_res

11.2.1.1. Worked Example: Calculating the CRC32

The examples in this section and in Section 11.2.2.1 illustrate checksum computation and verification using CHECKSUM_ALG_CRC32 as the worked algorithm. Other registered checksum algorithms (CHECKSUM_ALG_CRC32C, CHECKSUM_ALG_FLETCHER4, CHECKSUM_ALG_SHA256, CHECKSUM_ALG_SHA512, CHECKSUM_ALG_BLAKE3; see Section 24.3) follow the same pattern -- the algorithm names a function over the header and chunk bytes, the writer fills cs_value with the computed output, and the reader recomputes and compares. Only the algorithm and the cs_value length differ.

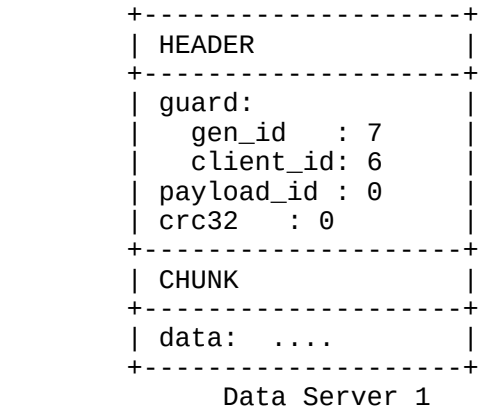


Figure 25: CRC32 Before Calculation

Assuming the header and payload as in Figure 25, the crc32 needs to be calculated in order to fill in the cwa_checksums entry 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 26.

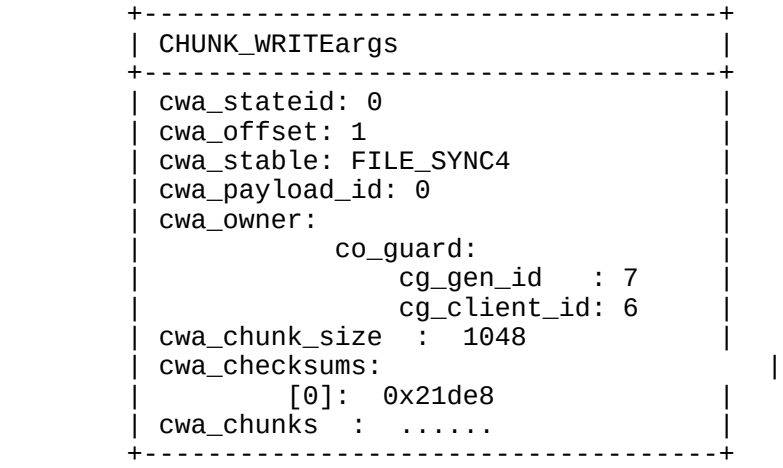


Figure 26: CRC32 After Calculation

11.2.2. Decoding a Data Block

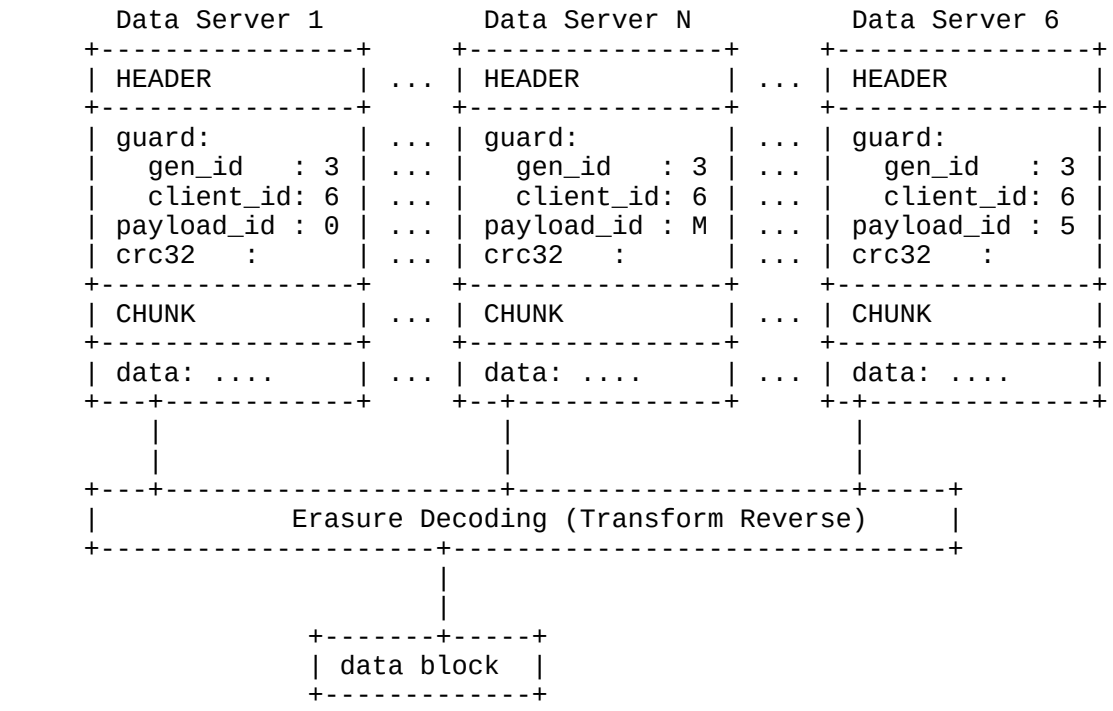


Figure 27: Decoding a Data Block

When reading chunks via a CHUNK_READ operation, the client will decode them into data blocks as shown in Figure 27.

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. Worked Example: Checking the CRC32

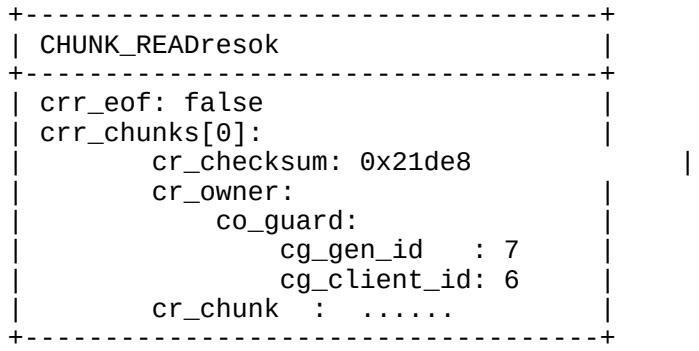


Figure 28: CRC32 on the Wire

Assuming the CHUNK_READ results as in Figure 28, the crc32 needs to be checked in order to ensure data integrity. Conceptually, a header and payload can be built as shown in Figure 29. 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.

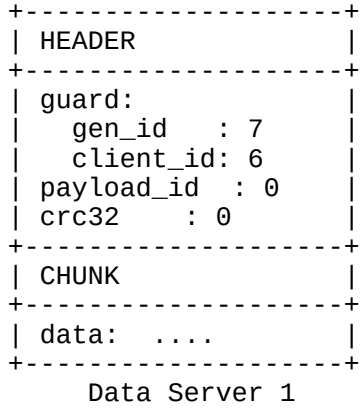


Figure 29: CRC32 Being Checked

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 ffv2l_flags in the ffv2_layout4 (see Figure 17) 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 non-atomicity. This directly follows from Section 11.2.7 and MUST be handled as discussed there. Once atomicity in the payload has been detected, the client can use those chunks as a basis for read/modify/update.

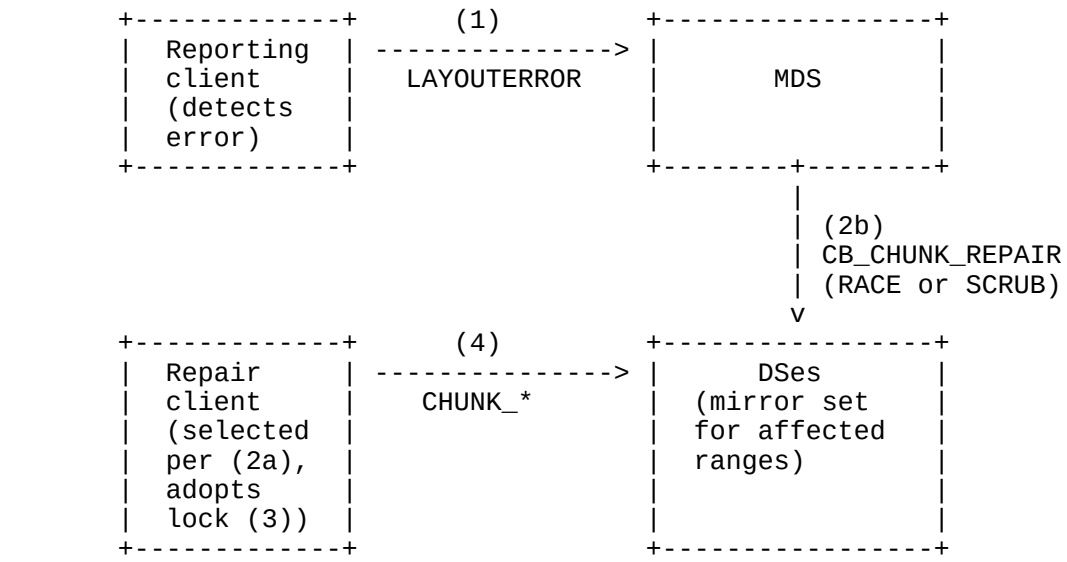
CHUNK_WRITE is a two pass operation in cooperation with CHUNK_FINALIZE (Section 25.3) and CHUNK_ROLLBACK (Section 25.8). It writes to the data file and the data server is responsible for retaining a copy of the old header and chunk. A subsequent CHUNK_READ would return the new chunk. However, until either 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 atomic data block, then the client will send a CHUNK_FINALIZE in a subsequent compound to inform the data server that the chunk is finalized and can be overwritten by another CHUNK_WRITE.

If the CHUNK_WRITE results in an non-atomic 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_ATOMIC.

11.2.4. Selecting the Repair Client

The repair topology involves three actors communicating along distinct paths, as shown in Figure 30.



(1) Reporting client LAYOUTERRORs the metadata server.
(2a) Metadata server selects a repair client (may be same as the reporting client).
(2b) Metadata server escrows the chunk lock and issues CB_CHUNK_REPAIR to the selected repair client.
(3) Repair client adopts the lock and drives the repair.
(4) Repair client issues CHUNK_LOCK_ADOPT, CHUNK_WRITE_REPAIR, CHUNK_FINALIZE, CHUNK_COMMIT, and CHUNK_REPAIRED against the mirror set.

Figure 30: Repair topology

The metadata server is the authority that selects which client (or, in a tightly coupled deployment, which data server) repairs an non-atomic 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 atomic 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_ATOMIC 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:

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.

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.

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.

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.

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.

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 non-atomic 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 ffv2l_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 checksum.

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, non-atomic 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 a failed state (PENDING with a CRC mismatch, or in the errored state set by a prior CHUNK_ERROR).

2. For each errored chunk, the repair client reconstructs the correct data using the erasure coding algorithm (RS matrix inversion or Mojette back-projection) from the surviving atomic chunks (treating each chunk's payload as a shard of the stripe).

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. Transitioning from Single Writer Mode to Multiple Writer Mode

When a second writer requests a write layout for a file currently covered by a single-writer layout (FFV2_FLAGS_ONLY_ONE_WRITER set), the metadata server recalls the existing layout before granting the new request. The sequence is:

1. The metadata server issues CB_LAYOUTRECALL to the single-writer client.

2. The single-writer client drains its outstanding I/O issued under the single-writer assumption (CHUNK_WRITE with cwa_guard.cwg_check = FALSE). Operations already underway complete under the layout that authorized them: CHUNK_FINALIZE and CHUNK_COMMIT proceed normally for blocks already written.

3. Once drained, the single-writer client issues LAYOUTRETURN.

4. The metadata server grants the new writer a layout without FFV2_FLAGS_ONLY_ONE_WRITER set. When the original writer next issues LAYOUTGET, it also receives a layout without the flag. Both clients then operate in multiple writer mode (Section 11.2.6.4), supplying cwa_guard.cwg_check = TRUE and a chunk_guard4 on every CHUNK_WRITE.

The transition uses standard NFSv4.1 layout recall semantics (Section 12.5.5 of [RFC8881]). Drained single-writer I/O does not need to be re-issued under multiple writer rules; it completed under the layout that authorized it. If the single-writer client fails to return the layout within the recall window, the metadata server escalates to layout revocation (Section 12.5.5.2.1 of [RFC8881]); any single-writer writes that did not complete before revocation are repaired via the multiple-writer repair path on subsequent access.

11.2.6.4. 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 chunk. The data server checks that no other client's chunk is in the PENDING state at this offset. If another client's chunk 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 chunks 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 chunks carrying the shards of an atomic erasure-coded stripe all carry the same chunk_guard4. A reader that encounters chunks with different guard values knows the stripe is not yet atomic and MUST either retry or report NFS4ERR_PAYLOAD_NOT_ATOMIC.

11.2.6.5. Repairing Multiple Writer Payloads

In multiple writer mode, non-atomic chunks can arise from two sources: a client failure leaving some chunks 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 atomic set across all k data shards.

3. If a atomic 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 non-atomic, followed by CHUNK_FINALIZE and CHUNK_COMMIT.

4. If no atomic 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 atomic, 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.

11.2.6.6. Transitioning from Multiple Writer Mode to Single Writer Mode

The reverse transition is optional. When the metadata server determines that only one writer holds a write layout for a file (for example, because other writers' layouts have been returned or their leases have expired), it MAY recall the remaining writer's layout and grant a fresh layout with FFV2_FLAGS_ONLY_ONE_WRITER set, restoring the single-writer optimization. The metadata server MAY also leave the writer in multiple writer mode indefinitely; single writer mode is an optimization, not a correctness requirement.

The metadata server's choice of when to grant FFV2_FLAGS_ONLY_ONE_WRITER is policy and is implementation-defined. A metadata server that aggressively grants single writer mode and then must recall it each time a second writer appears can produce recall churn under workloads with irregular concurrent access: each single-writer-to-multiple-writer transition costs a CB_LAYOUTRECALL round trip and drain time for in-flight I/O. Strategies to limit churn include withholding FFV2_FLAGS_ONLY_ONE_WRITER until sustained single-writer behavior is observed, deferring the single-writer grant after a recent recall, or never granting single writer mode for files with a history of concurrent access.

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 atomic depend on both the Erasure Coding Type and the level of protection selected. If the client has enough atomic chunks in the payload, then it can proceed to use them to build a data block. If it does not have enough atomic chunks in the payload, then it can either decide to return a LAYOUTERROR of NFS4ERR_PAYLOAD_NOT_ATOMIC to the metadata server or it can retry the CHUNK_READ until there are enough atomic 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 atomic. As such, it might even be the non-atomic 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 *Proxy Server* 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 Proxy Server 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 proxy-server-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 Proxy Server mechanism is specified in [I-D.haynes-nfsv4-flexfiles-v2-proxy-server].

Implementations that do not support the Proxy Server 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 Proxy Server 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. Mixing coding types in a single file's mirror set addresses several use cases:

* Assimilation of a non-erasure-coded file into an erasure-coded representation, or export of an erasure-coded file to a non-erasure-coded representation.

* Online migration between encodings, for example from a Reed-Solomon Vandermonde encoding to a Mojette systematic encoding when a read-access-pattern change makes the new codec a better fit. Both representations remain addressable through the layout throughout the transition.

* Cross-encoding recovery: when one encoding loses data to a correlated failure mode (a codec implementation bug, a memory-corruption pattern that affects parity shards identically), a second mirror in a different encoding provides an independent recovery surface.

* Client-capability routing: a Proxy Server ([I-D.haynes-nfsv4-flexfiles-v2-proxy-server]) sees the full mirror set and chooses between encodings on behalf of clients that do not implement every codec the file is represented in.

Consider a layout that exposes a file in two encodings simultaneously: a PASSTHROUGH mirror over the original byte stream and a Reed-Solomon Vandermonde (FFV2_ENCODING_RS_VANDERMONDE) mirror with 8 active data shards (plus 2 parity and 2 spare data servers). A layout for such a file might appear as in Figure 31. Both representations are active and addressable through the layout simultaneously. This is the transition-window pattern: a file may transiently span encodings while it is being assimilated from a non-FFv2 source or migrated between codecs. Steady state is homogeneous; the multi-encoding window is what the protocol must accommodate.

The active mirrors serve different access patterns concurrently:

* A client that speaks only the file-layout READ path issues READ (Section 18.22 of [RFC8881]) calls to index 0 (the PASSTHROUGH mirror).

* A client that speaks the chunked path issues CHUNK_READ (Section 25.6) calls to index 1 (the RS_VANDERMONDE mirror).

* A Proxy Server fronting legacy clients chooses between the two encodings on the client’s behalf ([I-D.haynes-nfsv4-flexfiles-v2-proxy-server]).

All three patterns coexist during the transition.

