Network File System Version 4 T. Haynes
Internet-Draft P. Evenou
Intended status: Standards Track Hammerspace
Expires: 20 September 2025 19 March 2025

The Mojette Transformation for the Erasure Coding of Files in NFSv4.2
draft-haynes-nfsv4-mojette-encoding-01

Abstract

Parallel NFS (pNFS) allows a separation between the metadata (onto a metadata server) and data (onto a storage device) for a file. The flex file layout type version 2 further allows for erasure encoding types to provide data integrity. In this document, a new erasure encoding type for the Mojette Transformation is introduced.

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/mojette_encoding.

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 20 September 2025.

Table of Contents

1. Introduction

In Parallel NFS (pNFS) (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. [I-D.haynes-nfsv4-erasure-encoding] defined the Flexible File Version 2 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]. This document introduces a new Erasure Encoding Type called Mojette Transformation.

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].

1.1. Requirements Language

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

2. Mojette Transform

2.1. Introduction

The Mojette Transform is an erasure coding technique that provides fault tolerance for data storage systems by enabling the recovery of lost data blocks. This section describes the integration of the systematic Mojette Transform into the NFS protocol, focusing on encoding and decoding file system blocks, typically sized at 4KB or 8KB.

2.1.1. Encoding

The Mojette Transform involves the following steps to encode a data block:

Initialization Each data block is treated as a 2D grid of data elements (pixels). Typically, a block is structured as a matrix of size $P \times Q$, where P and Q are the dimensions of the grid.

Projections Calculation Projections are computed along specific directions defined by pairs of coprime integers (p_i, q_i). Each projection sums the values of the data elements (pixels) along a line defined by these directions. The size of a projection is given as in Figure 1. For a given projection direction (p_i, q_i), and if we let Δ be 1 if the argument is zero and 0 otherwise, the projection values are calculated as in

$$\text{Projection}(b, p_i, q_i) = \sum_{k=0}^{Q-1} \sum_{l=0}^{P-1} \text{Data}(k, l) \ \Delta(b - l \ p_i + k \ q_i)$$

$$\text{Size of projection} = (P - 1) \times |q| + (Q - 1) \times |p| + 1$$

Figure 1: Size of Projection Calculation

2.1.2. Decoding

Decoding the Mojette Transform is the inverse of encoding, involving the reconstruction of data from projection data to fill an empty grid. This involves solving a system of linear equations defined by the projection differences and the projection directions (p_i, q_i). The algorithm iterates to refine the values of the missing data elements until the original data block is reconstructed.

Data reconstruction is possible if Katz's criterion holds, which was extended to any convex shape. It specifies that reconstruction is valid if for a given set of n projections along n directions (p_i, q_i) either $\sum_{i=0}^{n} q_i \ Q$ or $\sum_{i=0}^{n} p_i \ P$.

Adjusting the number of lines Q and the projections set allows setting a desired fault-tolerance threshold.

For example a 64x4 grid can be decoded by the projection set {(0, 1), (1, 1), (2, 1), (3, 1)} as $\sum_{i=1}^{4} q_i = 4$.

2.1.3. Systematic and Non Systematic Implementations

A systematic code is an error-correcting code where the input data is embedded directly in the encoded output. In contrast, a non-systematic code produces an output that does not contain the original input symbols. The Mojette Transform can be implemented in both ways, allowing it to adapt to various use cases.

2.1.4. Data Block Representation

In the context of NFS, a data block corresponds to a file system block, which is a contiguous segment of data, typically 4KB or 8KB in size. The Mojette Transform encodes these blocks to ensure data integrity and availability in distributed storage environments.

2.2. Non-Systematic Mojette Transform

2.2.1. Block Encoding

In the non-systematic version of the Mojette Transform, the original data block is not directly included in the encoded output. Instead, the entire encoded output consists of projections computed from the original data. The number of computed projections n is larger than the number of projections m required to rebuild the initial data.

2.2.2. Block Decoding

To decode a file system block that has undergone the non-systematic Mojette Transform, the following steps are followed:

Identify Available Projections Determine which projections are available. A least m projections (what ever they are) out of n must be available.

Recompute Data Block Apply the inverse Mojette Transform to rebuild the original Data.

2.2.3. Example

Assume a file system block of 4KB is divided into a 128x4 matrix of 128-bit elements. Using the non-systematic Mojette Transform, we compute projections along selected directions, such as (-2,1), (-1, -1), (0,1), (1,1), (2,1) and (3,1). The original data is not stored directly; instead, the projections are stored.

