XRootD Client: a robust technology for LHC Run-3 and beyond

G. Amadio[1], C. Caffy[1], A. B. Hanushevsky[2], M. K. Simon[1], D. Smith[1] for the XRootD Development Team
[1]CERN, [2]SLAC

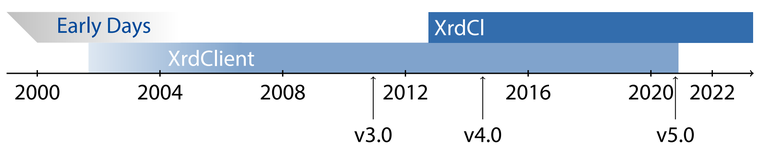
Introduction

The challenges for Run-3 of the LHC, which began in 2022, include managing the massive amounts of data generated by the experiments and ensuring their efficient storage, access, and analysis. In order to address these challenges, XRootD is now widely used by the HEP community. XRootD supports multiple data access models, including hierarchical, file-based, and object-based access, and it can operate on various types of storage systems, including disk, tape, and cloud storage. It is designed for scalable, high-performance, and fault-tolerant data access, transfer, and management.

XRootD has become a critical component in the data management strategy of the LHC along its history, which spans more than 20 years. In the early days, it was developed as a replacement for `rootd` within ROOT, therefore its name. In the early 2000s, the code was moved into its own repository, and XRootD became an independent project.

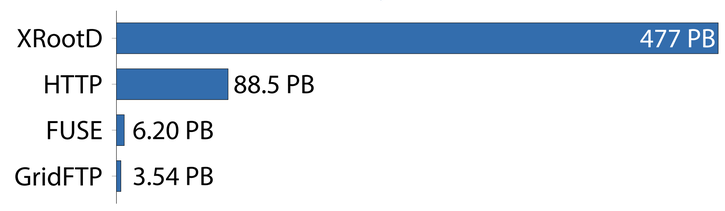
The XRootD Client

The first production version of the XRootD client, *XrdClient*, was added to the repository in Sep 2004 by Fabrizio Furano. In the second half of 2012, a newly rewritten version of the client, *XrdCl*, was introduced by Lukasz Janyst. Its debut happened in XRootD 4.0, at which point *XrdClient* was declared obsolete, and in XRootD 5.0 the code was finally removed from the repository.

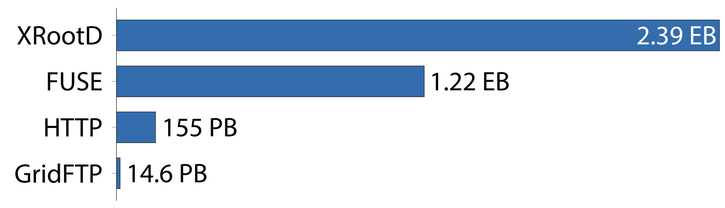


The XRootD client is the main tool used to transfer experimental data into and out of the data center at CERN. In 2022, over 570 PB of data have been written into EOS physics instances by the main LHC experiments. The XRootD protocol accounted for the majority of this amount, as it can be seen in the statistics data shown below.

Data Written to EOS Physics Instances in 2022



Data Read from EOS Physics Instances in 2022



Whilst the volume of data written is already massive, it is dwarfed by the amount of data read for analysis. In the same year of 2022, about 3.78 EB of data have been read from the data center, 2.4 EB of which using the XRootD protocol, either by the standalone XRootD client, `xrdcp`, or via XRootD client code integrated into other applications, such as ROOT. Moreover, since the EOS FUSE client also relies on the XRootD client code in its implementation, it means that reads via FUSE are also served by XRootD.

XRootD Client Primer

The standalone client for transferring data via the XRootD protocol is `xrdcp`. Below we demonstrate how to use some of its main features.

