C++ Declarative API – Implementation Overview Within the XRootD Framework

[redacted]
[redacted]
[redacted]
[redacted]
[redacted]

[redacted]
[redacted]
[redacted]
[redacted]
[redacted]

***Abstract*—A brief description of the XRootD architecture and its purpose within the Worldwide Large Hadron Collider Computing Grid (WLCG), alongside an overview of the server- and client- sides of the XRootD framework are discussed in the present work. The client-side of XRootD has a relatively new feature called Declarative API. Its main objective is to provide the user with an asynchronous interface that is more in line with the modern C++ paradigm. A discussion on the development process for the new API is made, together with a case study that involves the implementation of an Erasure Coding plug-in for the client.**

Keywords—XRootD, asynchronous programming, declarative API, pipeline, server, client.

I. Introduction

Started as a protocol which granted remote access to root format specific files, with a primary use case focused on data analysis (rather than data transfer), XRootD became widely used within the scientific community at CERN (European Organization for Nuclear Research) and other large-scale facilities (e.g., SLAC-Stanford Linear Accelerator Center). Over the last years, the framework evolved a lot, and it now supports data analysis, data transfers, data management, and also features like staging data from tape.

In terms of storage capacity, only the ATLAS and CMS collaborations alone produce a total of around 150 Petabytes of data which needs to be accessed by thousands of physicists within the Worldwide Large Hadron Collider Compute Grid - WLCG community [7]. As a result, a key objective of the WLCG is to assure both the process of moving the data between sites and deliver the data to any end-user application.

The addition of XRootD on top of the Anydata, Anytime, Anywhere project (or AAA for short) [2] allowed the high Energy Physics community to achieve global data storage federations that have a single data-access entry point and also a common data-access protocol, which changed the old paradigm of distributed multi-tiered storage. This is possible through the hierarchical deployment of redirectors that allow real-time discovery of sites that are hosting the data. Although XRootD supported multi-storage deployments for a long time, the addition of a feature that allowed its proper functionality within a global, multi-site environment was in fact the core idea of AAA.

To emphasize the importance of XRootD, it is worth mentioning that currently at CERN, the main storage solution is EOS – a technology developed in-house and built on top of the XRootD framework. The LHC experiments (e.g. ATLAS, CMS, LHCb, ALICE) and also other smaller experiments hosted at CERN (e.g. AMS), including their user communities, use EOS for data storage and access. In the following section, we provide a clearer picture of the XRootD framework, both in terms of its server-side as well as its client-side, since both implementations are crucial in understanding the overall workflow of data access and data manipulation within the WLCG community.

II. XRootD framework

The main objective of any scientific project that is based on experiments that ran at CERN (but also to the other places within WLCG) is the access to the computing resources which are used for submitting jobs that aim to solve a particular task. An old model of such a workflow is described in Fig.1 (also called "jobs go to data" paradigm [2]).

Whenever the user wants a local copy of the data that is studied, a change of workflow must be made from using the data analysis tools to the usage of data transfer toolset (which is usually provided by the experiment's computing facility or the grid middleware [2]). This introduces significant overhead and as a result, slows down the process of tackling the actual tasks which have to be performed by the physicists with the required data. A Federated storage system is the implementation that aims at solving such issues. Defined in [2] as a collection of unpaired storage resources that are managed by a set of domains that are cooperating (but also independent) and also are accessible via a common namespace. By using multiple dedicated XRootD servers at each site and a centralized redirector (an in-depth description of an XRootD redirector can be found in [2]), it is possible to build these storage spaces where the user makes direct contact with the central endpoint and is redirected to a site which can provide the necessary data.