If a data loss occurs, for instance, if two projections are lost, the missing elements can be recovered by using the remaining projections and solving the inverse problem. Any set of 4 projections among the 6 generated can rebuild exactly the original data block

2.3. Systematic Mojette Transform

2.3.1. Block Encoding

In the systematic version, the original data block (file system block) is part of the encoded output. Additional projections are calculated to provide redundancy. If k is the number of original data blocks and n is the total number of encoded blocks (including projections), the systematic code will have the first k blocks as the original data and the remaining $n - k$ blocks as projections.

2.3.2. Block Decoding

To decode a file system block that has undergone the Systematic Mojette Transform, the following steps are followed:

Identify Missing Data Determine which data lines are missing in the block. Let e be the number of missing lines.

Recompute Projections Compute the projections of the available (partial) data blocks. Calculate the differences between the projections of the full data and the partial data.

Combine Existing Data and Recomputed projection Recreate the full block of data by combining the existing data with the reconstructed data according to their positions in the block.

2.3.3. Example

Assume a file system block of 4KB is divided into a 64×4 matrix of 128-bit elements. Using the systematic Mojette Transform, we first compute projections along selected directions, such as (0,1), and (1,1). The original 4 blocks of 64 128-bit elements remains part of the encoded data, and the 2 additional projections are stored for redundancy. If a data loss occurs, the missing elements can be recovered by using the projections and solving the inverse problem.

2.4. Conclusion

The Mojette Transform provides two implementations for an efficient and effective way to enhance data reliability by encoding file system blocks with additional projections. These methods ensure that data can be reconstructed even in the presence of failures, thereby enhancing the fault tolerance of the file system.

In summary, the Mojette Transform offers robust solutions for data reliability in file systems, balancing redundancy, efficiency, and performance to ensure data integrity and quick recovery from failures.

2.4.1. Benefits of Non-Systematic Mojette Transform

Fast Failure Detection By storing only encoded data and having the ability to rebuild the original block from any sufficient subset of projections, the non-systematic encoding implementation offers great flexibility and allows fast failure detection and recovery.

Constant Performance The non-systematic decoding algorithm is the most performant of all implementations. Although decoding always occurs, the overhead is low, and unlike systematic encoding, performance remains constant regardless of the number of failures.

2.4.2. Benefits of Systematic Mojette Transform

Redundancy Reduction Systematic Mojette coding reduces redundancy by integrating the original data blocks into the encoded data, unlike non-systematic codes that generate entirely new data from the original.

Efficiency Fewer projections need to be calculated and stored, reducing both computational and storage overhead.

Performance Decoding is faster and simpler, especially when some original data blocks are available, enabling quicker data access. However, performance is slightly degraded in the case of failures and depends on the number of failures.

3. Extension of Flexible File Layout Type Version 2 Encoding Type

3.1. ffv2_encoding_type4

```
/// enum ffv2_encoding_type4 {
///     FFV2_ENCODING_MIRRORED               = 0x1;
///     FFV2_ENCODING_MOJETTE_SYSTEMATIC     = 0x2;
///     FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC = 0x3;
/// };
```

Figure 2: enum ffv2_encoding_type4

fv2_encoding_type4 is extended in Figure 2 to introduce two different erasure encoding types: FFV2_ENCODING_MOJETTE_SYSTEMATIC and FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC. They are intoduced at this level instead of at the ffv2_encoding_type_data4 (Figure 4) in order for the client to negotiate the support of one over the other with the NFS4ERR_ERASURE_ENCODING_NOT_SUPPORTED error (Part ZZZ of [I-D.haynes-nfsv4-erasure-encoding]).

3.2. ffv2_mojette_faulty_devices4

```
/// enum ffv2_mojette_faulty_devices4 {
///         FFV2_MOJETTE_FAULTY_DEVICES_2_1       = 0x1;
///         FFV2_MOJETTE_FAULTY_DEVICES_4_1       = 0x2;
///         FFV2_MOJETTE_FAULTY_DEVICES_4_2       = 0x3;
///         FFV2_MOJETTE_FAULTY_DEVICES_8_1       = 0x4;
///         FFV2_MOJETTE_FAULTY_DEVICES_8_2       = 0x5;
///         FFV2_MOJETTE_FAULTY_DEVICES_8_3       = 0x6;
///         FFV2_MOJETTE_FAULTY_DEVICES_8_4       = 0x7;
/// };
```