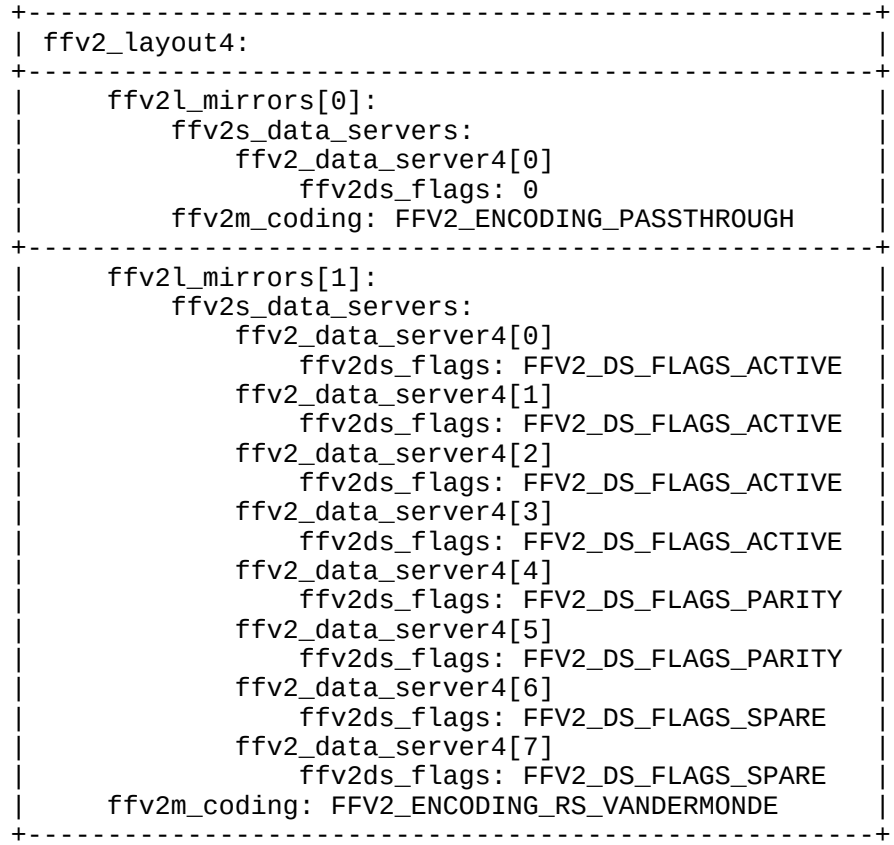


Figure 31: Example of Mixed Coding Types in a Layout

When performing I/O via a FFV2_ENCODING_PASSTHROUGH coding type, the non-transformed data will be used; whereas with the chunked coding types (FFV2_ENCODING_MIRRORED, FFV2_ENCODING_MOJETTE_*, FFV2_ENCODING_RS_VANDERMONDE), 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 72), 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 translate 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 fixed-width words combined with bitwise XOR -- the additive group of GF(2)^W where W is the element width in bits. Encoders and decoders MUST use XOR; modular integer addition is not equivalent and is not interoperable. XOR has no carry chain, is its own inverse (so the residual subtraction in reconstruction is identical to the forward accumulation), and scales straightforwardly to wider SIMD lanes (NEON, SSE, AVX, AVX-512) without requiring multiplicative Galois field operations. The element width W is an implementation choice; 64-bit elements are the conventional choice and align well with NEON, SSE2, and AVX2 vector widths.

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 XORs the grid elements that lie on a discrete line through the grid. This specification adopts the bin convention of [NORMAND]: a grid cell at (row, col) maps to bin

```
b = row * p + col * q - off
```

where off is chosen so that the smallest reachable bin index is 0 (off = min over all (row, col) in [0, Q) x [0, P) of row * p + col * q). All implementations MUST use this convention -- the alternative "transposed" convention b = col * p - row * q + off produces a different bin ordering and is not interoperable.

The full forward transform along direction (p, q) is then:

```
Projection(b, p, q) = XOR over all (row, col) where
                        row * p + col * q - off = 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 choice of inverse algorithm is purely an implementer concern: all correct inverses produce byte-identical plaintext from the same shards and bin layout, so the choice has no on-the-wire impact. Two well-known algorithms apply.

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, XOR it back through all projections.

4. Repeat until all elements are recovered.

Corner peeling runs in O(n * P * Q) and is the simplest correct inverse. Implementations MAY instead use the geometry-driven inverse of [NORMAND], which precomputes a recurrence over the sorted projection slopes and walks each line once: it eliminates the inner singleton search and runs substantially faster on the parameter ranges typical of flexible file v2 layout deployments (high redundancy, wide stripes), with no change to the shards or to the reconstructed plaintext.

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 cost	Zero	Zero	Full decode
GF operations	Yes (GF(2^8))	No	No

Table 4: Comparison of erasure encoding types

Reed-Solomon uses uniform shard sizes and GF(2^8) operations. Mojette systematic provides zero-cost healthy reads with variable parity shard sizes; reconstruction cost scales as O(m * k) rather

than O(k^2). Mojette non-systematic encodes all k + m shards as projections, providing constant decode cost regardless of failure count at a higher baseline read cost than systematic. The choice among these is a deployment decision driven by workload characteristics and operational priorities.

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 non-atomic 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 atomic 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-data server 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 chunk identifies which transaction wrote it. A reader that finds chunks with different guard values detects the non-atomicity and either retries (if a concurrent write is still in progress) or reports NFS4ERR_PAYLOAD_NOT_ATOMIC 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-data server 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 12.5) 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-data server 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. Chunks Are Not Blocks

The chunk is a protocol-level primitive distinct from a block. Throughout this document, "block" refers to a byte range in the file's address space (the application's view); "chunk" refers to the addressable unit carried by the CHUNK_* operations, which has an envelope that blocks do not.

A chunk carries five properties that a block does not:

* *Atomicity.* The chunk_guard4 compare-and-swap guard (Section 24.1) sequences concurrent writers and rejects torn-write attempts. Block I/O has no comparable primitive; concurrent block writes either serialize at the storage layer or interleave unpredictably.

* *Integrity.* The checksum in each chunk header is computed over the header and payload and verified end-to-end on the read path (Section 25.6). Block I/O carries no integrity tag; data-corruption detection is delegated to the underlying storage medium or is absent.

* *Provenance.* The chunk_owner4 (Section 24.2) records which transaction produced the chunk. Block I/O carries no per-write provenance; a block's bytes have no protocol-visible producer.

* *Lifecycle state.* A chunk progresses through PENDING -> FINALIZED -> COMMITTED via CHUNK_FINALIZE / CHUNK_COMMIT (Section 12.5). Block I/O has no lifecycle states; a block is either present or absent.

* *Lock continuity across revocation.* The chunk's lock (Section 25.5) is transferred to the metadata server in escrow when a holder's stateid is revoked, and adopted by a repair client via CHUNK_LOCK_FLAGS_ADOPT. Block I/O has no per-block locking and no continuity mechanism; client failure leaves any external lock indeterminate.

Each of these properties is load-bearing for some part of the flexible file v2 layout's consistency story: the chunk_guard4 CAS underlies multi-writer correctness; the checksum underlies end-to-end integrity; lock escrow underlies repair coordination across stateid revocation; the state machine underlies the PENDING / FINALIZED / COMMITTED distinction that enables rollback and repair. Removing any one of them would change what the protocol can guarantee.

A protocol that exchanges file data as byte ranges with no envelope -- whether described as "block I/O" or as "generic data movement" -- is not interoperable with this specification's CHUNK_* operations. The CHUNK_* operations are not a byte-range I/O surface with optional integrity bolted on; they are a chunk-protocol surface in which the envelope is the primitive.

12.3. 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.

An entity MAY simultaneously hold more than one of these roles with respect to a given data server, with each role bound to a distinct session. A metadata server that opens a control session to a data server (presenting EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID; see Section 6.4.2) issues TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID on that session; on a separate client-side session (presenting EXCHGID4_FLAG_USE_NON_PNFS), the same metadata server MAY also issue CHUNK_* operations as a data-path client. A data server MUST NOT assume that the metadata server is not also one of its clients; it distinguishes MDS-only operations from client-side operations by the EXCHANGE_ID flags of the session that carries the operation, not by the requester's IP address or principal.

A data server MAY likewise act as a client of another data server -- for example, when selected as the repair client by an MDS-directed CB_CHUNK_REPAIR. Independent of the actor role, any entity may operate as codec-aware (issuing CHUNK_* operations directly against data servers) or codec-unaware (operating through the proxy-server-mediated READ / WRITE path described in [I-D.haynes-nfsv4-flexfiles-v2-proxy-server]). Proxy-server registration carries the codec capability explicitly; direct pNFS clients reveal their codec posture implicitly through the operations they issue.

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 metadata server failover) is implementation-defined and out of scope for this document; see Section 12.9.

12.4. 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 checksum (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 12.10.

Authenticated writers and their own data: An authenticated client

may write arbitrary (even semantically-invalid) bytes into chunks it owns. The checksum 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 metadata server 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 non-atomic 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 coding has lost too many shards across both sides of the partition to reconstruct any single atomic 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.

12.5. 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.

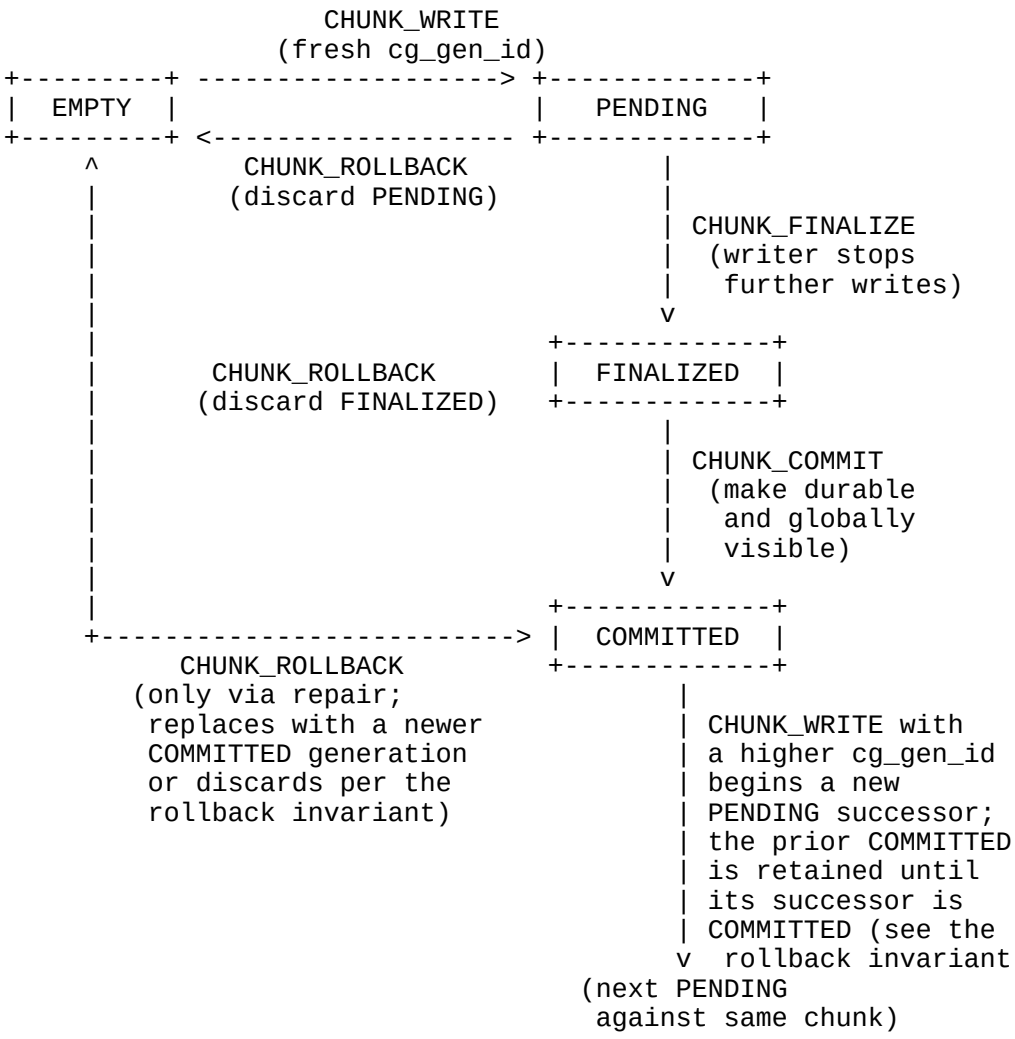


Figure 32: Chunk lifecycle on the data server

CHUNK_WRITE against a chunk already in PENDING from the same writer with the same cg_gen_id is a self-transition on PENDING: the data server replaces the PENDING payload in place and the state does not change. This case is not drawn in Figure 32 for clarity.

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 12.6). 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 12.6 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.

Figure 32 covers the lifecycle of a chunk’s payload but not the lock that may be held on it. The lock has its own state machine, shown in Figure 33.

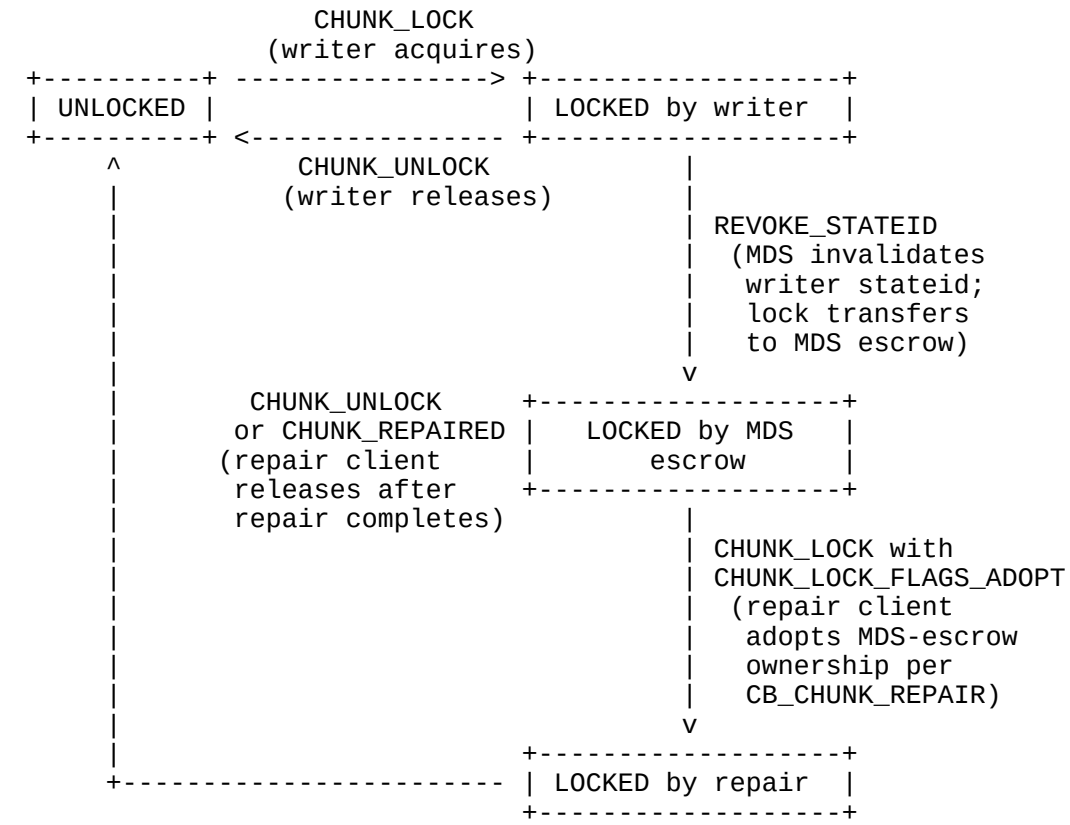


Figure 33: Chunk lock ownership on the data server

The lock state machine is orthogonal to the chunk lifecycle in Figure 32: a chunk in any of EMPTY, PENDING, FINALIZED, or COMMITTED MAY simultaneously be in any of the four lock states. The errored bit (set by CHUNK_ERROR, cleared by CHUNK_REPAIRED) is a third orthogonal axis and is not drawn; CHUNK_ERROR may set the bit in any state, and CHUNK_REPAIRED clears it as part of completing a repair sequence. A CHUNK_LOCK that arrives while the chunk is already LOCKED by a different owner returns NFS4ERR_CHUNK_LOCKED with the existing owner’s chunk_owner4 in clr_owner (Section 25.5).

12.6. 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 ffv2m_striping_unit_size and the stripe width W in Figure 21; 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 12.7 for the normative scoping rule.

12.7. Ownership and Scope of Retained Prior Content

The rollback invariant in Section 12.6 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 metadata server will treat as the authoritative owner for purposes of Section 12.8.

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 stated in the "Visibility of non-committed state" bullet of Section 12.6.

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 12.8 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.

12.8. 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 coding 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 12.7) 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 12.7. 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 24.1.4) 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.

12.9. 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:

Designated coordinator: The metadata server is the coordinator for a file. Clients accept the metadata server'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.

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.

Pessimistic locks off the critical path: CHUNK_LOCK is used only during repair, never on the normal write path. Lock escrow (see Section 24.1.4) preserves the "exactly one owner" invariant across stateid revocation without requiring a consensus round to elect the next owner.

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.

12.10. 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. Metadata server high availability, 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 checksum; see Section 27.1. A bit-flip-class checksum is a transport-integrity check, not an adversarial-integrity check; cryptographic-class checksums detect adversarial modification by anyone other than the authenticated writer.

13. NFSv4.2 Operations Allowed to Data Files

In the Flexible File 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 Flexible File 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 12.3).

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 metadata server 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 proxy server operation). The list above is the minimum set a flexible file v2 layout data server MUST support for the metadata server's use.

13.2. Data Path: Client to Data Server

A pNFS client with an active flexible file 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. Stateid Model on the Data Server

The stateid presented on a CHUNK_* operation is a *layout stateid* returned by a prior LAYOUTGET against the metadata server (see Section 18.43 of [RFC8881]), NOT an open stateid, byte-range lock stateid, or delegation stateid. A pNFS client does NOT issue OPEN against the data server. This is a meaningful departure from the stateid model in Section 18.32 of [RFC8881] (which states that the WRITE stateid "represents a value returned from a previous byte-range LOCK or OPEN request or the stateid associated with a delegation"), and clients implementing Flexible File Version 2 MUST NOT carry over

those expectations to the data path.

The three roles the RFC 8881 stateid plays on a regular NFSv4 server split apart in the Flexible File Version 2 data-server model:

Open and share-mode tracking: Lives at the metadata server, established by OPEN ([RFC8881] Section 18.16) on the metadata-server filehandle. The metadata server's open stateid is NOT exposed to data servers; share-mode conflicts are resolved at the metadata server before LAYOUTGET grants a layout.

Byte-range lock tracking: Does not apply at the data server. Locking on the data path is chunk-range rather than byte-range, expressed via CHUNK_LOCK (Section 25.5), and the lock holder is identified by chunk_owner4 (the {cg_client_id, cg_gen_id} pair) rather than by a lock stateid. A client wanting byte-range locks on a file MUST acquire them on the metadata-server filehandle, where standard [RFC8881] Section 12 byte-range locking applies.

I/O authorization on the data server: The layout stateid carried on CHUNK_* operations. Under tight coupling (Section 6.4), the metadata server registers each issued layout stateid with the data server via TRUST_STATEID (Section 25.12) and the data server validates subsequent CHUNK_* stateids against the trust table. Under loose coupling, the data server treats the layout stateid as an opaque per-client token and authorizes by the synthetic uid/gid the layout carries (see Section 6.2).

Because the layout stateid does authorization but does not identify a per-open or per-lock owner, a single client may present the same layout stateid on many CHUNK_* operations across many parallel writers within the client, without any of the open-owner ordering constraints [RFC8881] Section 8.2.2 imposes on regular NFSv4 stateids. Chunk- level write ordering and contention are resolved by the per-chunk chunk_guard4 CAS (Section 24.1) and the chunk-range CHUNK_LOCK, not by stateid-owner sequencing.

13.2.3. 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 proxy server described informally in Section 12.3 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 proxy server transition. A client that uses a data-server GETATTR result to determine the file's visible size will observe inconsistencies.

13.2.4. 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 metadata server 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. See Section 13.2.5 for how the metadata server handles truncate on erasure-coded files. Similarly, a client MUST NOT issue DEALLOCATE against a data file; see the next subsection.

13.2.5. MDS-Driven Truncate on Erasure-Coded Files

A client that wants to truncate an erasure-coded file MUST issue SETATTR(FATTR4_SIZE) to the metadata-server filehandle (see Section 13.2.4). The metadata server translates the logical truncate into per-shard size changes across the data servers in each mirror.

Stripe-aligned truncate: When the new size lies on a stripe boundary (including zero), no chunk re-encoding is required. The metadata server computes per-shard sizes from the codec geometry it issued in the layout (k, m, and the projection parameters for Mojette; see Section 11.5) and issues per-data-server SETATTR(FATTR4_SIZE) with the computed per-shard size. Geometry parameters are sufficient arithmetic; no codec implementation is required at the metadata server.

Non-stripe-aligned truncate: When the new size falls within a stripe, the data shards covering the partial stripe must be truncated and the parity shards re-encoded from the truncated data. Because re-encoding requires running the erasure transform,

the metadata server MUST delegate this case to a codec-aware actor: either a Proxy Server ([I-D.haynes-nfsv4-flexfiles-v2-proxy-server]) for proxy-mediated truncate, or a codec-aware client selected per Section 11.2.4 via CB_CHUNK_REPAIR with the affected partial-stripe chunks as the repair target. If neither path is available, the metadata server MUST return NFS4ERR_NOTSUPP to the originating SETATTR.

The metadata server knows codec geometry from the layout but is not required to include a codec implementation. The delegation rule above accommodates a metadata server that has geometry knowledge only.

13.2.6. PASSTHROUGH Data Files (FFV2_ENCODING_PASSTHROUGH)

For a mirror whose ffv2m_coding_type_data is FFV2_ENCODING_PASSTHROUGH (see Section 8.1.2), client operations on the data file follow the same pattern as the File Layout Type in [RFC8881] Section 13.6 and the Flexible File Version 1 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.

13.2.7. Chunked Data Files (FFV2_ENCODING_MIRRORED, FFV2_ENCODING_MOJETTE_*, FFV2_ENCODING_RS_VANDERMONDE)

For a mirror whose ffv2m_coding_type_data is any of the chunked coding types defined in this document (FFV2_ENCODING_MIRRORED, 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 ffv2m_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.

* SETATTR against an erasure-coded data file (the general prohibition in Section 13.2.4 applies to all data files; truncate in particular is handled by the metadata server per Section 13.2.5).

13.2.8. 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 metadata server 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

The classification below adapts the operation taxonomy of [RFC8881] Section 17 (REQUIRED / RECOMMENDED / OPTIONAL / MUST NOT IMPLEMENT) to the two-direction per-operation view a Flexible File Version 2 data server requires. Two of the four labels in the table below match [RFC8881] usage; the other two are extensions specific to this document.

REQUIRED: The data server MUST support the operation on this path. Matches [RFC8881] Section 17 REQ.

OPTIONAL: The data server MAY support the operation; if it does, the actor in this column MUST tolerate the absence of support. Matches [RFC8881] Section 17 OPT.

MUST NOT: The actor in this column MUST NOT send the operation, and the data server MUST reject it with NFS4ERR_NOTSUPP. This per-direction prohibition extends [RFC8881] Section 17's single-axis MUST NOT IMPLEMENT classification: an operation may be forbidden on one path (client to data server) while required on another (metadata server to data server). SETATTR is the canonical example.

MAY: The metadata server MAY use the operation as an implementation-defined control-plane action. Not in [RFC8881] Section 17; specific to the metadata-server-to- data-server path in this document.

Operation	Client -> data server	metadata server -> data server
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	OPTIONAL (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	OPTIONAL (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	OPTIONAL (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 Version 2 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 ffv2_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 ffv2_layoutreturn4) allows for an NFSv4.1 metadata server to utilize the data.

14.1. I/O Error Reporting

14.1.1. ffv2_ioerr4

