Erasure Coding XRootD Object Store

Andrew Hanushevsky[1], *Michal Kamil* Simon[2], and *Wei* Yang[1*]

[1]SLAC National Accelerator Laboratory, 2575 Sand Hill Road, Menlo Park, CA, USA
[2]European Organization for Nuclear Research, Espl. des Particules 1, 1211 Meyrin, Switzerland

Abstract. An erasure coding (EC) algorithm was implemented in XRootD client library, and an EC-enabled XRootD storage prototype was set up at SLAC for evaluation. The architecture and configuration of the prototype is almost identical to that of a traditional non-EC XRootD storage behind a firewall: a backend XRootD storage cluster in its simplest form, and an internet facing XRootD proxy. This proxy handles EC and spreads the data stripes of a file/object across several backend nodes. It also supports all functions used on a WLCG storage system: HTTP(s) and XRootD protocols, Third Party Copy, X509/VOMS/Token, etc. The cross-node EC architecture brings significant advantages in both performance and resilience: e.g. parallel data access, tolerance of downtime and hardware failure. It also pushes the preferred network-IO mode from Posix-like to object-like. This paper will describe the prototype's architecture and its design choices, the performance in high concurrent throughputs and file/object operations, failure modes and their handling, data recovery methods, and administration.

1 Introduction

As the size of storage system increases, Erasure Coding (EC), in particular, cross node EC become important to meet the data safety requirement. The implementation of EC in XRootD [11] was initially developed at CERN [8] for the EOS Open Storage System [1]. At there, the XRootD client library uses the Intel Intelligent Storage Acceleration Library (ISA-L) [2] to place data blocks on multiple storage nodes. When writing a file "A" to an EC enabled XRootD storage cluster, the XRootD client will choose several storage nodes in the cluster, and a zip archive file with the same name "A" will then be placed on each of the nodes. The number of nodes chosen equals to the EC stripe size (number of data and parity chunks in a stripe, e.g. 8+3). Inside each zip archive file is a sequence of data chunks like *obj.0.i, obj.1.j, obj.2.k*, … where *0, 1, 2*, … refer to the index of the EC stripes, and *i, j* and *k* refer to the index of data/parity chunk within an EC stripe (e.g. 0 to 8+3-1). If the remaining data at the end of a file is smaller than the EC strip size, empty zip archives will be added and will have no effect when calculate parity. The zip archive's CRC32 feature is utilized to validate the integrity of the data chunks. Note that data and parity chunks in a stripe will be spread across chosen nodes in random orders.

When reading "A" back, the XRootD client will find all zip archive file "A"s in the cluster, read the zip central directories from them and build a map of all data/parity chunks.

* Corresponding author: yangw@slac.stanford.edu

It will use the stored CRC32 to detect data corruptions, and when needed, perform error correction using the Reed Solomon algorithm [3] that was built in the ISA-L.

2 Architecture Design of the XRootD Erasure Coding Prototype

The prototype of an Erasure Coding XRootD storage system built at the SLAC National Accelerator Laboratory [9] aimed at implementing such a system in its simplest form, and yet support all WLCG storage system requirements. The prototype consists of two main parts:

- A vanilla XRootD storage cluster at the backend, with no knowledge of EC. The only requirement is to have enough data storage nodes to spread the chunks in a data stripe on them. It can have more nodes than the EC stripe width.
- One or more frontend nodes that run the XRootD proxy services. These nodes provide all the EC related functions, as well as the WLCG [10] data access and transfer functions. EC parameters such as data chunk size (usually ranging from 64k to a few MB), and stripe width (e.g. 8+3) are set on these nodes. These nodes can be independent, or they can form a proxy cluster.

A vanilla XRootD storage with a user facing WLCG capable XRootD proxy service is a common installation at many research institutes. This prototype just adds the EC function on top of that.

2.1 Design Choices and Features

Several architecture design choices and implementation choices were made when building the prototype in order to keep it simple, easy to understand, easy to maintain and easy to use.

2.1.1 Access Mode

Since the EC functionalities are implemented in the XRootD client library as a plugin, it is possible for clients to directly load the plugin and access the vanilla XRootD storage cluster. However, this mode puts the burden of loading and configuring the plugin on the client, at both write time and read time. Since the client-side plugin is responsible for spreading the data/parity chunks, it is inefficient if the client and the backend storage are separated by the wide area network. This mode of operation is useful to perform administrative tasks within a LAN environment, but it is not attractive when used to provide user facing services.