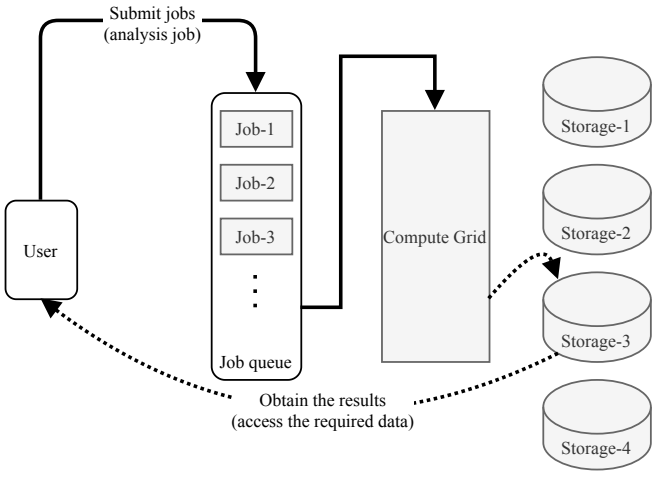

Fig. 1. Process of accessing data in the "jobs go to data" mode. User submits an analysis job that is sent through the compute grid to the sites that host the data.

A. Server-side XRootD

At its core, XRootD acts like any remote data server, however, it does seem innovative in terms of its scalability, robustness, fault tolerance, job recovery (in case of job failing during an execution), cache mechanism, and many more. A tremendous work (progress) has been done in recent years, especially for extending the scalability features (e.g., [1], [8]), caching [2], and many more. The development team is constantly committing new or improved features, and these are documented on the official repository [10]. Other characteristics of an XRootD server that assure a high-performance data availability are load-balancing, optimal use of hardware resources (like sockets, memory, CPU), cooperation between different (XRootD) servers, minimize the number of jobs that have to be restarted due to a server or network problem.

The XRootD server architecture is composed of four main layers, namely: Network layer, Protocol layer, File-system layer, and a Storage layer (see Fig. 2). Being developed on a run-time plug-in mechanism, new features can be added to the framework with little effort. The main reasons for developing the XRootD within a layered model are the optimization of specific functionalities while minimizing resource usage and isolation of services allows for dynamical loading (determining in this way which implementation works best in a particular environment [1]).

III. The XrdCl Implementation

The XrdCl implementation developed within a multi-threaded C++ paradigm within the XRootD client and it is based on a concept called event-loop [6]. XrdCl is provided by the libXrdCl library which is part of the client source code [10]. The entire C++ API for the XRootD framework is available on the official documentation page [11] – provided by the development team.

A fully asynchronous implementation is available for XrdCl (each asynchronous call has been implemented into the corresponding synchronous version of it). This is a great achievement for the XRootD framework, as having asynchronous behavior of the function calls for requesting/service data access to users greatly improves performance (however, at the cost of a slight increase in code complexity).

All the requests submitted by the user within the application are queued and then executed by the socket event-loop. This execution of requests is done in a single-threaded manner – sequentially - although there is a possibility to increase performance by upping the number of event-loops.

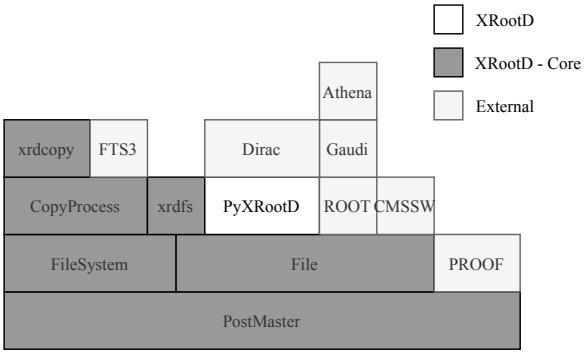

Fig. 3. Structure of the XrdCl API stack.