```
/// struct ffv2_ioerr4 {
///         offset4        ffv2ie_offset;
///         length4        ffv2ie_length;
///         stateid4       ffv2ie_stateid;
///         device_error4  ffv2ie_errors<>;
/// };
///
```

Figure 35: ffv2_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 ffv2_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, ffv2ie_errors.de_deviceid, ffv2ie_offset, and ffv2ie_length represent the storage device and byte range within the file in which the error occurred; ffv2ie_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       ffv2il_ops_requested;
///         uint64_t       ffv2il_bytes_requested;
///         uint64_t       ffv2il_ops_completed;
///         uint64_t       ffv2il_bytes_completed;
///         uint64_t       ffv2il_bytes_not_delivered;
///         nfstime4       ffv2il_total_busy_time;
///         nfstime4       ffv2il_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 ffv2l_read. Both WRITE and COMMIT operations are used for the ff_io_latency4 ffv2l_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.

ffv2il_bytes_not_delivered is used to track the aggregate number of bytes requested but not fulfilled due to error conditions. ffv2il_total_busy_time is the aggregate time spent with outstanding RPC calls. ffv2il_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         ffv2l_addr;
///         nfs_fh4          ffv2l_fhandle;
///         ffv2_io_latency4 ffv2l_read;
///         ffv2_io_latency4 ffv2l_write;
///         nfstime4         ffv2l_duration;
///         bool             ffv2l_local;
/// };
///
```

Figure 38: ff_layoutupdate4

ffv2l_addr differentiates which network address the client is connected to on the storage device. In the case of multipathing, ffv2l_fhandle indicates which read-only copy was selected. ffv2l_read and ffv2l_write convey the latencies for both READ and WRITE operations, respectively. ffv2l_duration is used to indicate the time period over which the statistics were collected. If true, ffv2l_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            ffv2is_offset;
///         length4            ffv2is_length;
///         stateid4           ffv2is_stateid;
///         io_info4           ffv2is_read;
///         io_info4           ffv2is_write;
///         deviceid4          ffv2is_deviceid;
///         ffv2_layoutupdate4 ffv2is_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, ffv2is_offset and ffv2is_length represent the starting offset of the range and the range length in bytes. ffv2is_read.ii_count, ffv2is_read.ii_bytes, ffv2is_write.ii_count, and ffv2is_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 ffv2is_deviceid and ffv2l_addr uniquely identifies both the storage path and the network route to it. Finally, ffv2l_fhandle allows the metadata server to differentiate between multiple read-only copies of the file on the same storage device.

14.3. ffv2_layoutreturn4

```
/// struct ffv2_layoutreturn4 {
///         ffv2_ioerr4     ffv2lr_ioerr_report<>;
///         ffv2_iostats4   ffv2lr_iostats_report<>;
/// };
///
```

Figure 41: ffv2_layoutreturn4

When data file I/O operations fail, ffv2lr_ioerr_report<> is used to report these errors to the metadata server as an array of elements of type ffv2_ioerr4. Each element in the array represents an error that occurred on the data file identified by ffv2ie_errors.de_deviceid. If no errors are to be reported, the size of the ffv2lr_ioerr_report<> array is set to zero. The client MAY also use ffv2lr_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 Version 2 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 v2 layout, this means that LAYOUTERROR4args is treated the same as ffv2_ioerr4.

16. Flexible File Version 2 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 v2 layout, this means that LAYOUTSTATS4args.lsa_layoutupdate is overloaded with the same contents as in ffv2is_layoutupdate.

17. Flexible File Version 2 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 flexible file v1 layouts. For flexible file v2 layouts, clients SHOULD use ffv2_layouthint4 (Figure 20) instead, which provides coding type selection and data protection geometry hints via ffv2_data_protection4 (Figure 19).

19. Recalling a Layout

While Section 12.5.5 of [RFC8881] discusses reasons independent of layout type for recalling a layout, the flexible file v2 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 v2 layouts the client is allowed to keep. The mask flags for the flexible file v2 layout 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. Layout Revocation and 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 metadata server MAY revoke client layouts and reassign these resources to other clients (see Section 12.5.5 of [RFC8881]). To avoid data corruption from a revoked client continuing to issue I/O, the metadata server MUST fence the revoked client from the affected data files. The mechanism varies by coupling model and by whether the client’s layout stateid has been registered with the data servers via TRUST_STATEID:

Loosely coupled, untrusted stateid: The metadata server rotates the

synthetic uid/gid on the affected data files per Section 6.2. The revoked client presents stale RPC credentials and receives NFS4ERR_ACCESS from the data server. This is the FFv1-style fencing mechanism; it operates per data file and does not distinguish between clients that hold layouts on the same file.

Tightly coupled, trusted stateid: When the metadata server has registered a client's layout stateid with the data servers via TRUST_STATEID (Section 25.12), it can revoke per-client access without rotating credentials by issuing REVOKE_STATEID (Section 25.13) to each affected data server, or BULK_REVOKE_STATEID (Section 25.14) when revoking all stateids belonging to a single clientid4 across a data server. Subsequent I/O from the revoked client carrying the revoked stateid receives NFS4ERR_BAD_STATEID. This is the preferred mechanism for erasure-coded layouts because it is per-client and avoids the FFv1 limitation of fencing all clients on a data file when only one needs to be revoked.

Mixed: A metadata server MAY combine the two mechanisms when a file's layout includes both PASSTHROUGH mirrors (where stateid registration is not in play) and erasure-coded mirrors with trusted stateids. The metadata server rotates synthetic ids for the PASSTHROUGH mirror's data file and issues REVOKE_STATEID for the erasure-coded mirror's data servers.

21. New NFSv4.2 Error Values

```
///
/// /* Erasure Coding error constants; added to nfsstat4 enum */
///
/// const NFS4ERR_CODING_NOT_SUPPORTED         = 10097;
/// const NFS4ERR_PAYLOAD_NOT_ATOMIC           = 10098;
/// const NFS4ERR_CHUNK_LOCKED                 = 10099;
/// const NFS4ERR_CHUNK_GUARDED                = 10100;
/// const NFS4ERR_PAYLOAD_LOST                 = 10101;
/// const NFS4ERR_LAYOUT_CHECKSUM_NOT_SUPPORTED = 10102;
///
```

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_ATOMIC	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
NFS4ERR_LAYOUT_CHECKSUM_NOT_SUPPORTED	10102	Section 21.1.6

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_ATOMIC (Error Code 10098)

The client encountered a payload in which the blocks were non-atomic and stay non-atomic. 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.

21.1.6. NFS4ERR_LAYOUT_CHECKSUM_NOT_SUPPORTED (Error Code 10102)

Returned by the client on LAYOUTRETURN to indicate that the layout's ffv2m_checksum_algorithm (Section 8.8) names a checksum_algorithm4 (Section 24.3) that the client does not implement. The client returns the layout with this error code rather than attempting CHUNK_* operations it cannot validate.

On receipt, the metadata server MAY:

* issue a new layout for the same file naming a different checksum_algorithm4 that the client supports (if the file's policy permits any of the algorithms the client does support); or

* deny the layout request, in which case the client MUST either fall back to MDS-mediated I/O or report an I/O error to the application.

NFS4ERR_LAYOUT_CHECKSUM_NOT_SUPPORTED is distinct from NFS4ERR_BADLAYOUT (generic "this layout shape is unusable"): the explicit per-checksum-algorithm signal lets the metadata server discriminate "client can't read this layout because of the checksum algorithm" from "client can't read this layout for some other reason" and respond accordingly.

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_BAD_STATEID, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_ERROR	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_FINALIZE	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_INVAL, NFS4ERR_IO, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_HEADER_READ	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_IO, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_LOCK	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_CHUNK_LOCKED, NFS4ERR_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_READ	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID,

		NFS4ERR_DELAY, NFS4ERR_FHEXPIRED, NFS4ERR_IO, NFS4ERR_NOTSUPP, NFS4ERR_PAYLOAD_NOT_ATOMIC, NFS4ERR_SERVERFAULT, NFS4ERR_STALE
CHUNK_REPAIRED	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_ROLLBACK	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_UNLOCK	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, NFS4ERR_INVAL, NFS4ERR_NOTSUPP, NFS4ERR_SERVERFAULT
CHUNK_WRITE	NFS4_OK, NFS4ERR_ACCESS, NFS4ERR_BADXDR, NFS4ERR_BAD_STATEID, 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_BAD_STATEID, 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.

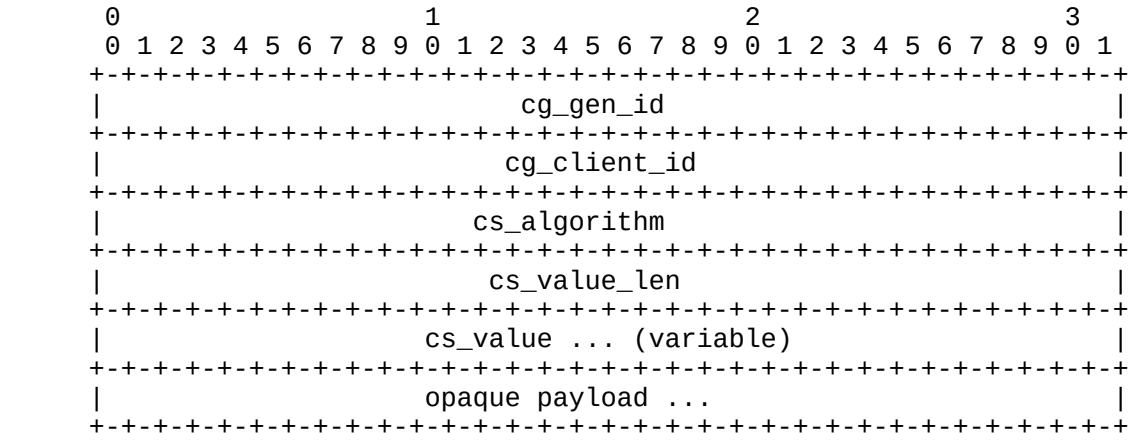
24. New NFSv4.2 Common Data Structures

24.1. chunk_guard4

```
/// const CHUNK_GUARD_CLIENT_ID_NONE = 0x00000000;
/// 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 chunk_guard4 header followed by the tagged checksum4 and then 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:	cs_algorithm	(checksum_algorithm4)
Bytes 12-15:	cs_value_len	(XDR opaque length prefix)
Bytes 16-N:	cs_value	(checksum bytes; length per cs_algorithm's registered output)
Bytes N+1-M:	opaque payload	(encoded shard; variable length)

The checksum block (cs_algorithm + cs_value_len + cs_value) is the XDR encoding of one checksum4 ({{fig-checksum4}}). For CHECKSUM_ALG_NONE the cs_value_len is zero and the payload follows immediately after byte 15.

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 ffv2m_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 ffv2m_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_NONE (0x00000000) MUST NOT be assigned to any client. Reserving 0 prevents an uninitialized cg_client_id field from passing as a real client and ensures the deterministic tiebreaker (numerically lowest wins) does not encode an implicit priority via assignment of 0.

* 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 24.1.4).

* 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_NONE

The value CHUNK_GUARD_CLIENT_ID_NONE (0x00000000) is reserved. It does not denote any client. Reserving 0 prevents an uninitialized cg_client_id field from passing as a real client and ensures the deterministic tiebreaker (numerically lowest wins, see Section 24.1) does not encode an implicit priority via assignment of 0.

Clients MUST NOT present CHUNK_GUARD_CLIENT_ID_NONE 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.

24.1.4. 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.

24.3. checksum4

```
/// typedef uint32_t   checksum_algorithm4;
///
/// const CHECKSUM_ALG_NONE      = 0;
/// const CHECKSUM_ALG_CRC32     = 1;
/// const CHECKSUM_ALG_CRC32C    = 2;
/// const CHECKSUM_ALG_FLETCHER4 = 3;
/// const CHECKSUM_ALG_SHA256    = 4;
/// const CHECKSUM_ALG_SHA512    = 5;
/// const CHECKSUM_ALG_BLAKE3    = 6;
/// /* Additional values registered with IANA;
///    see Section "Checksum Algorithm Registry" in
///    the IANA Considerations. */
///
/// struct checksum4 {
///     checksum_algorithm4   cs_algorithm;
///     opaque                cs_value<>;
/// };
```

Figure 53: XDR for checksum4

The checksum4 (see Figure 53) is a tagged checksum value used to detect transport corruption and on-disk bit rot of chunk payloads. Every chunk on the wire and at rest carries a checksum4 alongside its chunk_owner4.

cs_algorithm: identifies the checksum algorithm. The values listed

above are registered by this document; additional values are managed by the IANA registry (see "Checksum Algorithm Registry" in the IANA Considerations section). CHECKSUM_ALG_NONE indicates the deployment relies on transport-layer (TLS, IPsec) or storage-layer integrity instead of a protocol-level per-chunk checksum.

cs_value: the checksum bytes. The length is fixed per registered algorithm:

* CHECKSUM_ALG_NONE: 0 bytes.

* CHECKSUM_ALG_CRC32: 4 bytes.

* CHECKSUM_ALG_CRC32C: 4 bytes.

* CHECKSUM_ALG_FLETCHER4: 32 bytes (four 64-bit accumulators, matching the ZFS Fletcher4 layout).

* CHECKSUM_ALG_SHA256: 32 bytes.

* CHECKSUM_ALG_SHA512: 64 bytes.

* CHECKSUM_ALG_BLAKE3: 32 bytes (BLAKE3 standard output length).

A checksum4 whose cs_value length does not match the registered length for cs_algorithm MUST be rejected with NFS4ERR_INVAL.

The checksum algorithm for a given file is selected by the metadata server at LAYOUTGET time and carried in the layout (see Section 8.8). A client that does not implement the algorithm a layout names returns the layout with NFS4ERR_LAYOUT_CHECKSUM_NOT_SUPPORTED (Section 21.1.6); the metadata server may then offer a layout with a different algorithm.

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       = 89,
///  OP_REVOKE_STATEID      = 90,
///  OP_BULK_REVOKE_STATEID = 91,
///
```

Figure 54: 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 55: 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 56: nfs_resop4 amendment block

Operations 78 through 88 (the CHUNK_* operations) are sent by clients to storage devices on the data path. Operations 89 through 91 (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.