For user facing services, a better architecture would be to put a XRootD proxy in between the backend storage and the clients. This is a common setting when the backend storage is behind a firewall. The proxy can be configured as an EC client to the backend storage (along with other functions to serve the WLCG requirements). In this mode, clients don't need any special setting for EC, they don't even have to limit themselves to only using the root protocol. This extra proxy service does introduce some performance penalty, which is discussion later in this paper. But overall, the benefit seems to overweight the performance loss, and hence this mode was chosen as the prototype architecture.

2.1.2 Location to Set the EC Parameters

The EC plugin needs to know a few parameters: The chunk size in bytes, the number of data chunks in a stripe, and the number of parity chunks in a stripe. These parameters can be obtained from the backend XRootD storage (if it is configured to supply them), or they can be set in the XRootD client configuration file. There are advantage and disadvantages in either way but they are mostly unimportant to this study. However, in order to use a vanilla

XRootD backend storage without special configuration, this prototype put these parameters in the XRootD client configuration files.

2.1.3 Choosing Storage Nodes and Locating Existing Files

A XRootD storage cluster uses a redirector to glue all the data nodes together. The mechanism is usually lightweight. It is also quick to discover the first copy of an existing file. However, it will introduce a delay (a.k.a the "qdl" delay, default of 5 seconds [7]) if the file doesn't exist, or if the locations of all copies are needed. The "location*"* refers to a list of all data nodes that host the same file.

The first scenario of delay happens when a new file is created, as XRootD needs to make sure it doesn't already exist anywhere in the storage cluster. Though painful for users accessing the storage interactively, the past decade usage shows that this delay is tolerable if most of the accessing are via batch jobs or data transfer systems.

The second scenario of delay happens when an EC client needs to locate all zip archives of a file in order to read that file back. This delay will likely have significant negative impact to the reading efficiency of interactive users, batch jobs and data transfers.

There are two ways for an EC client to locate the needed zip archives: Have the EC client (the proxy) query via the redirector or have the EC client directly query the data nodes. Assuming that the "cost" of network latency in a LAN environment is a constant x, and the cost of doing a Posix stat() call on a data node is y. In a setup with stripe width of $d+p$ (d is the number of data chunks and p is the number of parity chunks) and a XRootD cluster of m data nodes

- The overall costs of query via redirector are x * [1 + m + (d+p) + 1] and y * m. Here the first "1" refers to the initial query traffic from the EC proxy to the redirector, and the last "1" refers to response traffic from the redirector to the EC proxy. "m" refers to the query from the redirector to the data nodes, and "d+p" refers to the response of data nodes to the redirector (data nodes without the zip archive file will not response. Posix stat() calls will be performed on all data nodes, and thus y * m. Note that these overall costs aren't the measure of time, but the measure of total operations of two kinds. Some of the operations are performance in parallel, but they all induce load on the XRootD storage system. There are other, minor traffic between the "xrootd" and "cmsd" processes on the same nodes. Their costs are ignored in this discussion. Performing a query this way also suffers a "qdl" delay.
- The overall costs of directly query the data nodes by the EC proxy are x* (1 + 1 + m + m) and y * m. The "1"s refer to EC proxy query to redirector to get a list of all data nodes, and the response. The first and second "m" refers to the query traffic from EC proxy to the data nodes, and the responses. The last "m" is the same as in the above paragraph (the Posix stat() calls). In this scenario, the "qdl" delay is avoided.

With the first choice, the information is usually cached in the redirector for a while, which reduces the cost of an immediate query of the same file. No such benefit exists in the second choice, though it can be improved if a caching mechanism is developed.

For a storage cluster where m is not significantly larger than d+p, and mostly dealing with large data processing (where repeated access of the same file in a short period of time is rare), being able to avoid the "qdl" delay makes the second choice a clear winner. It is hence used in the prototype.

2.1.4 Handling the Leftover Zip Archive Files and File Stat Info

Sometimes a zip archive file was left behind on the XRootD storage cluster while all other peer zip archives files were deleted. This could happen when a data node was down during

the deletion. Even worse, it is possible that the data node won't come back online until the data file was recreated. In this case, there will be an extra zip archive file that isn't compatible with all newly created zip archive files. To identity this leftover "debris", a creation time info is recorded in the zip archive file's extended attribute, serving as both the version info and *mtime* info of the file. Only zip archive files with the latest (and same) version will be used. Also recorded in the zip archive file's extended attributes are the actual file size info, and checksum info.