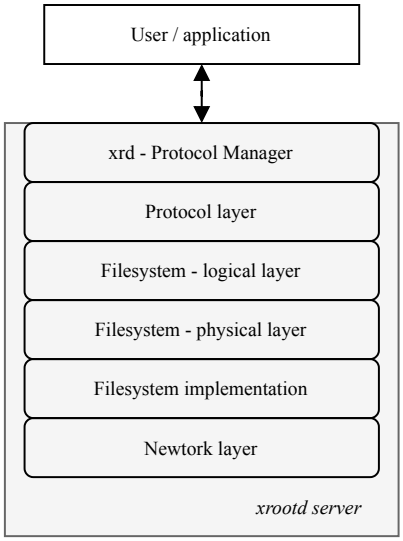

Fig. 2. XRootD server architecture.

On the other hand, all of the incoming responses are processed by the event-loop, but the response handlers are executed in a so-called thread-pool [6]. XrdCl is flexible, meaning that it can be configured using a configuration file with environment variables or directly from the C++ API.

A. The event-loop in XRootD

A user might want to retrieve some data using the XRootD client from a file that is located on a server. Interaction between the XRootD client and that particular server is done through a TCP implementation. The mechanism that allows the client to receive feedback from the TCP kernel is called event-loop. The feedback consists of communicating whether there is available space in the TCP-output buffer for writing data (i.e., requests which will be sent to the server) or if there is some data in the TCP-receive buffer for reading responses from the server. In this event-based workflow, there is a queue of requests that the client is issuing to the server, and with each write-event, a request is removed from the queue and it is being written on the socket. It is worth mentioning that the TCP buffers (for both sending and receiving data) might not have enough size to allow requests/responses to be written/read in a single event, meaning that it can take several write/read events to process an entire request to the server or a response from the server. Furthermore, each request has a corresponding message handler, so that after a request is written to the socket (in order to be sent to the server), the accompanying message handler is moved into a queue for incoming responses. During a read-event yielded by the event-loop, the client is informed that it can read from the socket, that is a server response. Once a response arrived from the server, its corresponding message handler (located inside the incoming queue) is also taken out from the queue, and finally, after the response is parsed, the callback function is being called. In terms of processing stages, the event-loop generates the following events: *ready-to-read*, *ready-to-write*, *read-timeout,* and *write-timeout*. According to each of the event types, the client is sending requests, is receiving responses, and handling timeouts. Fig. 4 aims to give a schematic representation of the entire workflow.