All CHUNK_* operations MUST be issued under an active flexible file v2 layout obtained via LAYOUTGET against the metadata server. A data server receiving a CHUNK_* operation from a client that does not hold a current layout stateid for the target file MUST reject the operation with NFS4ERR_BAD_STATEID. In trusted-stateid tight coupling, the stateid presented MUST be present in the data server's trust table; an unknown stateid MUST be rejected with NFS4ERR_BAD_STATEID per Section 25.12.

The chunk envelope's safety properties (atomicity via chunk_guard4 CAS, integrity via checksum, lock continuity across revocation) depend on metadata-server coordination of layout grants, guard generation, and lock escrow. A client that issues CHUNK_* operations outside an active layout is operating outside this specification; the data server's behaviour in that case is undefined. See Section 12.2 for the distinction between the CHUNK_* surface and a generic block I/O interface.

Operation	Number	Target Server	Description
CHUNK_COMMIT	78	data server (client)	Section 25.1
CHUNK_ERROR	79	data server (client)	Section 25.2
CHUNK_FINALIZE	80	data server (client)	Section 25.3
CHUNK_HEADER_READ	81	data server (client)	Section 25.4
CHUNK_LOCK	82	data server (client)	Section 25.5
CHUNK_READ	83	data server (client)	Section 25.6
CHUNK_REPAIRED	84	data server (client)	Section 25.7

CHUNK_ROLLBACK	85	data server (client)	Section 25.8
CHUNK_UNLOCK	86	data server (client)	Section 25.9
CHUNK_WRITE	87	data server (client)	Section 25.10
CHUNK_WRITE_REPAIR	88	data server (client)	Section 25.11
TRUST_STATEID	89	data server (metadata server control)	Section 25.12
REVOKE_STATEID	90	data server (metadata server control)	Section 25.13
BULK_REVOKE_STATEID	91	data server (metadata server 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 57: XDR for CHUNK_COMMIT4args

25.1.2. RESULTS

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

Figure 58: XDR for CHUNK_COMMIT4resok

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

Figure 59: XDR for CHUNK_COMMIT4res

25.1.3. DESCRIPTION

The CHUNK_COMMIT operation is based upon the NFSv4.1 COMMIT operation (see Section 18.3 of [RFC8881]) and similarly commits previously written data to stable storage on the regular file identified by the current filehandle, with the difference that CHUNK_COMMIT operates on the chunk coordinate system used by Flexible File Version 2 layouts rather than on the byte coordinate system, and that CHUNK_COMMIT advances each named chunk through the chunk state machine from FINALIZED to COMMITTED (Figure 32) rather than acting on a byte range without a state-machine context.

The client provides cca_offset and cca_count to bound the chunk range, and cca_chunks to name the specific (chunk_owner4) generations within that range to commit:

cca_offset: starting chunk index in the file (not a byte offset).

cca_count: number of chunks the range covers, starting at cca_offset. A zero cca_count, or a cca_offset beyond the data server's highest chunk, is not an error; the data server returns NFS4_OK with an empty ccr_status array.

cca_chunks: an array of chunk_owner4 entries (Figure 52) naming the specific (cg_gen_id, cg_client_id, co_id) generations to commit. Each entry's co_id MUST fall within [cca_offset, cca_offset + cca_count); an entry whose co_id is outside the range is rejected with NFS4ERR_INVAL in the corresponding ccr_status slot. The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear as the cg_client_id of any cca_chunks entry; see Section 24.1.3 and Section 24.1.4.

cca_offset and cca_count would appear redundant given cca_chunks contains explicit co_id values, but they exist because a chunk index MAY have multiple persisted generations at the moment CHUNK_COMMIT arrives -- an older COMMITTED generation retained for the rollback invariant (Section 12.6) alongside a newer FINALIZED successor. cca_chunks selects which (cg_gen_id, cg_client_id) generation to advance to COMMITTED; cca_offset and cca_count bound the work scope so the data server can reject malformed requests that name chunks outside the intended commit window.

The CHUNK_COMMIT result reports the outcome per chunk in the same order as cca_chunks:

ccr_writeverf: a verifier identifying the data server's incarnation at the time the commit completed. A client compares ccr_writeverf to the cwr_writeverf returned by the prior CHUNK_WRITE (Section 25.10) to detect a data server restart that lost UNSTABLE4 writes between the write and the commit; on a mismatch the client MUST re-issue the CHUNK_WRITE before any committed bytes are considered durable. ccr_writeverf changes on every data server restart that loses uncommitted state.

ccr_status: per-chunk commit status, one entry per cca_chunks entry, co-indexed. NFS4_OK indicates that the named chunk is COMMITTED on return. Other per-entry failure codes are described in "Interaction with CHUNK_FINALIZE" and "Interaction with a Locked Chunk" below. The top-level CHUNK_COMMIT status is NFS4_OK as long as the data server could evaluate each cca_chunks entry; per-chunk failures are reported in ccr_status rather than by failing the whole operation. The top-level status returns a non-OK code only when the request could not be evaluated at all (for example, NFS4ERR_BADXDR, NFS4ERR_SERVERFAULT).

Unlike CHUNK_READ (Section 25.6) and CHUNK_WRITE (Section 25.10), CHUNK_COMMIT has no explicit stateid field in its arguments. The data server authorizes CHUNK_COMMIT against the stateid context the compound has already established, typically the stateid carried on an immediately preceding PUTFH or an earlier CHUNK_* operation in the same compound. Under trusted-stateid tight coupling (Section 25.12), the data server applies the trust-table check to whichever layout stateid the compound has presented; if no layout stateid has been presented or the presented stateid is not in the trust table, the data server rejects CHUNK_COMMIT with NFS4ERR_BAD_STATEID.

If the current filehandle is not an ordinary file, an error MUST be returned. If the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. If the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases of non-regular-file filehandles, NFS4ERR_WRONG_TYPE is returned.

25.1.3.1. Interaction with CHUNK_FINALIZE

CHUNK_COMMIT transitions a chunk from FINALIZED to COMMITTED (see Section 12.5). 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_ATOMIC 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_ATOMIC 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).

25.1.3.2. Pipelining Considerations

The three-step CHUNK_WRITE -> CHUNK_FINALIZE -> CHUNK_COMMIT sequence MAY be pipelined within a single NFSv4.2 compound (see Section 12.8 of [RFC8881]) in single-writer mode, where no other writer can race the client's per-chunk transitions and the CHUNK_WRITE per-block status array reports only local-failure cases (NFS4ERR_NOSPC, NFS4ERR_IO, and so on).

Same-compound pipelining is NOT RECOMMENDED in multiple-writer mode. CHUNK_WRITE reports per-block outcomes in cwr_status (Section 25.10); a partial-success outcome (some chunks accepted, others rejected with NFS4ERR_CHUNK_GUARDED on a lost race) leaves the client without an opportunity to react before a same-compound CHUNK_FINALIZE /

CHUNK_COMMIT proceeds against whichever chunks happen to be PENDING. The compound-level status is NFS4_OK in this case because per-block failures are reported in the per-op status array rather than as a compound- level error, so NFSv4 compound short-circuit (Section 2.10.6.4 of [RFC8881]) does not stop the trailing ops. A client that wants atomic-or-none semantics across multiple chunks MUST examine the per-block status returned by each CHUNK_WRITE before issuing the corresponding CHUNK_FINALIZE.

For multi-chunk pipelines in multiple-writer mode, the recommended pattern is to stagger the three steps across compounds so each trailing operation acts only on chunks whose preceding operation's status the client has already inspected:

```
Compound A:  SEQUENCE PUTFH CHUNK_WRITE(a)
Compound B:  SEQUENCE PUTFH CHUNK_WRITE(b) CHUNK_FINALIZE(a)
Compound C:  SEQUENCE PUTFH CHUNK_WRITE(c) CHUNK_FINALIZE(b)
                            CHUNK_COMMIT(a)
Compound D:  SEQUENCE PUTFH CHUNK_WRITE(d) CHUNK_FINALIZE(c)
                            CHUNK_COMMIT(b)
...
```

Figure 60: Staggered three-stage chunk pipeline (multiple-writer mode)

In each compound, the CHUNK_WRITE acts on the trailing chunk the client wants to enqueue next; the CHUNK_FINALIZE operates on a chunk whose CHUNK_WRITE the client has already inspected in a previous compound; the CHUNK_COMMIT operates on a chunk whose CHUNK_FINALIZE the client has already inspected. If any per-block status in compound N reports a guard loss or other failure, the client abandons the affected chunk (via CHUNK_ROLLBACK in compound N+1 or later) without ever issuing the trailing FINALIZE / COMMIT for it.

This pattern adds two compounds of latency between a chunk's write and its commit (one for the FINALIZE wait, one for the COMMIT wait), but provides the client with the per-step inspection point required for atomic-or-none multi-chunk writes under contention.

25.1.3.3. 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 24.1.4) 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 12.6 calls "lock continuity across revocation": the COMMIT privilege follows the lock without gaps in which a non-owner could race.

25.1.4. RESPONSE CODES

NFS4_OK: every named chunk transitioned to COMMITTED.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to commit on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_DELAY: the data server is temporarily unable to process the request.

NFS4ERR_FHEXPIRED: the current filehandle has expired.

NFS4ERR_INVAL: arguments named chunks outside the file's mirror set or in a non-atomic state.

NFS4ERR_IO: an I/O error occurred while persisting the commit.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_COMMIT.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

NFS4ERR_STALE: the current filehandle no longer identifies a valid file.

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 61: XDR for CHUNK_ERROR4args

25.2.2. RESULTS

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

Figure 62: XDR for CHUNK_ERROR4res

25.2.3. DESCRIPTION

CHUNK_ERROR allows a client that has detected corruption or inconsistency in a chunk to report the condition to the data server, so that the data server can mark the affected chunks as errored. Errored chunks are excluded from subsequent CHUNK_READ responses until they are repaired via CHUNK_WRITE_REPAIR (Section 25.11) and the repair is confirmed via CHUNK_REPAIRED (Section 25.7).

CHUNK_ERROR has no direct analog in [RFC8881]. The closest parallel is LAYOUTERROR ([RFC7862] Section 15.6), which reports layout-level errors to the metadata server. CHUNK_ERROR is the data-path counterpart: it reports a chunk-level integrity finding directly to the data server so that the corrupted chunks are quarantined before the metadata server has had time to coordinate repair. A client SHOULD issue CHUNK_ERROR to the data server holding the bad chunks before issuing LAYOUTERROR to the metadata server.

The client provides:

cea_stateid: the layout stateid the metadata server granted for this file. Under trusted-stateid tight coupling (Section 25.12), this stateid MUST be in the data server's trust table; otherwise the data server rejects the operation with NFS4ERR_BAD_STATEID.

cea_offset: starting chunk index of the affected range (not a byte offset).

cea_count: number of chunks the affected range covers, starting at cea_offset.

cea_error: the nfsstat4 error code that describes the integrity finding. Typical values include NFS4ERR_PAYLOAD_NOT_ATOMIC (the chunk's persisted checksum or guard did not match the value the client expected), NFS4ERR_IO (the client's CHUNK_READ returned an I/O error from this data server), and NFS4ERR_INVAL (the chunk's chunk_owner4 did not match the expected generation across mirrors). The data server MAY record the supplied error code in operator logs but does not otherwise interpret it; the chunk-level effect (mark errored) is the same for any cea_error value.

cea_owner: the chunk_owner4 (Figure 52) the client read when it observed the error, so the data server can record which (cg_gen_id, cg_client_id) generation was reported as corrupted. The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear in cea_owner; see Section 24.1.3 and Section 24.1.4.

CHUNK_ERROR returns a single top-level status in cer_status; there is no per-chunk status array because the data server either accepts the report for the whole range or returns a top-level error. Once a CHUNK_ERROR has been accepted, the affected chunks transition into the errored state described in Section 12.5; subsequent CHUNK_READ operations against those chunks return NFS4ERR_PAYLOAD_NOT_ATOMIC in the per-chunk cr_status slot until a successful CHUNK_REPAIRED sequence clears the errored flag.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

25.2.4. RESPONSE CODES

NFS4_OK: the client's chunk error report has been recorded.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to report errors on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_INVAL: the reported chunk range or error code was not recognized.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_ERROR.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

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 63: XDR for CHUNK_FINALIZE4args

25.3.2. RESULTS

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

Figure 64: XDR for CHUNK_FINALIZE4resok

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

Figure 65: XDR for CHUNK_FINALIZE4res

25.3.3. DESCRIPTION

CHUNK_FINALIZE transitions chunks from the PENDING state (set by CHUNK_WRITE, see Section 25.10) to the FINALIZED state in the chunk state machine (Figure 32). A FINALIZED chunk is visible on the owning stateid for reads (Section 12.6) and is eligible for CHUNK_COMMIT (Section 25.1); the FINALIZED transition is the writer's signal that it will issue no further CHUNK_WRITEs for the named (cg_gen_id, cg_client_id) generation of each chunk.

CHUNK_FINALIZE has no direct analog in [RFC8881]: the COMMIT operation in [RFC8881] Section 18.3 combines the "no more writes" signal and the "make durable and globally visible" step into one operation; the Flexible File Version 2 chunk lifecycle separates them so a writer in multiple-writer mode can validate the per-chunk acceptance status reported by CHUNK_WRITE before committing any chunk to durable storage (see "Pipelining Considerations" in Section 25.1).

The client provides cfa_offset and cfa_count to bound the chunk range, and cfa_chunks to name the specific (chunk_owner4) generations within that range to finalize:

cfa_offset: starting chunk index in the file (not a byte offset).

cfa_count: number of chunks the range covers, starting at cfa_offset. A zero cfa_count, or a cfa_offset beyond the data server's highest chunk, is not an error; the data server returns NFS4_OK with an empty cfr_status array.

cfa_chunks: an array of chunk_owner4 entries (Figure 52) naming the specific (cg_gen_id, cg_client_id, co_id) generations to finalize. Each entry's co_id MUST fall within [cfa_offset, cfa_offset + cfa_count); an entry whose co_id is outside the range is rejected with NFS4ERR_INVAL in the corresponding cfr_status slot. The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear as the cg_client_id of any cfa_chunks entry; see Section 24.1.3 and Section 24.1.4.

The CHUNK_FINALIZE result reports the outcome per chunk in the same order as cfa_chunks:

cfr_writeverf: a verifier identifying the data server's incarnation at the time the finalization completed. Semantics match cwr_writeverf in CHUNK_WRITE (Section 25.10): a client that observes a different writeverf on a subsequent CHUNK_COMMIT MUST re-issue the CHUNK_WRITE before treating any of the finalized chunks as durable.

cfr_status: per-chunk finalization status, one entry per cfa_chunks entry, co-indexed. NFS4_OK indicates that the named chunk is FINALIZED on return. Other per-entry failure cases:

* NFS4ERR_INVAL -- the named generation is not in the PENDING state at this offset (the chunk is EMPTY, FINALIZED at a different generation, or COMMITTED), or the entry's co_id is outside the [cfa_offset, cfa_offset + cfa_count) range.

* NFS4ERR_CHUNK_GUARDED -- the chunk is PENDING but at a different (cg_gen_id, cg_client_id) than the one named in the cfa_chunks entry. A client that sees this has lost a race; see Section 24.1.