2.1.5 Providing a Consistent File View and Listing

Users usual want to check what are the storage systems via a listing. In the case of EC enabled XRootD storage such as the prototype, the XRootD Posix layer was enhanced to aggregate the file metadata from the above extended attributes, and presents to the user a single file entry with the *mtime* and file size information.

2.2 Understanding the Impact to Users during Failure

The performance of the XRootD client library can be tuned via several Unix environment variables. These "parameters" are usually optimized for a WAN environment. For optimal performance in a LAN that links the frontend XRootD proxy and backend XRootD storage cluster, the prototype sets several parameters to values different from their defaults. Some of them affect what users will experience when a failure happens. In the case of an EC enabled XRootD system such as that in the prototype, when a data node's *xrootd* process goes down unexpectedly:

- A user who is reading from that data node will experience a freezing of XRD_CONNECTIONTIMEOUT * (XRD_CONNECTIONRETRY - 1) [4] seconds. After that, the reading will continue.
- A user who is writing to that data node will see the writing continue for a while. But soon it will fail, likely with a "connection refused" error (when the failure happens during writing or file closing).
- A user who initiates a writing within T seconds after a node went down may see writing failure, where T is larger of XRD_STREAMERRORWINDOW [4] seconds, and the XRD_CONNECTIONTIMEOUT * (XRD_CONNECTIONRETRY - 1) seconds. This is because it takes up to T seconds for the EC clients to be aware of the change.

Negative impacts to users can be avoid in an orderly shutdown when a data node's *cmsd* process is stopped first, followed by the *xrootd* process after seeing no connection to it.

3 Prototype and Performance

The EC enabled XRootD system prototype was setup with several goals: a) prove the system functionalities; b) understand the overhead of the chosen access mode, c) measure the performance of the chosen access mode, d) perform administrative operations. This section describes the prototype's environment itself, as well b) and c). It does not discuss a) since that is self-proving by successful measurement in c). The last goal d) is described in the next section.

3.1 The prototype Testing Environment

The prototype consists of a backend XRootD storage cluster and a frontend XRootD proxy node:

- The Backend XRootD Storage Cluster
  - 19 nodes of Dell 510 (9-year in service as of the date of testing), each has:
  - 24GB RAM, 1Gbps NIC, 12x 3TB HDD (some nodes have 11)
  - Each HDD presents itself to the operating system as its own SCSI device (via a LSI RAID controller)
  - CentOS 7 and XRootD release 5.3.4 (later upgraded to 5.4.0).
- The Frontend XRootD Proxy
  - 60-core, 128GB RAM, 100Gbps NIC
  - CentOS 7 and an unrelease XRootD (main branch at 2021-12-17 and EC related private patches, this version is newer than XRootD 5.4.0).
  - EC configuration: 1 stripe consists of 8 data chunks and 2 parity chunks. The chunk size is 1MB.

All measurements and tests were performed using the root protocol since that is where the EC client plugin was implemented. Since HTTP protocol is only used to provide user and application facing functionalities, it was not a focus of the test and measurement here. Though tests had been done using the HTTP protocol to verify that it worked as expected.

3.2 Understanding the Overhead of Accessing Data via a XRootD Proxy

There are two access modes described in section 2.1.1. To understand the overhead of using a XRootD proxy, the performance of a single 8GB file writing and reading were measured in each mode. In the case of writing, the input file is already in the client's RAM disk and will be written to backend storage cluster's page cache. In the case of reading, the data are already loaded into the backend storage cluster page cache and will be written to client's RAM disk. This RAM-to-RAM test eliminates the performance impact of the (old) HDD.

3.2.1 EC Client writing directly to XRootD Storage Cluster

A typical client that directly uses EC to write to and read from a vanilla XRootD storage cluster would be a xrdcp [4] loaded with the EC client plugin. It was run in the frontend XRootD proxy. The following performance numbers of reported by the xrdcp tool.

- Writing: ~904MB/s. The backend storage saw 904*(8+2)/8 = 1130MB/s. This is close to the wire limit.
- Reading: 1017MB/s. This indicates that the EC client skipped the parity chunks because the network hardware limitation would not allow this speed if the parity chunks were also read.

3.2.2 EC Client writing to XRootD Storage Cluster via an EC enabled XRootD Proxy

A client in this case doesn't load EC. It runs on another node with hardware and network configuration identical to the frontend proxy. It is also on the same subnet as the proxy.