Figure 3: enum ffv2_mojette_faulty_devices4

The ffv2_mojette_faulty_devices4 (see Figure 4) can be used in both the layout_hint (Section 5.12.4 of [RFC8881] and Part XXX of [I-D.haynes-nfsv4-erasure-encoding]) and the ffl_encoding_type_data (Part YYY of [I-D.haynes-nfsv4-erasure-encoding]) to convey the distribution of FFV2_DS_FLAGS_ACTIVE and FFV2_DS_FLAGS_SPARE projection blocks (Part XXX of [I-D.haynes-nfsv4-erasure-encoding]) in the layouts for the Flexible File Version 2 Layout Type. The name of each of the enum targets ends with 'X_Y', which states that there is a need for X + Y files to compose the projection. X is the number of FFV2_DS_FLAGS_ACTIVE blocks and Y is the number of FFV2_DS_FLAGS_SPARE blocks.

3.3. ffv2_encoding_type_data4

```
/// union ffv2_encoding_type_data4
///          switch (ffv2_encoding_type4 fetd_type) {
///     case FFV2_ENCODING_MIRRORED:
///         void;
///     case FFV2_ENCODING_MOJETTE_SYSTEMATIC:
///     case FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC:
///         ffv2_mojette_faulty_devices4
///                         fetd_mojette_potection_configuration;
/// };
```

Figure 4: union ffv2_encoding_type_data4

This addition of FFV2_ENCODING_MOJETTE_SYSTEMATIC and FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC to the ffv2_encoding_type_data4 (Figure 4) allows for the metadata server to inform the client as to the expected distribution of FFV2_DS_FLAGS_ACTIVE and FFV2_DS_FLAGS_SPARE projection blocks inside the ffm_data_servers arm for the Mojette erasure encoding types.

4. Examples Tying It All Together

4.1. Block_Sizes

Consider a FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC encoding type which needs 6 active blocks and 2 spare blocks in the payload (Not a valid ffv2_mojette_faulty_devices4, but needed to show varied block sizes). As can be seen in Table 1 , 4kB data blocks need blocks about 1kB in size. But not all blocks are the same size.

Projection ID	0	1	2	3	4	5	6	7
p value	-3	-2	-1	0	1	2	0	0
q value	1	1	1	1	1	1	0	0
size (bytes)	1048	1080	1080	1048	1064	1048	0	0

Table 1: Example sizes of a Mojette Projection

5. Extraction of XDR

This document contains the external data representation (XDR) [RFC4506] description of the uncacheable attribute. The XDR description is presented in a manner that facilitates easy extraction into a ready-to-compile format. To extract the machine-readable XDR description, use the following shell script:

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

For example, if the script is named 'extract.sh' and this document is named 'spec.txt', execute the following command:

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

This script removes leading blank spaces and the sentinel sequence '///' from each line. 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 NFSv4.2 nfs4_prot.x file (generated from [RFC7863]). 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 code snippets should be placed in their appropriate sections within the existing XDR.

6. Security Considerations

This document has the same security considerations as both Flex Files Layout Type version 1 (Section 15 of [RFC8435]) and NFSv4.2 (Section 17 of [RFC7862]).

7. IANA Considerations

This document introduces changes in the 'Flex Files V2 Erasure Encoding Type Registry'. This document defines both the FFV2_ENCODING_MOJETTE_SYSTEMATIC and FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC types for Client-Side Mojette Transformations.

Erasure Encoding Type Name	Value	RFC	How	Minor Versions
FFV2_ENCODING_MOJETTE_SYSTEMATIC	2	RFCTBD10	L	2
FFV2_ENCODING_MOJETTE_NON_SYSTEMATIC	3	RFCTBD10	L	2

Table 2: Flex Files V2 Erasure Encoding Type Assignments

8. References

8.1. Normative References

[I-D.haynes-nfsv4-erasure-encoding]
Haynes, T., "Erasure Encoding of Files in NFSv4.2", Work in Progress, Internet-Draft, draft-haynes-nfsv4-erasure-encoding-03, 5 November 2024, <https://datatracker.ietf.org/doc/html/draft-haynes-nfsv4-erasure-encoding-03>.

[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>.

[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>.

[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>.

[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>.

[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>.

8.2. Informative References

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

Acknowledgments

The following from Hammerspace were instrumental in driving the incorporation of the Mojette Transformation into an encoding type for Flexible Files Version 2 Layout Type: David Flynn, Trond Myklebust, Tom Haynes, Didier Feron, Jean-Pierre Monchanin, Pierre Evenou, and Brian Pawlowski.

Authors' Addresses

Thomas Haynes
Hammerspace
Email: loghyr@gmail.com

Pierre Evenou
Hammerspace
Email: pierre.evenou@hammerspace.com