* NFS4ERR_CHUNK_LOCKED -- the chunk is locked by a CHUNK_LOCK (Section 25.5) held by a different stateid; the finalize is rejected.

The top-level CHUNK_FINALIZE status is NFS4_OK as long as the data server could evaluate each cfa_chunks entry; per-chunk failures are reported in cfr_status rather than by failing the whole operation. The top-level status returns a non-OK code only when the request could not be evaluated at all (for example, NFS4ERR_BADXDR, NFS4ERR_SERVERFAULT).

CHUNK_FINALIZE serves as the CRC validation checkpoint for the chunk lifecycle. The data server SHOULD have validated each chunk's checksum against the value supplied in cwa_checksums at CHUNK_WRITE time; the FINALIZE transition persists the chunk metadata (CRC, owner, state) to stable storage so it survives a data server restart. An implementation MAY defer some metadata persistence to CHUNK_COMMIT instead of CHUNK_FINALIZE; in that case the FINALIZED state is recovered by replay of the data server's local journal on restart.

A chunk that has been FINALIZED but not yet COMMITTED MAY be rolled back via CHUNK_ROLLBACK (Section 25.8), which returns the chunk to the EMPTY state (or to the prior COMMITTED generation, if one exists).

Like CHUNK_COMMIT, CHUNK_FINALIZE has no explicit stateid field in its arguments. The data server authorizes CHUNK_FINALIZE against the stateid context the compound has already established, typically the stateid carried on an immediately preceding PUTFH or an earlier CHUNK_* operation in the same compound. Under trusted-stateid tight coupling (Section 25.12), the data server applies the trust-table check to whichever layout stateid the compound has presented; if no layout stateid has been presented or the presented stateid is not in the trust table, the data server rejects CHUNK_FINALIZE with NFS4ERR_BAD_STATEID.

If the current filehandle is not an ordinary file, an error MUST be returned. If the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. If the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases of non-regular-file filehandles, NFS4ERR_WRONG_TYPE is returned.

25.3.4. RESPONSE CODES

NFS4_OK: every named chunk transitioned from PENDING to FINALIZED.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to finalize on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_DELAY: the data server is temporarily unable to process the request.

NFS4ERR_FHEXPIRED: the current filehandle has expired.

NFS4ERR_INVAL: arguments named chunks not in PENDING or outside the file's mirror set.

NFS4ERR_IO: an I/O error occurred while persisting the transition.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_FINALIZE.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

NFS4ERR_STALE: the current filehandle no longer identifies a valid file.

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 66: 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 67: XDR for CHUNK_HEADER_READ4resok

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

Figure 68: XDR for CHUNK_HEADER_READ4resok

25.4.3. DESCRIPTION

CHUNK_HEADER_READ returns the per-chunk metadata (chunk_owner4, lock state, and per-chunk status) for a range of chunks in the target data file without returning the chunk payloads. The operation enables clients and repair coordinators to inspect chunk lifecycle and ownership cheaply, without the data-transfer cost of CHUNK_READ (Section 25.6). CHUNK_HEADER_READ has no direct analog in [RFC8881]; it is the chunk-protocol counterpart of a stat-like fast probe and exists because chunks are first-class state-bearing objects whose ownership, lock state, and lifecycle status are not recoverable from a byte-offset query.

The client provides:

chra_stateid: the layout stateid the metadata server granted for this file. Under trusted-stateid tight coupling (Section 25.12), this stateid MUST be in the data server’s trust table; otherwise the data server rejects the operation with NFS4ERR_BAD_STATEID.

chra_offset: starting chunk index of the range to inspect (not a byte offset).

chra_count: number of chunks the inspection range covers, starting at chra_offset.

The CHUNK_HEADER_READ result returns four co-indexed arrays, one entry per chunk in the requested range in chunk-offset order from chra_offset:

chrr_eof: TRUE if the requested range extended at or past the data server's last chunk for this file. Same per-data-server semantics as crr_eof in CHUNK_READ (Section 25.6).

chrr_status: per-chunk lifecycle state encoded as an nfsstat4 (see "Per-Chunk Status Encoding" below).

chrr_locked: per-chunk boolean. TRUE if the chunk currently has a CHUNK_LOCK (Section 25.5) held by some chunk_owner4; FALSE otherwise. Lock state is reported orthogonally to chrr_status so that a locked chunk still surfaces its lifecycle state and chunk_owner4 to the inspector.

chrr_chunks: per-chunk chunk_owner4 (Figure 52). For a chunk whose chrr_status is NFS4_OK the field is the COMMITTED generation's owner. For NFS4ERR_PAYLOAD_NOT_ATOMIC the field is the writer of the in-progress (PENDING or FINALIZED) generation. For NFS4ERR_NOENT (EMPTY chunk) the chunk_owner4 is unspecified.

The operation has several uses:

Whole-file repair scan: A repair client selected via CB_CHUNK_REPAIR (Section 26.1) walks the affected chunk range and uses the per-chunk chunk_owner4 returned by each mirror's data server to identify which chunks carry an atomic stripe (all k data shards share the same chunk_guard4) and which require reconstruction. CHUNK_HEADER_READ is the discovery primitive that drives the per-chunk decisions described in Section 11.2.6.5; without it, a repair client would have to issue CHUNK_READ to retrieve the full payload of every chunk merely to inspect its guard.

Client-side recovery from partial writes: After a network disruption or client restart, a writer that holds the file's layout MAY issue CHUNK_HEADER_READ to learn which of its prior CHUNK_WRITEs reached the data server. Chunks whose chunk_owner4 reports the writer's own (cg_client_id, cg_gen_id) pair are PENDING or FINALIZED and recoverable; chunks absent from the response or carrying another writer's owner are not. The writer can then re-issue CHUNK_WRITE for the missing chunks or CHUNK_ROLLBACK for the abandoned ones without reading payloads it has already committed locally.

Read-side atomicity check: Before issuing a multi-chunk CHUNK_READ in multiple-writer mode, a client MAY issue CHUNK_HEADER_READ to verify that the chunks in the target range share a common chunk_guard4 (the cohort-atomicity property in Section 12.6). If the guards diverge, the client knows the read will not be atomic and can wait for a writer to commit, retry, or report NFS4ERR_PAYLOAD_NOT_ATOMIC via LAYOUTERROR. This is a hint rather

than a guarantee: a concurrent writer MAY advance a chunk's state between the CHUNK_HEADER_READ response and the subsequent CHUNK_READ.

Lock probe before write: A client MAY issue CHUNK_HEADER_READ and inspect the chrr_locked array to discover whether any chunk in the target range is currently held by a CHUNK_LOCK (Section 25.5) before attempting CHUNK_WRITE, avoiding the round-trip cost of receiving NFS4ERR_CHUNK_LOCKED. As above, this is a hint; a lock MAY be acquired between the header read and the write.

CHUNK_HEADER_READ does not change any chunk state.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

25.4.3.1. Per-Chunk Status Encoding

The per-chunk chrr_status field reports the chunk's lifecycle state encoded as an nfsstat4:

NFS4_OK: the chunk is COMMITTED and the chunk_owner4 in the corresponding chrr_chunks slot is the COMMITTED generation's owner.

NFS4ERR_PAYLOAD_NOT_ATOMIC: the chunk is PENDING or FINALIZED (a non-globally- visible generation is in progress). The chunk_owner4 in the corresponding chrr_chunks slot names the writer of that in-progress generation.

NFS4ERR_NOENT: the chunk is EMPTY (no COMMITTED generation has been written at this offset). The chunk_owner4 in the corresponding chrr_chunks slot is unspecified.

CHUNK_HEADER_READ never returns NFS4ERR_CHUNK_LOCKED in chrr_status; lock state is reported orthogonally via chrr_locked so that locked chunks still surface their chunk_owner4 to the inspector.

25.4.4. RESPONSE CODES

NFS4_OK: the chunk headers have been returned.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to read chunk headers on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in

trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_DELAY: the data server is temporarily unable to process the request.

NFS4ERR_FHEXPIRED: the current filehandle has expired.

NFS4ERR_IO: an I/O error occurred while reading chunk headers.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_HEADER_READ.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

NFS4ERR_STALE: the current filehandle no longer identifies a valid file.

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 69: 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 70: XDR for CHUNK_LOCK4res

25.5.3. DESCRIPTION

CHUNK_LOCK acquires an exclusive chunk-range lock on the range specified by cla_offset and cla_count. While the lock is held, CHUNK_WRITE, CHUNK_WRITE_REPAIR, CHUNK_FINALIZE, CHUNK_COMMIT, CHUNK_ROLLBACK, and CHUNK_UNLOCK (Section 25.9) operations on any of the locked chunks from any other chunk_owner4 receive NFS4ERR_CHUNK_LOCKED in the corresponding per-chunk status slot. The lock is associated with the chunk_owner4 in cla_owner.

CHUNK_LOCK is loosely analogous to LOCK ([RFC8881] Section 18.10) in that it acquires an exclusive guard against concurrent modification, but the two operate on different coordinate systems and use different naming: LOCK is byte-range and stateid-based; CHUNK_LOCK is chunk-range and chunk_owner4-based. CHUNK_LOCK is used in multiple-writer mode (Section 11.2.6.4) to serialize racing writers on a common chunk range, and in the repair flow (Section 11.2.4) to transfer lock ownership to a repair client via CHUNK_LOCK_FLAGS_ADOPT.

The client provides:

cla_stateid: the layout stateid the metadata server granted for this file. Under trusted-stateid tight coupling (Section 25.12), this stateid MUST be in the data server’s trust table; otherwise the data server rejects the operation with NFS4ERR_BAD_STATEID.

cla_offset: starting chunk index of the lock range (not a byte offset).

cla_count: number of chunks the lock range covers, starting at cla_offset.

cla_flags: bitmask of CHUNK_LOCK_FLAGS_* values. Currently defined: CHUNK_LOCK_FLAGS_ADOPT (lock-ownership transfer; see "Lock Transfer via CHUNK_LOCK_FLAGS_ADOPT" below). Unknown bits MUST be rejected with NFS4ERR_INVAL.

cla_owner: the chunk_owner4 (Figure 52) that will hold the lock on success. The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear as the cg_client_id of cla_owner; see Section 24.1.3 and Section 24.1.4. (A client requesting CHUNK_LOCK_FLAGS_ADOPT MUST use its own cg_client_id, not the MDS-escrow sentinel, even when adopting from an MDS-escrow holder.)

The CHUNK_LOCK result returns:

clr_status: NFS4_OK if the lock was acquired (or transferred via

ADOPT). NFS4ERR_CHUNK_LOCKED if one or more chunks in the range are already locked and the request does not carry CHUNK_LOCK_FLAGS_ADOPT.

clr_owner (NFS4ERR_CHUNK_LOCKED case only): the chunk_owner4 of the current lock holder, so the caller can identify the blocking writer.

The lock is released by CHUNK_UNLOCK (Section 25.9) or implicitly when the holder's lease expires; on lease expiry without explicit release, the data server transitions the lock to the MDS-escrow owner if the metadata server has revoked the holder's stateid via REVOKE_STATEID (Section 25.13), per the lock-continuity- across-revocation invariant in Section 12.6.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

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 24.1.4). 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.

25.5.4. RESPONSE CODES

NFS4_OK: the requested chunk range has been locked.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to lock chunks on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_CHUNK_LOCKED: one or more chunks in the requested range are already locked by another writer.

NFS4ERR_INVAL: the requested range was malformed or outside the file's mirror set.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_LOCK.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

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 71: XDR for CHUNK_READ4args

25.6.2. RESULTS

```
/// struct read_chunk4 {
///     checksum4       cr_checksum;
///     uint32_t        cr_effective_len;
///     chunk_owner4    cr_owner;
///     uint32_t        cr_payload_id;
///     bool            cr_locked;
///     nfsstat4        cr_status;
///     opaque          cr_chunk<>;
/// };
```

Figure 72: XDR for read_chunk4

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

Figure 73: XDR for CHUNK_READ4resok

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

Figure 74: XDR for CHUNK_READ4res

25.6.3. DESCRIPTION

The CHUNK_READ operation is based upon the NFSv4.1 READ operation (see Section 18.22 of [RFC8881]) and similarly reads data from the regular file identified by the current filehandle, with the difference that CHUNK_READ operates on the chunk coordinate system used by Flexible File Version 2 layouts rather than on the byte coordinate system.

The client provides a cra_offset of where the CHUNK_READ is to start and a cra_count of how many chunks are to be read. cra_offset is the starting chunk index in the file (not a byte offset); the chunk at index N occupies the bytes [N * chunk_size, (N + 1) * chunk_size) for codecs with a uniform chunk size, where chunk_size is taken from ffv2m_striping_unit_size in the file's layout (Section 8.8). For codecs whose parity shards have variable sizes (the Mojette family), the parity-shard chunks on a given data server may use a smaller per-shard chunk size; see Section 11.5. cra_count is a count of chunks to read and not bytes to read.

A cra_offset of zero starts reading at the first chunk of the file. If cra_offset is greater than or equal to the number of chunks the data server holds for this file, the status NFS4_OK is returned with crr_chunks empty and crr_eof set to TRUE.

If cra_count is zero, the CHUNK_READ succeeds and returns zero chunks. In all situations the data server MAY choose to return fewer chunks than the client requested; the client must be prepared to handle a short read and reissue CHUNK_READ for the remaining chunks.

The CHUNK_READ result is comprised of an array of read_chunk4, each describing the metadata and payload of one chunk. The array entries are in chunk-index order starting from cra_offset. Within each read_chunk4 (Figure 72):

cr_checksum: the checksum4 (Section 24.3) the data server computed over the chunk payload (cr_chunk) at CHUNK_FINALIZE or CHUNK_COMMIT time and persisted with the chunk metadata. The cs_algorithm field matches the layout's ffv2m_checksum_algorithm (Section 8.8); the cs_value carries the computed bytes at the length registered for that algorithm. The client uses cr_checksum to detect transport corruption between the data server and the client; see Section 27.1 for the scope and limits of checksum protection per algorithm class.

cr_effective_len: the byte length of cr_chunk. This may be smaller

than the layout's chunk_size when the chunk is the final chunk of a file whose size is not chunk-aligned, or when the chunk belongs to a variable-size Mojette parity shard.

cr_owner: the chunk_owner4 carrying the chunk_guard4 and chunk-id of the COMMITTED generation being returned. A client reading from multiple data servers in an erasure-coded layout MUST compare cr_owner.co_guard across data servers; agreement of the chunk_guard4 across the k data shards is the atomicity invariant on which reconstruction depends. See Section 12.6.

cr_payload_id: the payload-id the writer associated with the chunk at CHUNK_WRITE time, used by repair coordinators to correlate chunks across mirrors.

cr_locked: TRUE if the chunk currently has a CHUNK_LOCK (Section 25.5) held against it; FALSE otherwise. Lock state does not block the read.

cr_status: per-chunk status. NFS4_OK indicates that cr_chunk is the COMMITTED payload. NFS4ERR_PAYLOAD_NOT_ATOMIC indicates the chunk's persisted checksum or guard check failed at read time, in which case cr_chunk content is undefined; see Section 21.1.2. NFS4ERR_NOENT indicates the chunk is EMPTY (no COMMITTED generation has been written at this offset).

cr_chunk: the chunk payload bytes. Empty for cr_status values other than NFS4_OK.

A chunk that is EMPTY at the requested offset is returned as a synthetic zero-filled chunk: cr_status is NFS4ERR_NOENT, cr_chunk is zero-filled to the layout's chunk_size, cr_owner is set to all-zeros (with cg_client_id = CHUNK_GUARD_CLIENT_ID_NONE, see Section 24.1.3), and cr_checksum is the checksum of the synthetic zero-filled payload. This lets a client reconstruct holes without a special-casing path.

The data server MAY signal end-of-file by setting crr_eof to TRUE. If the CHUNK_READ ended at the last chunk that exists on this data server (the read returned chunks up to and including the data server's last chunk) or extended beyond it, crr_eof MUST be TRUE. Otherwise crr_eof is FALSE. A successful CHUNK_READ of an empty file always returns crr_eof as TRUE with crr_chunks empty. Note that crr_eof reflects the state at the data server only; in a multi-data-server erasure-coded layout the file's logical size is reconstructed at the client from the surviving shards' chunk_owner4 values, not from any single data server's crr_eof.

Except when special stateids are used, the cra_stateid value represents a layout stateid returned by a prior LAYOUTGET against the metadata server (see Section 18.43 of [RFC8881]). The data server uses cra_stateid to verify that the client holds a valid layout that authorizes reading this file. Under trusted-stateid tight coupling (Section 25.12), the data server additionally checks that the metadata server has registered the stateid via TRUST_STATEID; an unregistered stateid (other than a special stateid) returns NFS4ERR_BAD_STATEID.

For a CHUNK_READ with a cra_stateid value of all bits equal to zero, the data server MAY allow the CHUNK_READ to be serviced subject to the chunk-lock state recorded in cr_locked. For a CHUNK_READ with a cra_stateid value of all bits equal to one, the data server MAY allow CHUNK_READ to bypass lock-state reporting at the data server. These special-stateid behaviours mirror the corresponding READ semantics in [RFC8881] adapted to the chunk-locking model (Section 25.5) rather than the byte-range locking model of [RFC8881] Section 12.

If the current filehandle is not an ordinary file, an error MUST be returned. If the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. If the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases of non-regular-file filehandles, NFS4ERR_WRONG_TYPE is returned.

Figure 75 shows a client requesting 4 chunks starting at chunk index 2. Data Server 2 responds as in Figure 76: there is valid data for chunks 2 and 4, a synthetic zero-filled hole at chunk 3, and no data for chunk 5 (the data server’s last chunk is chunk 4, so crr_eof is TRUE). The data server calculates a valid cr_checksum for chunk 3 based on the synthetic zero-filled payload.