- Writing: ~900MB/s. This is identical to the performance above, indicating that the EC-enabled XRootD proxy didn't introduce a performance penalty.
- Reading: ~95MB/s if XRootD Storage Cluster used async-IO in the root protocol, and ~155 MB/s when sync-IO was used. The speed was further increased to ~505MB/s when the memory caching function (8MB page size, one page prefetching) was enabled in the XRootD proxy.

The reading performance degradation is likely caused by the 2MB per file reading buffer allocated by the XRootD proxy. While this affects single file reading performance, it should still allow much higher aggregated throughput via parallel reading of many files. It is also possible to enable XRootD server's maxi-buffers to bypass this problem but this was not ried.

Though using memory caching and prefetching would boost the reading performance (and could be used in production), it was not turned on throughout the testing (unless indicated) because memory caching effect was not part of this studying. Turning it on would also distort the performance measurement when concurrent reading of the same file happens.

3.3 Aggregated Throughput via EC-enabled XRootD Proxy

A list of 312 randomly chosen files with size ranging from 30MB to 1.1GB was loaded to the EC prototype, to be used to test the aggregated performance. The first test used 150 batch jobs, each read these 312 file 5 times (in the same order). Fig. 1 shows aggregated reading performance with and without memory caching.

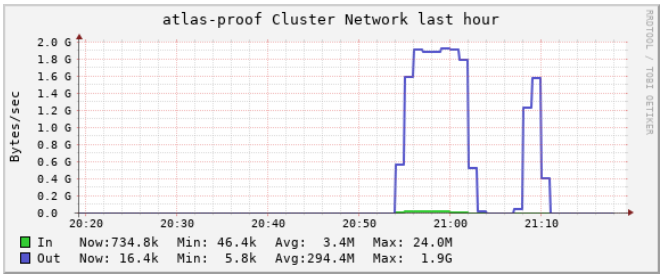

Fig. 1. Aggregated reading performance from the EC storage prototype. Network "out" means data being sent from the EC storage. The left peak was measured without using the memory cache, while the right peak was measured with memory cache turn on in the XRootD proxy cluster – the cache significantly reduced the amount of data being read from the EC storage.

The second test used 200 batch jobs, with each batch jobs randomly choosing 20 files from the list. Each file was read and then written back to another location in the EC storage prototype. Fig. 2 shows the aggregate reading and writing performance.

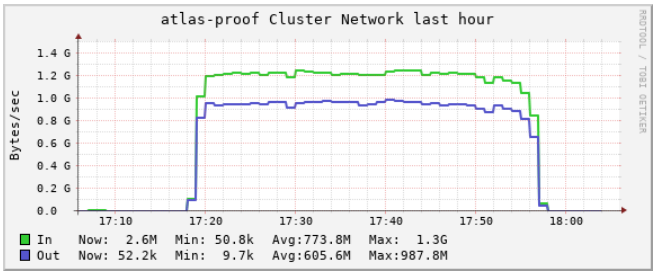

Fig. 2. Aggregated and concurrent reading ("Out") and writing ("In") performance, the memory cache was turned off.

3.4 Performance of Small File and Metadata Operations

A set of 10,000 files, each 27KB, were prepared to test the performance of concurrent get/put/delete operations. For each type of test (get/put/delete), these 10K files were dispatched to 1 or 2 or 4 or 8… concurrent clients and run against the EC enabled XRootD proxy. The clients were run on the EC enabled XRootD proxy itself instead of in batch nodes (This is because the network security setting at the SLAC computing facility does not allow high frequency creation/demolition of TCP connections, as demonstrated below). Fig. 3 shows that the performance of all three types of operations maintains constant as the number of client increase. The slow decreasing of performance toward 1000 concurrent clients could

be caused by the limitation of the backend XRootD cluster, the limitation of the EC-enabled XRootD proxy, or most likely, because a single machine (the XRootD proxy) can't efficiently handle 1000 simultaneous clients.

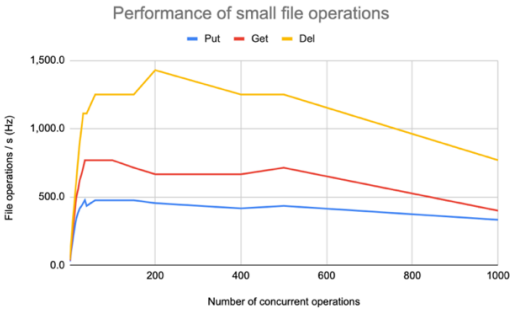

Fig. 3. Performance of small file operations vs the number of concurrent clients. The clients all run on the XRootD proxy node to avoid being throttled by the network switch.