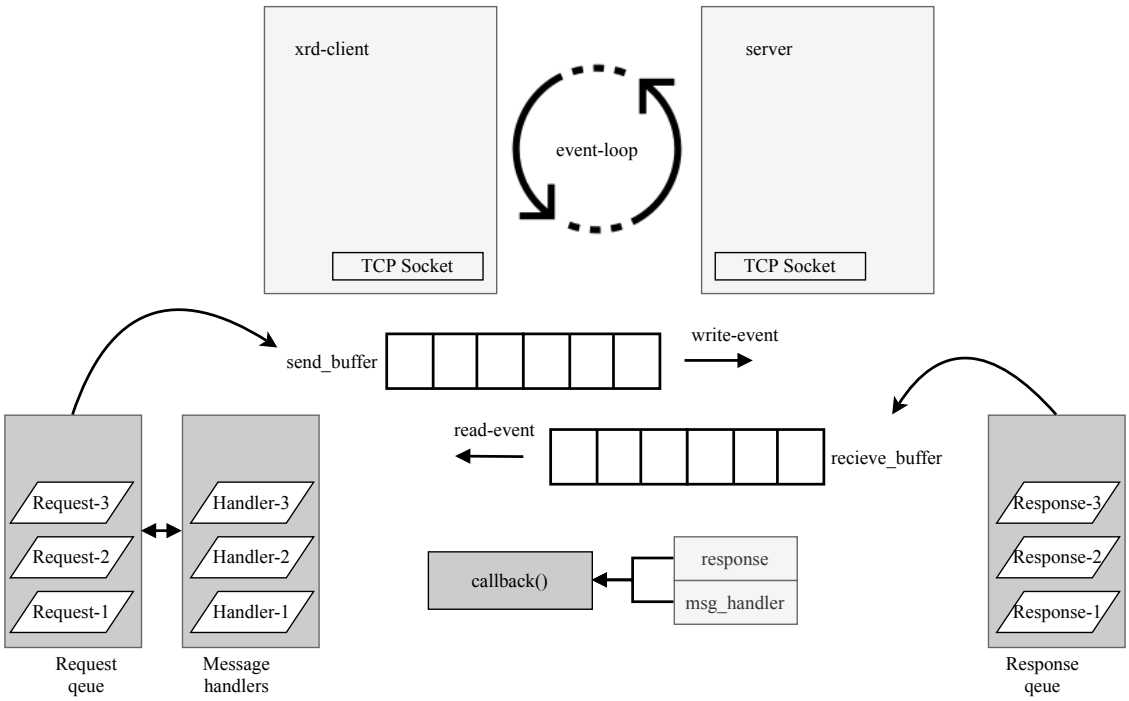

Fig. 4. The event-loop in XRootD.

XrdCl hides any of the low-level connection handling from the user. The workflow of an application starts with a request which is issued as a connection between the client and the server. Once the connection has been automatically established, it will be kept alive for further usage, until the time-to-live timeout elapses. The XrdCl implementation is routing all the requests through a single (physical) connection, although, it is possible to force the component to use up to 16 simultaneous physical connections, improving in this way the performance over a network of WAN type. Disconnection of the client from a server can be forced, depending on the actual needs; for example, the user might want to re-establish the connection with new credentials. In fact, the client authenticates (on server request) during the XRootD handshake that happens when the connection is being established.

The entire XrdCl API stack is represented as a block diagram in Fig. 3, where each of the components is organized in three main categories: XRootD-Core, XRootD, and External.

```
XRootDStatus File::Open(const std::string &url,
                        OpenFlags::Flags   flags,
                        Access::Mode       mode,
                        uint16_t           timeout)
```

Listing 2: Synchronous version of the Open method.

```
XRootDStatus File::Open(const std::string &url,
                        OpenFlags::Flags   flags,
                        Access::Mode       mode,
                        ResponseHandler   *handler,
                        uint16_t           timeout)
```

Listing 1: Asynchronous version of the Open method.

Since only the File and FileSystem objects within the XrdCl stack are part of the focus within the current work, it is worth mentioning some key characteristics of these implementations.

The XrdCl library is the foundation part of the following components of the XRootD client:

- The command-line interface [6].
- Python bindings: designed to facilitate the usage of the XRootD client, by writing Python instead of having to write C++ [9].
- SSI Client: a multi-threaded XRootD plug-in that implements a request-response framework [12].
- The POSIX API.
- New: C++ Declarative Client API (with release of XRootD v.4.9.0).

B. Asynchronous implementations within XRootD client interface

The fact that XRootD is mainly used with file-based data repositories, a crucial component is indeed the file access API. It was already mentioned that these objects have both synchronous and asynchronous behavior. What this means from an API standpoint is that the asynchronous functions

```
class Handler: public ResponseHandler
{
public:
 void HandleResponse( XRootDStatus *status,
                      AnyObject *response)
 {
  Response *res=GetResponse<Response>(status,
                                      response);

  //Perform operations using the response object

  delete status;
  delete response;
  delete this;
}
```

Listing 3: An example with a handler class implementation.

within File and FileSystem objects take one extra argument: the so-called handler objects (see Listings 1 and 2 for the core difference).

The handler object implements specific interfaces, and its methods are called whenever the response from the asynchronous function call arrives. An example for such a response handler object is given in the Listing 3.

Whenever the client uses the asynchronous function, it is called within the call stack and ran in the background. This means that the program does not have to wait for the function execution (asynchronous calls => non-blocking operations). It is in fact the response handler that takes care of the function callback once it has been executed; in other words, the handler controls the proper flow of the execution pipeline. The flow of operations follows works in such a way that each next function from the pipeline needs to be called within the handler of the previous function.