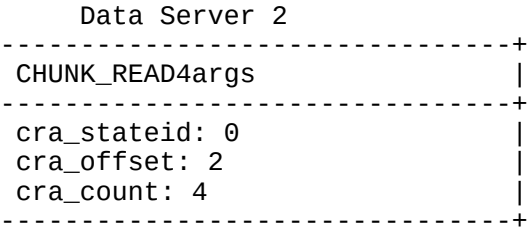


Figure 75: Example: CHUNK_READ4args parameters

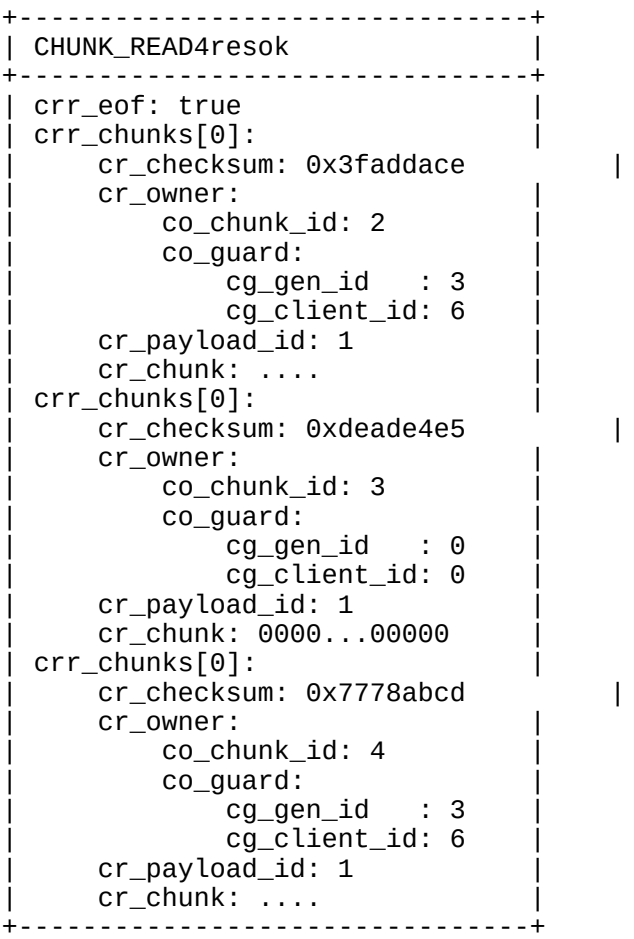


Figure 76: Example: Resulting CHUNK_READ4resok reply

25.6.4. RESPONSE CODES

NFS4_OK: the requested chunks have been returned.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to read this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_DELAY: the data server is temporarily unable to process the request.

NFS4ERR_FHEXPIRED: the current filehandle has expired.

NFS4ERR_IO: an I/O error occurred while reading the chunks.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_READ.

NFS4ERR_PAYLOAD_NOT_ATOMIC: one or more chunks failed their persisted guard or CRC check. See Section 21.1.2.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

NFS4ERR_STALE: the current filehandle no longer identifies a valid file.

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 77: XDR for CHUNK_REPAIRED4args

25.7.2. RESULTS

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

Figure 78: XDR for CHUNK_REPAIRED4res

25.7.3. DESCRIPTION

CHUNK_REPAIRED signals that chunks previously marked as errored (via CHUNK_ERROR, see Section 25.2) have been repaired and the errored state can be cleared. The repair client writes replacement data via CHUNK_WRITE_REPAIR (Section 25.11), advances the new chunks through CHUNK_FINALIZE (Section 25.3) and CHUNK_COMMIT (Section 25.1), and only then issues CHUNK_REPAIRED to make the repaired chunks visible to normal CHUNK_READ traffic again.

CHUNK_REPAIRED has no direct analog in [RFC8881]; it is the chunk-protocol equivalent of clearing a "needs scrub" flag after a RAID controller has rewritten a parity stripe. Together with CHUNK_ERROR it forms the data-server-side state-bit pair that quarantines damaged chunks from ordinary reads during the repair window.

The client provides:

cra_stateid: the layout stateid the metadata server granted to the repair client. Under trusted-stateid tight coupling (Section 25.12), this stateid MUST be in the data server's trust table; otherwise the data server rejects the operation with NFS4ERR_BAD_STATEID.

cra_offset: starting chunk index of the repaired range (not a byte offset).

cra_count: number of chunks the repaired range covers, starting at cra_offset. The cra_offset / cra_count range MUST match the range named in the original CHUNK_ERROR that marked these chunks errored; mismatched ranges are rejected with NFS4ERR_INVAL.

cra_owner: the chunk_owner4 (Figure 52) identifying the repair client. The data server uses this to record which actor cleared the errored state. The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear in cra_owner; see Section 24.1.3 and Section 24.1.4.

CHUNK_REPAIRED returns a single top-level status; there is no per-chunk status array because the data server either accepts the confirmation for the whole range or returns a top-level error.

The data server MUST verify before accepting the confirmation that:

* every chunk in [cra_offset, cra_offset + cra_count) is currently in the errored state, and

* every chunk in the range has been advanced to COMMITTED by a CHUNK_WRITE_REPAIR / CHUNK_FINALIZE / CHUNK_COMMIT sequence since the CHUNK_ERROR that marked them errored.

If either precondition fails, the data server returns NFS4ERR_INVAL and the errored state is left in place. A repair client that sees NFS4ERR_INVAL SHOULD verify the chunks via CHUNK_HEADER_READ (Section 25.4) before retrying.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

25.7.4. RESPONSE CODES

NFS4_OK: the repair confirmation has been recorded.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to confirm repair on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_INVAL: the chunks named were not in an errored state, or the repair did not match the recorded error.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_REPAIRED.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

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 79: XDR for CHUNK_ROLLBACK4args

25.8.2. RESULTS

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

Figure 80: XDR for CHUNK_ROLLBACK4resok

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

Figure 81: XDR for CHUNK_ROLLBACK4res

25.8.3. DESCRIPTION

CHUNK_ROLLBACK reverts chunks from the PENDING or FINALIZED state to their previous state, effectively undoing a CHUNK_WRITE (Section 25.10) that has not yet reached COMMITTED via CHUNK_COMMIT (Section 25.1). The reversion target is the prior COMMITTED generation, if one exists for the affected chunk; otherwise the chunk returns to the EMPTY state (Figure 32). CHUNK_ROLLBACK against a chunk already in the COMMITTED state is permitted only on the repair path; see "Rollback of COMMITTED Chunks" below.

CHUNK_ROLLBACK has no direct analog in [RFC8881]: NFS WRITE has no separate finalization or commit step that a client could undo without contacting other components. CHUNK_ROLLBACK exists because the chunk state machine exposes the PENDING and FINALIZED states explicitly, and the writer needs a way to abandon a non-committed generation without committing it.

The client provides cra_offset and cra_count to bound the chunk range, and cra_chunks to name the specific (chunk_owner4) generations within that range to roll back:

cra_offset: starting chunk index in the file (not a byte offset).

cra_count: number of chunks the range covers, starting at cra_offset.

cra_chunks: an array of chunk_owner4 entries (Figure 52) naming the specific (cg_gen_id, cg_client_id, co_id) generations to roll back. Each entry's co_id MUST fall within [cra_offset, cra_offset + cra_count); entries outside the range are rejected with

NFS4ERR_INVAL in the corresponding crr_status slot (the result struct is sized to match cra_chunks). The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear as the cg_client_id of any cra_chunks entry; see Section 24.1.3 and Section 24.1.4.

The CHUNK_ROLLBACK result returns:

crr_writeverf: a verifier identifying the data server's incarnation. Semantics match cwr_writeverf in CHUNK_WRITE.

CHUNK_ROLLBACK has two principal scenarios:

1. A writer in multiple-writer mode that observed per-chunk failures in the CHUNK_WRITE response (e.g., NFS4ERR_CHUNK_GUARDED on a subset of chunks) needs to abandon the partial write before issuing CHUNK_FINALIZE on the chunks that did succeed. CHUNK_ROLLBACK on the abandoned chunks releases their PENDING generation cleanly.

2. A repair client that wrote reconstructed data via CHUNK_WRITE_REPAIR (Section 25.11) and subsequently discovered the reconstruction was wrong (for example, a CRC mismatch detected during cross-mirror verification) needs to abandon the repair before any client commits it.

The data server effects the rollback as follows:

* Chunks in PENDING with a matching chunk_owner4: the data server deletes the PENDING payload and restores the chunk to its prior state (EMPTY, or the prior COMMITTED generation if the rollback invariant in Section 12.6 required retention).

* Chunks in FINALIZED with a matching chunk_owner4: the data server deletes the FINALIZED payload and the persisted finalization metadata, restoring the chunk to its prior state.

* Chunks not in PENDING or FINALIZED at the named generation, or whose chunk_owner4 does not match: the corresponding crr_status slot reports NFS4ERR_INVAL and the chunk is left unchanged.

25.8.3.1. Rollback of COMMITTED Chunks

CHUNK_ROLLBACK against a COMMITTED chunk is permitted ONLY on the repair path, when a repair client is restoring a prior COMMITTED generation that another client incorrectly advanced. In this case the data server replaces the current COMMITTED generation with the chunk_owner4 named in the cra_chunks entry, which MUST itself name a generation already persisted at the data server (typically the prior COMMITTED kept under the rollback invariant). A non-repair CHUNK_ROLLBACK against a COMMITTED chunk is rejected with NFS4ERR_INVAL.

25.8.3.2. Stateid and Authorization

Like CHUNK_COMMIT, CHUNK_ROLLBACK has no explicit stateid field in its arguments. The data server authorizes CHUNK_ROLLBACK against the stateid context the compound has already established, typically the stateid carried on an immediately preceding PUTFH or an earlier CHUNK_* operation. Under trusted-stateid tight coupling (Section 25.12), the data server applies the trust-table check to whichever layout stateid the compound has presented; if no layout stateid has been presented or the presented stateid is not in the trust table, the data server rejects CHUNK_ROLLBACK with NFS4ERR_BAD_STATEID.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

25.8.4. RESPONSE CODES

NFS4_OK: the named chunks have been rolled back.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to roll back chunks on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_INVAL: arguments named chunks not eligible for rollback or outside the file’s mirror set.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_ROLLBACK.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

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 82: XDR for CHUNK_UNLOCK4args

25.9.2. RESULTS

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

Figure 83: XDR for CHUNK_UNLOCK4res

25.9.3. DESCRIPTION

CHUNK_UNLOCK releases the exclusive chunk-range lock previously acquired by CHUNK_LOCK (Section 25.5). CHUNK_UNLOCK is loosely analogous to LOCKU ([RFC8881] Section 18.12) in that it releases an exclusive guard, but it operates on chunk-range coordinates and is matched against the chunk_owner4 that acquired the lock rather than against an open / lock stateid.

The client provides:

cua_stateid: the layout stateid the metadata server granted for this file. Under trusted-stateid tight coupling (Section 25.12), this stateid MUST be in the data server’s trust table; otherwise the data server rejects the operation with NFS4ERR_BAD_STATEID.

cua_offset: starting chunk index of the range to unlock (not a byte offset).

cua_count: number of chunks the unlock range covers, starting at cua_offset. The range MUST match exactly the range of an outstanding CHUNK_LOCK held by cua_owner; partial-range unlock is not supported.

cua_owner: the chunk_owner4 (Figure 52) that holds the lock. The cg_client_id MUST match the chunk_owner4 that was supplied on the CHUNK_LOCK that acquired the lock (including the case of a lock transferred via CHUNK_LOCK_FLAGS_ADOPT, in which the adopter's chunk_owner4 is the current holder). The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear as the cg_client_id of cua_owner; see Section 24.1.3 and Section 24.1.4. In particular, a repair client releasing a lock it adopted from the MDS-escrow owner uses its own cg_client_id in cua_owner, not CHUNK_GUARD_CLIENT_ID_MDS.

The CHUNK_UNLOCK result returns a single top-level status; there is no per-chunk status array because the unlock either succeeds for the whole range or returns a top-level error.

CHUNK_UNLOCK is idempotent in the sense that releasing chunks that are not currently locked returns NFS4_OK without effect. Releasing chunks that are locked by a different cua_owner returns NFS4ERR_INVAL and leaves the lock in place.

A client SHOULD issue CHUNK_UNLOCK promptly after completing the write, write-repair, or commit sequence that the lock guarded. Locks not explicitly released are released implicitly when the holder's lease expires; if the metadata server has revoked the holder's stateid via REVOKE_STATEID (Section 25.13) before the lease lapses, the lock transitions to the MDS-escrow owner per the lock-continuity invariant in Section 12.6 rather than being released outright.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

25.9.4. RESPONSE CODES

NFS4_OK: the named chunks have been unlocked, or no lock was held (idempotent).

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to unlock chunks on this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_INVAL: arguments named chunks not in a locked state owned by this caller.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_UNLOCK.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