Basic Usage

```
# Copy local file to a remote server
$ xrdcp file.root  root://example.cern.ch//path/to/destination/
# Copy remote file to the local disk
$ xrdcp root://example.cern.ch//path/to/source/file.root  .
# Third-party copy, client triggers copy from server to server
$ xrdcp --tpc only root://src.cern.ch/file.root root://dest.cern.ch/data/
# Copy while preserving extended file attributes
$ xrdcp --xattr root://src.cern.ch//file  root://dst.cern.ch//path/
# Retry if an error occurs
$ xrdcp --retry root://src.cern.ch//file  root://dst.cern.ch//path/
# Continue timed out transfer
$ xrdcp --continue root://src.cern.ch//file  root://dst.cern.ch//path/
# Limit maximum transfer rate to 150MB/s
$ xrdcp --xrate 150m root://src.cern//file  root://dst.cern//path/
# Ensure minimum transfer rate of 50MB/s
$ xrdcp --xrate-threshold 50m root://src.cern//file root://dst.cern//path/
```

Encryption with TLS

```
# Enable encryption by using roots:// or xroots:// protocol
$ xrdcp roots://example.cern.ch//path/to/file .
# Encrypt only the control channel, data is not encrypted
$ xrdcp --tlsnodata roots://example.cern.ch//path/to/file .
# Allow falling back to no encryption, for backward compatibility
$ xrdcp --notlsok roots://example.cern.ch//path/to/file .
# Use encryption with metalink files
$ xrdcp --tlsmetalink roots://example.cern.ch//path/to/file.meta4 .
```

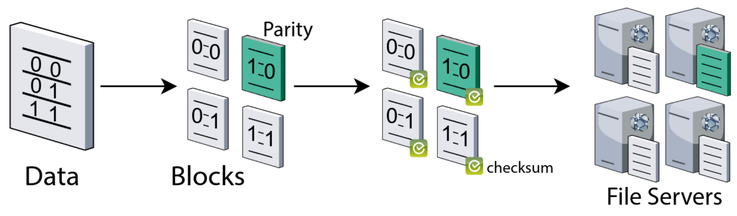
Working with ZIP Archives

```
# Copy file from remote ZIP archive to local disk
$ xrdcp --zip file.root  root://src.cern.ch//archive.zip  .
# Append a local file to a remote ZIP archive
$ xrdcp --zip-append file.root  root://dest.cern.ch//path/archive.zip
```

Erasure Coding Plugin (XrdEc) v5.2

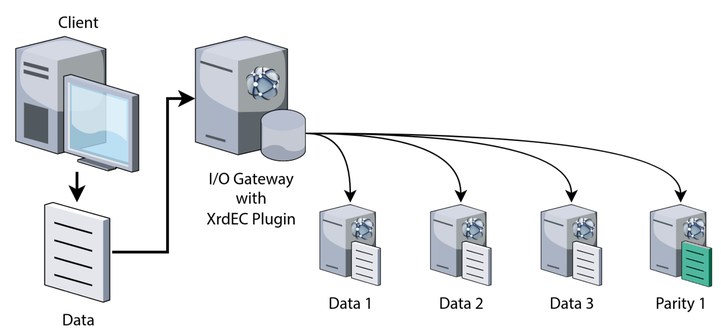
Running the LHC is expensive, hence the possibility of data loss has to be minimized. Traditionally, data durability has been achieved by replicating data at CERN as well as distributing copies across the world within the Worldwide LHC Computing Grid (WLCG). This strategy, however, may become prohibitively expensive at the increased data rates to be produced by the High-Luminosity LHC (HL-LHC). The solution, implemented originally for EOS and later in XRootD, is to use erasure coding to provide redundancy at much smaller overhead than simple data replication. Erasure coding works by dividing data into chunks and adding parity blocks that can be used to reconstruct the original data in the event of hardware failure.

Erasure Coding in an Nutshell



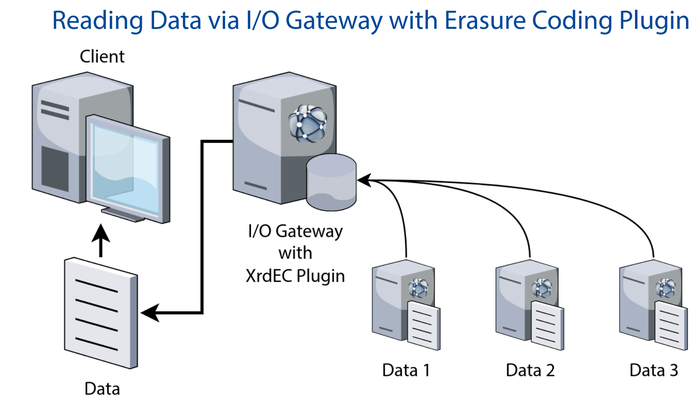
When configured to use erasure coding, the server will automatically split files into *n* data blocks and *k* parity blocks according to the configuration and distribute each piece to a different disk or file storage server as depicted below.