A usual execution flow that involves file operations might consist of a function that tries to open the file (e.g. Open). Its response handler must have the second operation (e.g. Read, Write) that needs to be called. The second function has a handler as well, which will eventually call the third operation (usually the closing operation on that specific file which the client has accessed; Close). It is also worth mentioning that each operation call requires different arguments and as a result, each handler will have its corresponding implementation. It is relatively easy to see that even a simple flow requires some serious amount of work, and this will scale up with each function that the client has to execute.

The asynchronous API can become quite cumbersome, especially in terms of code readability. In order to emphasize this, in Listing 4 an example is shown. Within that snippet, the first operation call is the Open method, while further execution of the flow of operations will take place in the handler. The user must go through an entire set of handlers, which could be time consuming and inconvenient.

```
const string path = "/path/to/testFile.dat";
const OpenFlags::Flags flags = OpenFlags::Read;
const Access::Mode mode = Access::None;

auto openHandler = new CustomOpenHandler();
File *file = new File();

file->Open(path, flags, mode, openHandler);
// Further execution in handler: Read->Close
```

Listing 4: Asynchronous implementation for reading a file within XrdCl.

Fortunately, a newly introduced implementation within the XrdCl framework aims at simplifying an asynchronous operation pipeline: Client Declarative API [6]. This new feature will be discussed in the following section.

IV. The Client Declarative API

Introduced in version 4.9.0 of the XRootD package, the Declarative API [6] was designed and implemented to facilitate the usage of XrdCl API by the users. It has been built on top of the existing API and provides an additional layer of abstraction (that layer itself is what makes a more convenient interface between the client and the end-user). This section is dedicated to describing the main parts and characteristics of the implementation.

Its key features that make the API easy to use are the following:

- Define an entire operation workflow in a contained space, without the need of fragmenting the logic into many classes and implementations.
- The syntax is declarative-centric, meaning that users should focus on the actual choice of operation rather than paying much attention (effort) to the execution flow.
- Proper signaling for the user of any incorrect declarations and configurations during the compilation phase.
- Error handling for the workflow is done consistently, showing proper error messages within the same space where the workflow is done.

The constructed API makes it so there is a communication protocol between the operations: the result of one operation is used to compute the following operation, making this implementation very robust. Listing 5 shows an example of operations workflow using the Declarative approach. One can see that the new API is more in line with the modern C++ language paradigm.

```
File *file = new File();

auto readHandler = new ResponseHandler();

// open, read from and close the file
auto &pipeline= Open(file)(path, flags, mode)
              | Read( file )(offset , size ,
                      buffer) >> readHandler
              | Close( file ) () ;

auto status = WaitFor(p);
```

Listing 5: Declarative syntax within XrdCl.

From Listing 5, one can see that the flow of file operations is assigned to a pipeline variable. For each operation (namely Open, Read, and Close) the corresponding object is created and the operator () is used for passing arguments to the operations. The operation itself dictates the number of arguments and their types. A handler is introduced after the second operation through the >> operator, although this is optional, as the operation handler is not controlling the flow anymore. The example shown in Listing 5 only has only one handler, that is for the Read function; the last line contains a utility for synchronous execution of the pipeline (current thread waits until the entire pipeline chain of operations is finished). The defined operations are connected to each other by the | operator. It is worth mentioning that all these implementations added within the Declarative API are possible because of the operator overloading feature of the C++ programming language.

The syntax also supports parallel execution of multiple flows of operations (XrdCl::Parallel implementation is part of the Operation Utilities within the Declarative API toolset).

```
auto &&o1   = Open(file1,path1,OpenFlags::Read);
auto &&o2   = Open(file2,path2,OpenFlags::Read);
auto &&o2   = Open(file2,path3,OpenFlags::Read);


// open 3 files in parallel
Pipeline p  = Parallel(o1,o2,o3);

auto status = WaitFor( p );
```

Listing 6: Parallel operations in XrdCl. Execution of multiple functions.