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 84: 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;
///     checksum4          cwa_checksums<>;
///     opaque             cwa_chunks<>;
/// };
```

Figure 85: 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 86: XDR for CHUNK_WRITE4resok

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

Figure 87: XDR for CHUNK_WRITE4res

25.10.3. DESCRIPTION

The CHUNK_WRITE operation is based upon the NFSv4.1 WRITE operation (see Section 18.32 of [RFC8881]) and similarly writes data to the regular file identified by the current filehandle, with the difference that CHUNK_WRITE operates on the chunk coordinate system used by Flexible File Version 2 layouts rather than on the byte coordinate system. Successful chunk writes initially enter the PENDING state in the chunk state machine (Figure 32); a subsequent CHUNK_FINALIZE (Section 25.3) and CHUNK_COMMIT (Section 25.1) (or the activation shortcut described below) progress them to COMMITTED.

The client provides a cwa_offset of where the CHUNK_WRITE is to start and a payload consisting of one or more chunks packed into the cwa_chunks opaque field. cwa_offset is the starting chunk index in the file (not a byte offset); each chunk occupies cwa_chunk_size bytes within cwa_chunks except the last, which MAY be shorter when the file size is not chunk-aligned or when the payload encodes a variable-size Mojette parity shard (Section 11.5). The number of chunks in the payload is ceil(len(cwa_chunks) / cwa_chunk_size).

cwa_owner (Figure 52) names the writer’s chunk_owner4: cg_gen_id is the writer’s per-chunk generation counter, cg_client_id is the writer’s metadata-server-assigned client identifier (the reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear in cwa_owner; see Section 24.1.3 and Section 24.1.4), and co_id is the chunk-index identifier of the first chunk in the payload (redundant with cwa_offset for a single-chunk write; the data server MUST treat them as the same value and MAY reject a mismatch with NFS4ERR_INVAL).

cwa_payload_id is a writer-chosen identifier that lets a repair coordinator correlate chunks of the same logical write across data servers.

cwa_checksums, when non-empty, MUST contain one checksum entry per chunk in the payload. Each entry’s cs_algorithm MUST match ffv2m_checksum_algorithm of the mirror named in the layout (see Section 8.8); a mismatch is rejected with NFS4ERR_INVAL. The data

server validates each chunk's checksum at CHUNK_WRITE time and rejects mismatched chunks with NFS4ERR_IO in the corresponding cwr_block_status slot. An empty cwa_checksums array (cwa_checksums_len == 0) indicates the client did not supply per-chunk checksums; the data server still computes and persists per-chunk checksums from the payload bytes for later integrity verification but cannot detect transport corruption at CHUNK_WRITE time without the client's reference values.

cwa_flags carries CHUNK_WRITE_FLAGS_ACTIVATE_IF_EMPTY (see "Stability and Activation" below).

cwa_guard (Figure 84) controls the chunk- guard CAS check (see "Guarding the Write" below).

A cwa_offset of zero starts writing at the first chunk of the file. Unlike READ in [RFC8881], a CHUNK_WRITE whose cwa_offset extends beyond the current end of the file is not an error: the data server extends the file's chunk store to cover the new chunks, with intervening offsets remaining EMPTY (Section 12.5) until they too are written. If the cwa_chunks payload is empty (zero bytes), the CHUNK_WRITE succeeds and writes zero chunks (cwr_count == 0).

In all situations the data server MAY choose to write fewer chunks than the client requested; the client must be prepared to handle a short write and reissue CHUNK_WRITE for the remaining chunks.

The CHUNK_WRITE result includes per-chunk outcomes in cwr_block_status, cwr_block_activated, and cwr_owners, all co-indexed and one entry per chunk in the payload:

cwr_count: the number of chunks the data server successfully accepted. Chunks that failed their guard check, checksum check, or any other local precondition do not contribute to cwr_count.

cwr_committed: the stable_how4 level the data server actually applied for accepted chunks. This MUST be at least as durable as cwa_stable; see "Stability and Activation" below.

cwr_writeverf: a verifier identifying the data server's incarnation. A client uses cwr_writeverf to detect a data server restart that lost UNSTABLE4 writes: if the client's subsequent CHUNK_COMMIT returns a different writeverf than was returned by an UNSTABLE4 CHUNK_WRITE earlier, the chunks may have been lost and the client SHOULD re-issue CHUNK_WRITE. cwr_writeverf changes on every data server restart that loses uncommitted state.

cwr_block_status: per-chunk acceptance status; see "Per-Block

Acceptance Semantics" below.

cwr_block_activated: per-chunk activation flag. TRUE indicates that the chunk is COMMITTED on return from CHUNK_WRITE -- the activation shortcut described under "Stability and Activation" below. FALSE indicates that the chunk is in the PENDING state and requires a subsequent CHUNK_FINALIZE and CHUNK_COMMIT to become COMMITTED.

cwr_owners: per-chunk chunk_owner4 the data server recorded. In normal operation this matches cwa_owner with cg_gen_id incremented for each chunk; the field is reported explicitly so a client that lost track of its per-chunk gen counter can recover the data server’s view.

Except when special stateids are used, cwa_stateid represents a layout stateid returned by a prior LAYOUTGET against the metadata server (see Section 18.43 of [RFC8881]) that authorizes write access to this file. Under trusted-stateid tight coupling (Section 25.12), the data server additionally checks that the metadata server has registered the stateid via TRUST_STATEID; an unregistered stateid (other than a special stateid) returns NFS4ERR_BAD_STATEID.

For a CHUNK_WRITE with a cwa_stateid value of all bits equal to zero, the data server MAY allow the CHUNK_WRITE to be serviced subject to any CHUNK_LOCK currently held on the target chunks. For a CHUNK_WRITE with a cwa_stateid value of all bits equal to one, the data server MAY allow CHUNK_WRITE to bypass lock-state checking at the data server. These special-stateid behaviours mirror the corresponding WRITE semantics in [RFC8881] adapted to the chunk-locking model (Section 25.5) rather than the byte-range locking model of [RFC8881] Section 12.

If the current filehandle is not an ordinary file, an error MUST be returned. If the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. If the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases of non-regular-file filehandles, NFS4ERR_WRONG_TYPE is returned.

25.10.3.1. Stability and Activation

The cwa_stable field controls the durability level the data server guarantees before returning:

FILE_SYNC4: The data server MUST commit all written chunks plus all chunk-store metadata to stable storage before returning.

DATA_SYNC4: The data server MUST commit all written chunk payloads to stable storage and enough of the chunk-store metadata to retrieve the data before returning. An implementation MAY treat DATA_SYNC4 identically to FILE_SYNC4 at a possible performance cost.

UNSTABLE4: The data server is free to commit any portion of the chunk payload and metadata to stable storage before returning, including all or none. The data server makes only two guarantees: it will not destroy any chunk payload it accepted without changing cwr_writeverf, and the durability level it ultimately applies will not be less than that requested.

The CHUNK_WRITE_FLAGS_ACTIVATE_IF_EMPTY flag in cwa_flags requests an activation shortcut for first-time writes: a chunk that was EMPTY before the CHUNK_WRITE and whose write reaches FILE_SYNC4 or DATA_SYNC4 durability MAY be transitioned directly to COMMITTED by the data server, with the corresponding cwr_block_activated entry set to TRUE in the response. Without the flag, or for chunks that were not EMPTY before the write, or for writes at UNSTABLE4 durability, the chunk enters the PENDING state and reaches COMMITTED only after a subsequent CHUNK_FINALIZE and CHUNK_COMMIT.

The activation shortcut interacts with concurrent writers and unstable writes in subtle ways:

* A chunk written with cwa_stable == UNSTABLE4 cannot be activated by CHUNK_WRITE_FLAGS_ACTIVATE_IF_EMPTY because the payload has not been committed to stable storage; the chunk enters the PENDING state regardless of the flag.

* Two clients racing on a chunk in multiple-writer mode each see chunk_guard4 contention. One client wins the per-chunk CAS; if its CHUNK_WRITE had CHUNK_WRITE_FLAGS_ACTIVATE_IF_EMPTY set and stable was FILE_SYNC4 or DATA_SYNC4, the winning chunk becomes COMMITTED. The losing client sees NFS4ERR_CHUNK_GUARDED in the corresponding cwr_block_status slot.

* A client that issues an UNSTABLE4 CHUNK_WRITE and observes a FALSE entry in cwr_block_activated for a chunk MAY still find that chunk COMMITTED on a subsequent CHUNK_READ -- another client could have activated it via the shortcut after this one’s response was sent. cwr_block_activated reflects the state at the moment the CHUNK_WRITE result was constructed, not a commitment to that state’s persistence.

25.10.3.2. 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.

25.10.3.3. 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 86):

* 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 12.6), 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.

25.10.4. RESPONSE CODES

NFS4_OK: the chunks have been written and are in the PENDING state.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to write to this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_CHUNK_GUARDED: the chunk_guard4 condition supplied by the client did not match the persisted state.

NFS4ERR_CHUNK_LOCKED: one or more chunks in the requested range are locked by another writer.

NFS4ERR_DELAY: the data server is temporarily unable to process the request.

NFS4ERR_FHEXPIRED: the current filehandle has expired.

NFS4ERR_IO: an I/O error occurred while persisting the chunks.

NFS4ERR_NOSPC: there is insufficient space at the data server.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_WRITE.

NFS4ERR_SERVERFAULT: the data server failed while processing the

request.

NFS4ERR_STALE: the current filehandle no longer identifies a valid file.

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;
///     checksum4          cwra_checksums<>;
///     opaque             cwra_chunks<>;
/// };
```

Figure 88: 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 89: XDR for CHUNK_WRITE_REPAIR4resok

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

Figure 90: XDR for CHUNK_WRITE_REPAIR4res

25.11.3. DESCRIPTION

CHUNK_WRITE_REPAIR is the repair-path variant of CHUNK_WRITE (Section 25.10). It writes reconstructed chunk data to a data server whose chunks have been reported errored (via CHUNK_ERROR, see Section 25.2) or to a replacement data server selected during whole-file repair. The data server applies repair-specific policies to the write that are not appropriate for normal client writes: the chunk_guard4 CAS check is bypassed (the repair client is writing a reconstructed value rather than competing in a multiple-writer race), and the data server MAY log the repair separately for operator audit.

CHUNK_WRITE_REPAIR has no direct analog in [RFC8881]; it is the chunk-protocol equivalent of writing reconstructed data into a RAID stripe whose other members are known healthy. The reconstructed data is produced by the repair client from surviving shards via the erasure-coding algorithm of the file's layout (RS matrix inversion or Mojette corner-peeling, see Section 11.4 and Section 11.5).

The repair workflow that invokes CHUNK_WRITE_REPAIR is:

1. The repair client (selected per Section 11.2.4) reads surviving chunks from the remaining data servers via CHUNK_READ (Section 25.6).

2. The repair client reconstructs the missing chunks using the erasure-coding algorithm of the file's layout.

3. The repair client acquires a CHUNK_LOCK (Section 25.5) on the target data server to prevent concurrent writes during repair. For repair that adopts an MDS-escrow lock, the CHUNK_LOCK carries CHUNK_LOCK_FLAGS_ADOPT (Section 24.1.4).

4. The repair client writes the reconstructed data via CHUNK_WRITE_REPAIR.

5. The repair client issues CHUNK_FINALIZE (Section 25.3) and CHUNK_COMMIT (Section 25.1) to persist the repair.

6. The repair client issues CHUNK_REPAIRED (Section 25.7) to clear the errored state.

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

The arguments mirror CHUNK_WRITE except that CHUNK_WRITE_REPAIR has no cwa_flags field (the activation-shortcut behaviour is not offered on the repair path) and no cwa_guard field (the guard CAS is bypassed by construction):

cwra_stateid: the layout stateid the metadata server granted to the repair client. Under trusted-stateid tight coupling (Section 25.12), this stateid MUST be in the data server's trust table; otherwise the data server rejects the operation with NFS4ERR_BAD_STATEID.

cwra_offset: starting chunk index in the file (not a byte offset).

cwra_stable: the stable_how4 durability level the data server MUST apply before returning. Semantics match cwa_stable in CHUNK_WRITE (see Section 25.10 "Stability and Activation").

cwra_owner: the chunk_owner4 (Figure 52) the repair client uses for the reconstructed payload. The cg_client_id MUST be the repair client's own ffv2m_client_id (not CHUNK_GUARD_CLIENT_ID_MDS); the cg_gen_id is the repair client's locally chosen per-chunk generation counter. The reserved sentinels CHUNK_GUARD_CLIENT_ID_NONE and CHUNK_GUARD_CLIENT_ID_MDS MUST NOT appear in cwra_owner; see Section 24.1.3 and Section 24.1.4.

cwra_payload_id: the payload-id the repair client associates with the reconstructed payload, used by the repair coordinator to correlate this repair across mirrors.

cwra_chunk_size: the nominal chunk size of the reconstructed payload, in bytes.

cwra_checksums: per-chunk checksum4 (Section 24.3) array. Semantics match cwa_checksums in CHUNK_WRITE: each entry's cs_algorithm MUST match ffv2m_checksum_algorithm of the mirror named in the layout (Section 8.8), with NFS4ERR_INVAL on mismatch.

cwra_chunks: the reconstructed chunk payload as an opaque blob, packed identically to cwa_chunks in CHUNK_WRITE.

The CHUNK_WRITE_REPAIR result reports per-chunk outcomes:

cwrr_count: the number of chunks the data server successfully accepted.

cwrr_committed: the stable_how4 level the data server actually applied. MUST be at least as durable as cwra_stable.

cwrr_writeverf: a verifier identifying the data server's incarnation. Semantics match cwr_writeverf in CHUNK_WRITE.

cwrr_status: per-chunk acceptance status, one entry per chunk in the payload, co-indexed. The top-level CHUNK_WRITE_REPAIR status is NFS4_OK as long as the data server could evaluate each chunk; per-chunk failures are reported in cwrr_status rather than by failing the whole operation.

The target chunks SHOULD be in the errored state (set by a prior CHUNK_ERROR) or EMPTY. If a target chunk is COMMITTED with valid data, the data server MAY reject the repair-write with NFS4ERR_INVAL in the corresponding cwrr_status slot to prevent overwriting good data; the repair client SHOULD re-verify the chunk before attempting another repair-write on the same range.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

25.11.4. RESPONSE CODES

NFS4_OK: the repair-write succeeded.

NFS4ERR_ACCESS: the layout stateid or credentials are not permitted to write repair data to this file.

NFS4ERR_BADXDR: arguments could not be decoded.

NFS4ERR_BAD_STATEID: no active layout stateid for this file (or, in trusted-stateid tight coupling, the stateid is not in the trust table). See Section 25.

NFS4ERR_DELAY: the data server is temporarily unable to process the request.

NFS4ERR_FHEXPIRED: the current filehandle has expired.

NFS4ERR_IO: an I/O error occurred while persisting the repair data.

NFS4ERR_NOSPC: there is insufficient space at the data server.

NFS4ERR_NOTSUPP: the data server does not implement CHUNK_WRITE_REPAIR.

NFS4ERR_SERVERFAULT: the data server failed while processing the request.

NFS4ERR_STALE: the current filehandle no longer identifies a valid

file.

25.12. Operation 89: 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 91: XDR for TRUST_STATEID4args

25.12.2. RESULTS

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

Figure 92: 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 has no analog in [RFC8881]: pNFS layouts in RFC 8881 do not register the layout stateid with data servers; data servers in the loose-coupling model trust the synthetic uid/gid the metadata server inserts on each I/O (Section 6.2). TRUST_STATEID is the new MDS- to-DS control-plane operation that replaces synthetic-uid fencing with per-client stateid-table validation for deployments that opt into tight coupling.

TRUST_STATEID is an MDS-to-DS operation; pNFS clients MUST NOT send it. The data server MUST verify that the operation arrived on a session whose owning client presented EXCHGID4_FLAG_USE_PNFS_MDS at

EXCHANGE_ID and reject any TRUST_STATEID received on a regular client session with NFS4ERR_PERM. TRUST_STATEID operates on the current filehandle; a PUTFH naming the data server's file MUST precede it in the same compound (except in the capability probe case, where the current filehandle is the root).

The metadata server provides:

tsa_layout_stateid: the stateid the metadata server issued in the LAYOUTGET that produced this layout. 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: the iomode of the layout (LAYOUTIOMODE4_READ or LAYOUTIOMODE4_RW). The data server MAY enforce this against the CHUNK_* operation presented later: a READ-iomode trust entry does not authorize CHUNK_WRITE.

tsa_expire: 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: 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 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 (Section 25.13) and BULK_REVOKE_STATEID (Section 25.14) so that revocation only affects entries registered by the same metadata server. 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.

TRUST_STATEID returns only a top-level status; there is no result body beyond the nfsstat4 discriminant.

If the current filehandle is not an ordinary file (except in the capability-probe case, where the current filehandle is the root and the operation is expected to be rejected with NFS4ERR_INVAL), an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

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 90: REVOKE_STATEID - Revoke Registered Stateid on Data Server

25.13.1. ARGUMENTS

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

Figure 93: XDR for REVOKE_STATEID4args

25.13.2. RESULTS

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

Figure 94: 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. REVOKE_STATEID is the per-file revoke counterpart to TRUST_STATEID (Section 25.12) -- registration and revocation form the matched pair that drives the per-file trust table for a tightly coupled deployment.

REVOKE_STATEID has no analog in [RFC8881]. RFC 8881 revokes pNFS layouts via LAYOUTRETURN with a special all-files marker or via implicit lease expiry; REVOKE_STATEID is the new MDS-to-DS surface that lets the metadata server force per-client invalidation at the data server without waiting for tsa_expire and without unsetting other clients' trust entries.

REVOKE_STATEID is an MDS-to-DS operation; pNFS clients MUST NOT send it. The data server MUST verify that the operation arrived on a session whose owning client presented EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID and reject any REVOKE_STATEID received on a regular client session with NFS4ERR_PERM. REVOKE_STATEID operates on the current filehandle; a PUTFH naming the data server's file MUST precede it in the same compound.

The metadata server provides:

rsa_layout_stateid: the stateid to revoke. Together with the current filehandle this identifies the trust entry to remove. MUST NOT be a special stateid; the anonymous stateid is rejected with NFS4ERR_INVAL and other special stateids 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 (Section 6.2).

* 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.

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 24.1.4). 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.

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.

REVOKE_STATEID returns only a top-level status; there is no result body beyond the nfsstat4 discriminant.

If the current filehandle is not an ordinary file, an error MUST be returned (NFS4ERR_ISDIR / NFS4ERR_SYMLINK / NFS4ERR_WRONG_TYPE).

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 91: BULK_REVOKE_STATEID - Revoke All Stateids for a Client

25.14.1. ARGUMENTS

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

Figure 95: XDR for BULK_REVOKE_STATEID4args

25.14.2. RESULTS

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

Figure 96: 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 a single named client. Unlike REVOKE_STATEID (Section 25.13), which removes one entry identified by a (filehandle, stateid) pair, BULK_REVOKE_STATEID does not target a specific filehandle or stateid; it instructs the data server to scan its trust table and remove every entry whose owning pNFS client matches brsa_clientid (and whose issuing metadata server matches the calling MDS).

BULK_REVOKE_STATEID has no analog in [RFC8881]. RFC 8881 recall-all is expressed at the layout layer (CB_LAYOUTRECALL with LAYOUTRECALL4_ALL); the per-client trust-table sweep introduced here is the data-server-side complement, replacing the N per-file REVOKE_STATEID compounds that per-entry revocation would require with a single round-trip.

BULK_REVOKE_STATEID is an MDS-to-DS operation; pNFS clients MUST NOT send it. The data server MUST verify that the operation arrived on a session whose owning client presented EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID and reject any BULK_REVOKE_STATEID received on a regular client session with NFS4ERR_PERM. BULK_REVOKE_STATEID does not operate on the current filehandle; no PUTFH is required in the compound.

The metadata server provides:

brsa_clientid: the clientid of the pNFS client whose trust entries are to be removed. The special all-zeros value means "remove every trust entry owned by the calling metadata server, regardless of which pNFS client registered it"; the data server MUST interpret this value as a sweep of its own entries only, NOT as the pNFS client whose clientid happens to be zero and NOT as a global cross-MDS table clear.

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 with brsa_clientid set to the expired client’s clientid sweeps every per-file trust entry the metadata server had registered for that client.

* 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 with brsa_clientid set to 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.

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 24.1.4), rather than dropping them. Subsequent CHUNK_* operations from the revoked client fail with NFS4ERR_BAD_STATEID; locks held under those revoked stateids remain until adopted by a repair client via CHUNK_LOCK with CHUNK_LOCK_FLAGS_ADOPT (Section 25.5).

BULK_REVOKE_STATEID returns only a top-level status; there is no result body beyond the nfsstat4 discriminant.

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 97: 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 98: nfs_cb_argop4 amendment block

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

Figure 99: 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 100: XDR for CB_CHUNK_REPAIR4args

26.1.2. RESULTS

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

Figure 101: XDR for CB_CHUNK_REPAIR4res

26.1.3. DESCRIPTION

CB_CHUNK_REPAIR is sent by the metadata server to a selected pNFS client to request that the client repair one or more non-atomic chunk ranges on the file's data servers. CB_CHUNK_REPAIR is the back-channel companion to the chunk repair flow: the metadata server selects a repair client per Section 11.2.4 (those rules are normative for how the client MUST respond on receipt of this callback) and uses CB_CHUNK_REPAIR to deliver the work item.

CB_CHUNK_REPAIR has no analog in [RFC8881]. RFC 8881 back-channel callbacks operate at the layout layer (CB_LAYOUTRECALL) or the file-state layer (CB_RECALL, CB_NOTIFY); CB_CHUNK_REPAIR is the new chunk-layer callback that drives reconstruction or rollback of non-atomic chunks without requiring a full layout return.

The metadata server provides:

ccra_fh: the filehandle of the file whose chunks are non-atomic. The callback compound carries the filehandle directly; there is no preceding PUTFH in callback compounds.

ccra_layout_stateid: 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 in that case) and MUST acquire one via LAYOUTGET before issuing any CHUNK_* operation on the ranges.

ccra_deadline: 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.

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 non-atomicity 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 non-atomic 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 and retry behaviour differs.

ccra_ranges: the list of 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.