4 Administrative Tasks and Tools

Many of the administration tasks of a XRootD storage cluster remains the same even when EC is enabled. However, there are a several EC specific tasks: when a hard disk is lost, the data is still accessible, and a manual EC-rebuild can be initiated. The process toward an EC-rebuild involves the following two steps:

4.1 Identify the Lost Files

A XRootD data node can be configured to use a name space and several data spaces on separate hardware [5]. Name space hosts symbolic links that point to data files in data space. Each data file (for EC, this refers to a zip archive file) contains an attribute pointing back to the name space. This safeguard mechanism can be used to either reconstruct the full name space, or identify what files resided on a lost data disk.

If both name space and a data space is lost, a brutal search of the whole XRootD storage cluster is needed to identity files that lost one of their zip archives. For this reason, it is important to put the name space and data spaces on separate hardware, and periodically backup the contents of the name space.

4.2 Perform the EC-rebuild

XRootDFS [6], which mounts a XRootD storage cluster as a file system, can be used for EC-rebuild. The rebuild process can be performed with three simple steps: a) copy the file to a new file, b) delete the old file and c) rename to new file to the old file name.

5 Limitation and Future Works

An Erasure Coding storage system such as this prototype prefers users to treat it as an object store rather than a Posix store. The EC implementation allows partial Posix operations. For example, files can be opened in read-only mode or write-only mode but not in read-write mode. In read-only mode, reading a data range (including Posix vector reads) is supported, though sequential reading is the most efficient. In writing-only mode, a file cannot be

overwritten or modified – an existing file has to be explicitly deleted in order to create a new file with the same name. Writing has to be sequential. These limitations suggest that generic object store get or put operations are preferred over Posix random IO pattern.

One improvement involves the mechanism described in section 2.1.3. A cache to store the file locations (a list of data nodes that have the corresponding zip archive files of a real file) will speed up repeated accessing pattern. The redirector's CMSd process has a place to cache this info, or it can be cache by a dedicated caching service.

The EC client has an internal algorithm to spread new data files in a way that will help balance the free spaces among data nodes. Turning the algorithm (to work more or less aggressively) is possible but currently require a restart.

The s3 [12] protocol is one of the populate object store access protocols. Through XRootD's HTTP plugin, it is possible to provide partial support of the s3 protocol. For example, it is possible to use a token for authentication, and perform object level operations such as get or put (though HTTP multi-part upload is not supported) or deletion. But bucket level operations are not supported because the current XRootD HTTP plugin returns information using HTML (to support web browsing) rather than XML, as required by the s3 protocol.

6 Conclusion

This working prototype demonstrated a model of integrating the Erasure Coding capability in the XRootD storage cluster. From the users' point of view, it is a standard XRootD based storage system that supports all WLCG authentication/authorization, data access and data transfer requirement. Internally, it provides cross-node EC capability for resilience, and it delivers reasonably good performance that can reach the hardware limitations. It is also relatively easy to manage. Several limitations and areas of future work have been identified and will be implemented as necessary.

References

1. A. Peters, M.K. Simon, E.A. Sindrilaru, *Erasure Coding for production in the EOS Open Storage system*, EPJ Web of Conferences **245**, 04008 (2020)
2. Intelligent Storage Acceleration Library: https://github.com/intel/isa-l
3. I.S. Reed, G. Solomon (1960), *Polynomial Codes over Certain Finite Fields*, Journal of the Society for Industrial and Applied Mathematics, 8 (2): **300–304** (1960)
4. Xrdcopy, https://xrootd.slac.stanford.edu/doc/man/xrdcp.1.html
5. Section 4.7.1, *Open File System & Open Storage System Configuration Reference*, https://xrootd.slac.stanford.edu/doc/dev56/ofs_config.htm#_Toc136617321
6. W. Yang, A. Hanushevsky, *XrootdFS: A Posix Filesystem for Xrootd*. J. Phys. Conf. Ser. **898** 062046 (2017)
7. Section 4.5.1, *XRootD Cluster Management Service Configuration Manual*, https://xrootd.slac.stanford.edu/doc/dev54/cms_config.htm#_Toc53611086
8. CERN: https://www.cern.ch
9. The SLAC National Accelerator Laboratory: https://www.slac.stanford.edu
10. WLCG: The Worldwide LHC Computing Grid: https://wlcg.web.cern.ch
11. XRootD: https://www.xrootd.org
12. *Amazon Web Services Launches "Amazon S3"* (Press release). 2006-03-14, https://press.aboutamazon.com/2006/3/amazon-web-services-launches