The Parallel utility aggregates several operations (those might be compound operations) for parallel execution. It also accepts a variable number of operations. An example pipeline with three parallel operations can be seen in Listing 6. Even more so, it is possible to have pipelines run in parallel. Such an example is shown in Listing 7.

```
auto &pipe1 = Open(file1)(path1,flags)
               | Read(file1)(offset,size,buff1)
               | Close(file1)();

auto &pipe2 = Open(file2)(path2,flags)
            | Read(file2)(offset,size,boff2)
            | Close(file2)();

auto &pipe = Open(lockFile)(lockFileURL,flags)
           | Parallel{&firstPipe,&secondPipe}
           | Close(lockFile)();
```

Listing 7: Parallel operations in XrdCl. Execution of multiple pipelines.

A. *Pipeline semantics*

In order to emphasize the overall flexibility and fluidity of the pipeline syntax, the following example is proposed: the user wants to access a file with a size of 0.5MB from a data batch on a server. Using the declarative approach, the procedure will look like Listing 8. The first line is for the declaration of a lock file, then the lock file is created with the first call of the Open function (taking as an argument the lock file itself). Once the lock file has been created, the pipeline continues by doing an open, a read, and a close for the actual file that needs to be accessed. The Rm function is used for deleting the lock file since it is not needed anymore.

```
File lock, file;
FileSystem fs(url);
std::future<ChunkInfo> resp; // server response

auto &&p = Open(lock,"root://host//path/to/.lock
", OpenFlags::New)
         | Close(lock)
         | Open(file, "root://host//path/to/file
.txt",OpenFlags::Read)
         | Read(file,0,512,buff) >> resp
         | Close(file);
         | Rm(fs, "root://host//path/to/.lock");


// wait for the pipeline to complete
auto status = WaitFor( p );
```

Listing 8: The pipeline semantic in XrdCl.

What is important to note is that if an operation fails to complete execution, any subsequent operations within that pipeline will not be executed, but their handlers will be called (with error status). Using the pipelining semantic makes the control flow clearer and more robust.

B. *Implementation of a plug-in for the XRootD client using Declarative API*

The Declarative API is tested in the development of an Erasure Coding plug-in for the client. Erasure Coding (EC) is a method of data protection in which data is broken into fragments, expanded and encoded with redundant data pieces, and stored across a set of different locations or storage media.

The goal of erasure coding is to enable data that becomes corrupted at some point in the disk storage process to be reconstructed by using information about the data that's stored elsewhere in the array. The drawback of erasure coding is that it can be more CPU-intensive, and that can translate into increased latency. In other words, erasure coding adds the redundancy to the system that tolerates failures. In terms of the workflow, EC takes the original data and encodes it in such a way that when needed, only a subset of all the chunks is required to recreate the original information. The data protection scheme is graphically represented in Figs. 5 and 6, where the decode and encode procedures, respectively, are explained.

Developing the EC procedure will imply that an entire block of data will be stripped into *N* data chunks and *K* parity chunks. One needs to open all stripes, write to all stripes, set extended attributes on all stripes (e.g., checksums), then finally close all stripes. The write operation and setting extended attributes should be done in parallel. In order to understand why the Declarative API simplifies the workflow, it is worth mentioning how the standard XrdCl asynchronous API would manage the entire EC pipeline.