CB_CHUNK_REPAIR returns only a top-level status in ccrr_status; see "RESPONSE CODES" below for the normative meanings the metadata server attaches to each returned nfsstat4.

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 v2 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. Checksum Integrity Scope

The checksum values carried in CHUNK_WRITE and returned from CHUNK_READ defend against accidental data corruption during storage or transmission -- bit flips on storage media, network errors, software bugs in the erasure transform. The threat model an individual deployment achieves depends on which checksum_algorithm4 (Section 24.3) the metadata server selects for the file's mirrors via ffv2m_checksum_algorithm (Section 8.8):

Bit-flip-class algorithms (CRC32, CRC32C, Fletcher4): Detect accidental bit-level corruption with high probability. Do NOT defend against an adversary who can modify the payload and recompute a valid checksum, because these algorithms are not cryptographic and the algorithm identifier and parameters are public. Suitable when the threat model excludes adversaries on the wire and at rest.

Cryptographic algorithms (SHA-256, SHA-512, BLAKE3): Detect accidental corruption AND defend against adversarial modification, provided that (a) the algorithm choice itself is communicated over a trusted channel (typically the layout from the MDS) and (b) the integrity-protected recomputation happens at trust boundaries the deployment controls. Suitable when chunks may be at rest on storage the deployment does not fully control, or when transit may cross hostile network segments.

CHECKSUM_ALG_NONE: No protocol-level integrity check. The deployment is relying on transport-layer integrity (RPC-over-TLS [RFC9289], RPCSEC_GSS_KRB5I) or storage-layer integrity (filesystem checksums on the data server, RAID-level integrity) instead. Suitable when those other layers are reliably present end-to-end and the per-chunk wire protection would be redundant.

Deployments requiring protection against active attackers SHOULD select one of the cryptographic algorithms, OR use CHECKSUM_ALG_NONE in conjunction with RPC-over-TLS (Section 27.4) or RPCSEC_GSS, whichever fits the deployment's existing security architecture.

An authenticated client is in the "active attacker" role with respect to its own chunks, in a restricted sense. The data server validates the checksum against the bytes the client provided, so an authenticated client that chooses to send semantically-invalid bytes with a correctly computed checksum 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 flexible file v2 layout 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.

Flexible file v2 layout does not attempt to defend against this authenticated-but-malicious case. The checksum 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_ATOMIC (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.

27.6. Trusted Stateids

The TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID operations (Section 25.12, Section 25.13, Section 25.14) introduce a per-stateid authorization channel between the metadata server and the data server. The security implications of that channel are distinct from those of the loosely coupled synthetic-uid model (Section 6.2) and warrant their own treatment.

27.6.1. Interaction with Kerberos and RPCSEC_GSS

Trusted stateids decouple the credential the data server uses to authorize I/O from the credential the client uses to authenticate to the data server. Under loose coupling (Section 6.2), the metadata server inserts a synthetic uid/gid into the layout and the client presents that synthetic credential on every data-server RPC; the data server has no independent verification of the client’s identity, and a client that learns another client’s synthetic uid/gid can impersonate it on the data path. Tight coupling via TRUST_STATEID changes this in three ways:

* The metadata server records the client’s authenticated principal in the trust entry via tsa_principal at TRUST_STATEID time (Section 25.12). Under RPCSEC_GSS (typically Kerberos V5 GSS-API per [RFC4121]), tsa_principal is the GSS display name (for example, "alice@REALM"); under AUTH_SYS and TLS, tsa_principal is the empty string.

* The client presents its own RPCSEC_GSS context on each CHUNK_* operation against the data server. Under tight coupling with GSS, the data server MUST verify that the principal carried in the inbound RPC's RPCSEC_GSS context matches the tsa_principal recorded for the stateid in its trust table; a mismatch returns NFS4ERR_ACCESS. A client that learned another client's layout stateid (from a log file, a packet capture of cleartext RPC, or any other leak) cannot use it because their own GSS principal would not match.

* The data server does NOT need its own Kerberos keytab to validate each client principal individually. In a loose-coupling Kerberos deployment the data server would have to be a service principal in every realm it serves clients from; under tight coupling the data server's keytab is only required for its session with the metadata server (the control session, Section 6.4). Operational complexity of Kerberos deployment is meaningfully reduced.

The mechanism does not authenticate the metadata server to the client; it authenticates the client to the data server using credentials the metadata server vouched for at LAYOUTGET time. Compromise of the metadata server allows an attacker to register arbitrary trust entries; the metadata server is the trust anchor for the layout grant, unchanged from the existing pNFS layout-issuance model.

27.6.2. Attack Surfaces and Mitigations

Compromised metadata server: An attacker controlling the metadata server can issue TRUST_STATEID for any (layout stateid, principal) pair. This is the same trust assumption pNFS already makes -- the metadata server grants layouts and the data servers honour them. Deployment defence is the same: restrict administrative access to the metadata server, require RPCSEC_GSS or RPC-over-TLS ([RFC9289]) with mutual authentication on the control session (Section 6.4), and monitor for anomalous TRUST_STATEID volume.

Compromised data server: A compromised data server sees plaintext chunk payloads at rest and on the wire (subject to whatever the deployment uses for at-rest encryption and transport security). It can return arbitrary content on CHUNK_READ with a correctly computed checksum; the checksum protects against transport corruption, not adversarial content (Section 27.1). This is the same as the RAID-stripe trust model: each shard host can lie about its shard. Deployment defences are encryption at rest, an integrity- protected transport (RPCSEC_GSS_KRB5I or TLS), and physical or logical isolation of data servers.

Stateid leak from client to attacker: Under tight coupling with RPCSEC_GSS, a leaked stateid is not exploitable: the attacker's own RPC principal will not match tsa_principal in the trust table, and the data server returns NFS4ERR_ACCESS. Under tight coupling with AUTH_SYS over TLS (where tsa_principal is empty), a leaked stateid is exploitable by any attacker who can also forge the source-address binding the data server's TLS session expects; this is the standard AUTH_SYS-over-TLS trust model, unchanged.

Replay of revoked stateid: After REVOKE_STATEID or BULK_REVOKE_STATEID the data server removes the trust entry and subsequent CHUNK_* operations presenting the revoked stateid fail with NFS4ERR_BAD_STATEID (Section 25.13). An in-flight CHUNK_* operation that arrived before the revoke completed MAY be allowed to finish; the chunk_guard4 CAS (Section 24.1) bounds the worst-case damage from such in-flight I/O to the chunks already PENDING at revocation time, and the lock-transfer-to-MDS-escrow rule (Section 24.1.4) prevents a write hole from opening during revocation.

Compromised control session: An attacker who controls the metadata-server-to-data- server control session can register or revoke arbitrary trust entries. The control session is the most security-sensitive surface introduced by tight coupling. Deployment MUST protect it with RPCSEC_GSS ([RFC7861]) using a service principal both sides trust, or with RPC-over-TLS ([RFC9289]) using mutual authentication and allowlisted certificates. The data server enforces that TRUST_STATEID, REVOKE_STATEID, and BULK_REVOKE_STATEID only arrive on sessions whose owning client presented EXCHGID4_FLAG_USE_PNFS_MDS at EXCHANGE_ID (Section 25.12), but that flag alone does not authenticate the metadata server.

Resource exhaustion via trust-table flood: A misbehaving metadata server could register an unbounded number of TRUST_STATEID entries to exhaust the data server's trust-table memory. The mechanism defending against this is the tsa_expire lease on each entry: trust entries that are not renewed before expiry are reaped by the data server. A data server under memory pressure MAY also return NFS4ERR_DELAY on new TRUST_STATEID requests, forcing the metadata server to back off.

Cross-metadata-server isolation: In a deployment where two metadata servers share a single data server, the per-entry metadata-server tag (derived from the control session's owning client; see Section 25.12) ensures that REVOKE_STATEID and BULK_REVOKE_STATEID from one metadata server cannot remove entries registered by the other. A compromised metadata server can, however, register

entries against any filehandle the data server exposes to it. Deployments concerned about cross-metadata-server isolation MUST partition the data server's filesystem namespace into per-metadata-server exports at the data server, rather than rely on the trust table alone to enforce file-level boundaries between metadata servers.

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 8.1.4). 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 five encoding types: the flexible file v1 layout-compatible PASSTHROUGH (see Section 8.1.2), the chunked MIRRORED (see Section 8.1.3), and three chunked erasure coding types (see Table 14).

Encoding Type Name	Value	RFC	How	Minor Versions
FFV2_ENCODING_PASSTHROUGH	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
FFV2_ENCODING_MIRRORED	5	RFCTBD10	L	2

Table 14: Flexible File Version 2 Layout Type Encoding Type Assignments

28.1. Checksum Algorithm Registry

This document introduces the "Flexible File Version 2 Layout Type Checksum Algorithm Registry". Values in this registry name the checksum_algorithm4 (Section 24.3) carried in checksum4 on the wire and selected per-mirror via ffv2m_checksum_algorithm (Section 8.8).

The registry uses a 32-bit value space. Registration policy is Specification Required [RFC8126]; the Designated Expert reviews each request for:

* a complete and publicly available specification of the algorithm sufficient for independent interoperable implementations;

* the exact length of the cs_value field for this algorithm (a single registered length per algorithm; variable-length variants register separately);

* collision risk against existing registrations (the Expert MAY decline to register an algorithm whose output overlaps substantially with an existing registration).

Initial registrations are listed in Table 15.

Name	Value	cs_value bytes	Class	RFC
CHECKSUM_ALG_NONE	0	0	none	RFCTBD10
CHECKSUM_ALG_CRC32	1	4	bit-flip	RFCTBD10
CHECKSUM_ALG_CRC32C	2	4	bit-flip	RFCTBD10
CHECKSUM_ALG_FLETCHER4	3	32	bit-flip	RFCTBD10
CHECKSUM_ALG_SHA256	4	32	cryptographic	RFCTBD10
CHECKSUM_ALG_SHA512	5	64	cryptographic	RFCTBD10
CHECKSUM_ALG_BLAKE3	6	32	cryptographic	RFCTBD10

Table 15: Initial Checksum Algorithm Registrations

CHECKSUM_ALG_NONE (value 0) indicates that no protocol-level checksum is computed. The deployment relies on transport-layer integrity (RPC-over-TLS, RPCSEC_GSS_KRB5I) or storage-layer integrity instead; see Section 27.1.

CHECKSUM_ALG_CRC32 (value 1) is the CRC32 algorithm used in this draft's predecessor revisions. It is registered for backward conceptual compatibility; deployments SHOULD prefer CHECKSUM_ALG_CRC32C for new files since CRC32C is hardware-accelerated on every modern CPU.

CHECKSUM_ALG_CRC32C (value 2) is the CRC32 with the Castagnoli polynomial (0x1EDC6F41), as used in iSCSI, SCTP, and the SSE4.2 / ARMv8 / RISC-V hardware-acceleration instructions.

CHECKSUM_ALG_FLETCHER4 (value 3) is the Fletcher's checksum variant used in ZFS, comprising four 64-bit accumulators concatenated to produce a 32-byte output. Other Fletcher4 implementations that truncate to a shorter output register separately.

CHECKSUM_ALG_SHA256 (value 4), CHECKSUM_ALG_SHA512 (value 5), and CHECKSUM_ALG_BLAKE3 (value 6) are cryptographic hashes with standard outputs at the lengths listed. BLAKE3 is registered at its standard 32-byte output; extended-output BLAKE3 (the algorithm's XOF mode at other lengths) registers separately.

A checksum4 whose cs_value length does not match the registered cs_value bytes for its cs_algorithm MUST be rejected with NFS4ERR_INVAL.

The "Class" column in Table 15 is informational and indicates the threat model the algorithm supports; see Section 27.1.

29. XDR Description of the Flexible File Version 2 Layout Type

This document contains the External Data Representation (XDR) [RFC4506] description of the flexible file v2 layout. 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 102 to produce the machine-readable XDR description of the flexible file v2 layout.

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

Figure 102: 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 103.

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

Figure 103: 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.

30. References

30.1. 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>.

30.2. 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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

Implementation Status

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

This appendix 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. This appendix is reviewer-aid material and is removed from the final RFC.

reffs (metadata server and data server) 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 flexible file v2 layout deployment. ec_demo is a client-side library with a demonstration driver that exercises the flexible file v2 layout 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 checksum 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-data server 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 flexible file v2 layout 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 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 data server 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), so its reconstruction overhead does not exhibit the same growth 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. The read cost is independent of failure count, which is the algorithmic trade-off of the non-systematic form.

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 quantify the algorithmic trade-offs each codec family makes: Mojette non-systematic's constant decode cost comes at a higher baseline read cost, and Reed-Solomon's matrix- inversion reconstruction grows as O(k^2) at wider geometries. The choice of default codec and geometry in a given deployment follows from these properties applied to the workload's read / write mix, fault-tolerance target, and acceptable encoding cost.

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?" At the systematic-codec operating points measured (Mojette systematic at 4+2 and 8+2), the benchmark answer 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 12.6), 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 Appendix "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

This appendix records design alternatives that were considered and rejected during the development of this specification. It is reviewer-aid material in this draft and is retained in the final RFC as design-history context for future implementers; the alternatives below are not part of the normative specification.

The design of flexible file v2 layout 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. The motivation was concrete: NFSv3 READ and WRITE arguments carry data as opaque data<> and provide no XDR room for per-write structured metadata such as codec geometry, integrity, or write-ordering tiebreakers. An NFSv3 server cannot be extended; if a flexible file v2 layout deployment wanted an NFSv3 server to participate as a data server in an erasure-coded layout, the only place to put codec metadata was inside that opaque payload, prepended to the data bytes. The data server stored the entire opaque blob without interpreting it; the reader peeled the 16-byte prefix off and acted on it.

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. A generic NFSv3 inspector watching the wire could not tell those bytes apart from application data, which among other things made debugging, traffic analysis, and middlebox processing rely on out-of-band knowledge.

The endianness objection raised at IETF 120 (July 2024) was the surface complaint; the structural objection -- that smuggling structured fields through an opaque type bypasses XDR's self-description -- was the deeper reason the working group declined the approach. Once the design accepted that data servers in a flexible file v2 layout deployment would speak NFSv4.2 (with new ops in this document), the constraint that forced the smuggling disappeared: chunk metadata could be expressed as proper XDR fields in CHUNK_WRITE / CHUNK_READ / chunk_guard4, visible to every observer of the wire.

Per-Client Swap Files with metadata server 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 metadata server replicas (see Section 12.9) -- 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

An alternative raised at IETF 122 (March 2025) was 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 Proxy Server mechanism (see [I-D.haynes-nfsv4-flexfiles-v2-proxy-server]) 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 proxy server 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

An alternative raised at IETF 121 (November 2024) was 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 flexible file v2 layout 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 12.9).

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

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

This appendix captures a working-group concern raised during the review of an earlier revision of this draft: the source of the concern, the question as the working group asked it, the authors' understanding of what was being asked, and how the current specification addresses it. This appendix is reviewer-aid material and is removed from the final RFC.

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

A proxy server is a peer of the metadata server 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 proxy server 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 proxy servers and data servers, 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 proxy server, not the client. Clients see ordinary NFSv4.2 semantics; the proxy server is responsible for converting those semantics into the chunk state-machine the data servers 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 proxy server is the answer for clients that either cannot speak the codec or are too old to be upgraded. In Christoph's framing, the proxy server is the inside of the storage boundary; codec-aware clients are implementations that have been admitted into that boundary by design.

The proxy server does carry a data-plane cost: client bytes traverse the proxy on the way to the data servers, so the proxy's link sees roughly twice the bandwidth of a direct client-to-data server path, and the proxy server 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 data servers directly.

Working Group Concern: Coherent Multi-data server Writes Without Recall Storms

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

This appendix captures a working-group concern raised during the review of an earlier revision of this draft: the source of the concern, the question as the working group asked it, the authors' understanding of what was being asked, and how the current specification addresses it. This appendix is reviewer-aid material and is removed from the final RFC.

Source

Christoph Hellwig, IETF 122, NFSv4 Working Group session, during the flexible file v2 layout 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 metadata server 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 metadata server can mutate. In an erasure-coded workload with many concurrent clients, this turns a localised data server 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 data server 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 metadata server issues a real layout stateid and fans out TRUST_STATEID to each data server in the mirror set, registering (stateid.other, fh, clientid, iomode, expire) in a per-data server trust table. CHUNK_WRITE and CHUNK_READ on the data server now validate against the trust table; an unknown, expired, or revoked stateid yields NFS4ERR_BAD_STATEID.

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

* When the metadata server 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 data server without affecting other clients.

The control-plane cost reshapes accordingly:

* Layout mutation is no longer global. The metadata server reroutes data to a spare data server, 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 metadata server 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 metadata server, the entire repair can complete before any client notices. The metadata server 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 data server. 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 data server 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 metadata server 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 metadata server reroutes mid-write the data servers can detect non-atomic 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 proxy server link, paid only by clients that route through a proxy server 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.

The Mojette Transform encoding type specification in Section 11.5 -- including the algebra, the bin convention, the projection sizing, and the reconstruction algorithms -- was contributed by Pierre Evenou, 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 flexible file v2 layout is RAID consistency across the shards 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 12.6.

The authors thank Dave Noveck, Chuck Lever, Tigran Mkrtchyan, Rick Macklem, and Christoph Hellwig for their detailed review of earlier revisions of this draft. Their comments shaped the system model presentation, the chunk lifecycle and guard semantics, the trusted-stateid design, and many smaller choices recorded throughout the document.

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

Author’s Address

Thomas Haynes
Hammerspace
Email: loghyr@gmail.com