Writing Data via I/O Gateway with Erasure Coding Plugin

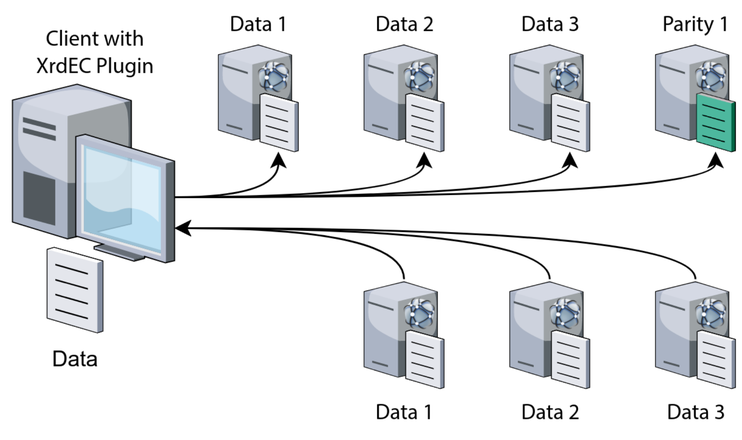


When reading, the data can be reconstructed either at the server, in case the client does not have support for erasure coding enabled, or each block can be read directly if the client has support for erasure coding enabled. The client can also write data directly in this case.

Reading Data via I/O Gateway with Erasure Coding Plugin



Client with Erasure Coding Plugin can Read/Write Data Directly



Declarative Asynchronous API v5.0–v5.4

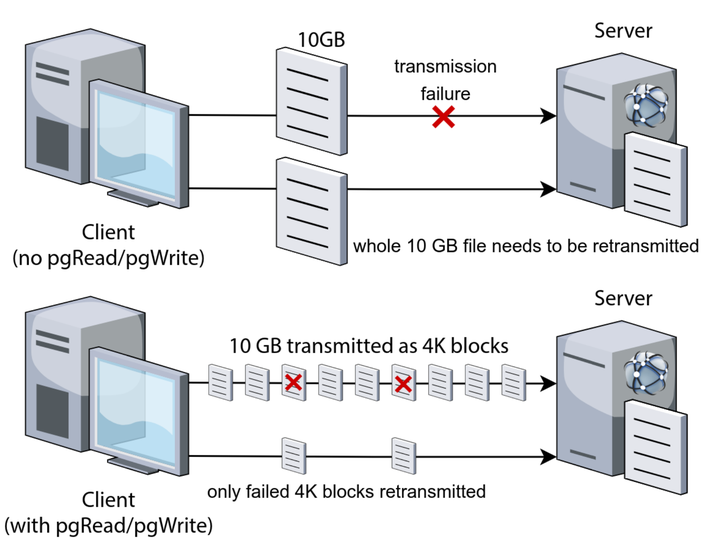
The declarative API for the XRootD client has been created with erasure coding support as its main use case. Its main objective is to simplify composition of a series of asynchronous operations on one or more remote files, such as writing a data block striped into *n* data chunks and *k* parity chunks in parallel. The advantage of the declarative API is that it allows composing asynchronous operations without requiring error-prone boiler plate code in the process.

```
using namespace XrdCl;

void ECWrite(uint64_t offset, uint64_t size, const void *buffer,
             ResponseHandler *handler)
{
  /* calculate number of chunks */
  std::vector<Pipeline> writes(nchunks);
  for (size_t i = 0; i < nchunks; ++i) {
    /* calculate offset, size, and buffer for each chunk */
    File f = new XrdCl::File();
    Pipeline p = Open(file, url, flags)
      | Parallel(  Write(file, chunk_offset, chunk_size, chunk_buffer),
                   SetXAttr(file, "xrdec.cksum", checksum) )
      | Close(file) >> [file](XRootDStatus&) { delete file; };
  }
  Async(Parallel(writes) >> [handler](XRootDStatus&) {
    handler->HandleResponse(new XRootDStatus(), 0);
  });
}
```

Data Integrity (pgRead/pgWrite) v5.0–v5.5

Disk failures can cause corruption for data at rest, but transmission errors while the network is under heavy load may also lead to data corruption across the wire. For small file transfers, this sort of corruption is not a problem, since the cost of retransmitting the data is low in case a checksum mismatch occurs at the end of the transfer on the destination. However, when transferring large files (>10GB), a better mechanism is necessary. Therefore, in addition to data durability features like erasure coding for data at rest, XRootD has also introduced pgRead and pgWrite to ensure data integrity for data tranfers across the network. With pgRead/pgWrite, data is checksummed at 4K page boundaries, and if any transmission errors occur, only pages with mismatched checksums need to be retransmitted, greatly improving reliablility.



Wire Layout for Data Transfers with pgRead/pgWrite

Read/write 6144 bytes at offset 0 (page aligned)

Offset 0		Offset 4096	
CRC32	**4096 bytes (1 page)**	**CRC32**	**2048 bytes**

Read/write 8000 bytes at offset 2040 (typical random I/O)

Offset 2040		Offset 4096		Offset 8192	
CRC32	**2056 bytes**	**CRC32**	**4096 bytes (1 page)**	**CRC32**	**1848 bytes**

Read/write 4000 bytes at offset 2040 (degenerate case)

Offset 2040		Offset 4096	
CRC32	**2056 bytes**	**CRC32**	**1944 bytes**

Record/Replay Plugin v5.5

The recorder plugin for the XRootD client, introduced in XRootD 5.5.0, allows users to record remote data access patterns in way that is transparent to client applications. It can be enabled by the user by creating a configuration file in its home directory at `$HOME/.xrootd/client.plugins.d/recorder.conf` with the following contents:

```
url = *
lib = /usr/lib64/libXrdClRecorder-5.so
enable = true
output = /tmp/xrdrecord.csv
```

This configuration will instruct the XRootD client to load the recorder plugin and record each operation into the `/tmp/xrdrecord.csv` file on disk. A simple example is shown below using ROOT to run an RDataFrame tutorial that reads CMS opendata via XRootD.

```
$ root.exe -l -b -q df102_NanoAODDimuonAnalysis.C

Processing df102_NanoAODDimuonAnalysis.C...
Info in <TCanvas::Print>: pdf file dimuon_spectrum.pdf has been created
Events with exactly two muons: pass=31104343 all=61540413 eff=50.54
Muons with opposite charge: pass=24067843 all=31104343 eff=77.38
```

The output file can be inspected and replayed with the `xrdreplay` command line tool. Without any options, it will read the CSV file and execute again the same operations, taking care of reproducing the timings from the original run as well. When run with the `-p` option, it produces a summary of the operations performed by the client:

```
$ xrdreplay -p /tmp/xrdrecord.csv
# ================================================
# IO Summary (print mode)
# ================================================
# Sampled Runtime  : 23.195940 s
# Playback Speed   : 1.00
# IO Volume (R)    : 2.24 GB [ std:581.69 KB vec:2.24 GB page:0 B ]
# IO Volume (W)    : 0 B [ std:0 B vec:0 B page:0 B ]
# IOPS      (R)    : 147 [ std:72 vec:75 page:0 ]
# IOPS      (W)    : 0 [ std:0 vec:0 page:0 ]
# Files     (R)    : 18
# Files     (W)    : 0
# Datasize  (R)    : 40.40 GB
# Datasize  (W)    : 0 B
# ------------------------------------------------
# Quality Estimation
# ------------------------------------------------
# Synchronicity(R) : 100.00%
# Synchronicity(W) : 0.00%
```

In the summary above, we can see that ROOT read 2.24 GB of data in 23.2 s from the remote file, mostly using vector reads.