To update a single chunk of data, one would have to write a function that opens, writes, sets extended attributes, and closes a file asynchronously. However, the write procedure (together with the following chain of operations) will be hidden in the callback of the handler corresponding to the Open() function. Furthermore, writing to the file, setting extended attributes, and closing the file would each have a handler that is taking care of the execution process. Keep in mind that this entire workflow is only for one chunk, and the functions for writing and setting extended attributes execute sequentially. One would need a handler-class to aggregate these procedures in a parallel execution. Updating the data stripes and parity stripes will also require a handler class in order to have a parallel execution. This entire workflow induces a lot of boilerplate for the user (when creating the required flow of operations), and it makes a repetitive process (by requiring handlers and handler-class construction when trying to execute in parallel). A clear workflow is hidden from the user, since the callbacks are embedded into the first handler operation: one needs to go through the entire set of handlers in order to understand the full execution pipeline. Using the Declarative API, the amount of code boilerplate is significantly reduced when comparing the standard

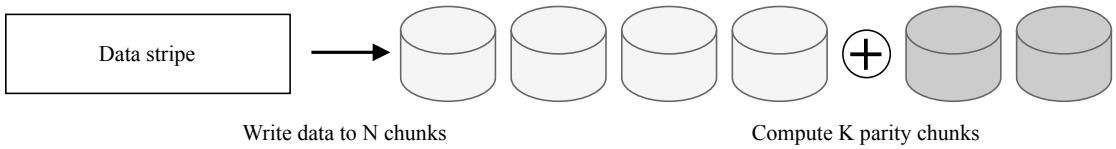

Fig. 5. Erasure coding: decode scheme, with N=4 chunks and K=2 parity chunks.

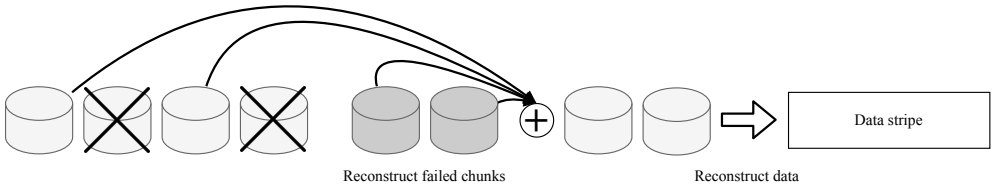

Fig. 7. Erasure coding: encode scheme, with 2 failures in the storage grid.

asynchronous operations. The process of writing the plug-in achieves a high degree of code readability, with a clear workflow and reduced complexity. The standard asynchronous operations hide the actual workflow of operations behind the first function callback (e.g. in the Open->Read->Close pipeline, the entire workflow is hidden in the callback of the Open() function). Fig. 7 describes the entire flow of operations (including the parallel execution of the write to each chunk and setting extended attributes).

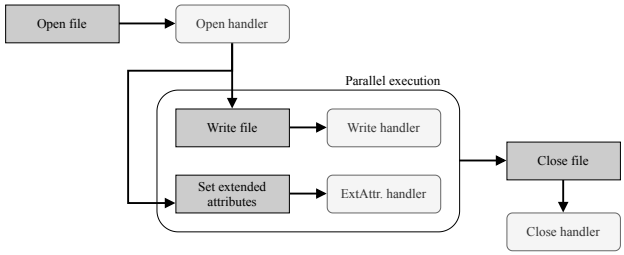

Fig. 6. The flow of operations to be executed when the client starts an Erasure Coding procedure.

Listing 9 contains the necessary workflow, reduced to a straightforward execution pipeline. The pipeline variable contains the composition of operations (constructed with the pipe "|" operator). Within this procedure, the parallel execution of operations is also constructed with the Parallel() function, with the arguments that write and set attributes to each data chunk. All the operations are finally executed asynchronously, with the callbacks marked by the stream operator ">>".

```
//Erasure
void ECWrite(uint64_t  offset,
                       uint32_t  size,
                       const void *buffer,
                       ResponseHandler
*userHandler)
{
    std::vector<Pipeline> writes;
    writes.reserve(n_chunks);
    for(size_t id=0;id<n_chunks;++id)
    {
        //compute offset, size and buffer for
each stripe/chunk
        File *file=new File();
        Pipeline p=Open(file, url, flags)
                        | Parallel(Write(file,
chunk_offset, chunk_size,chunk_buffer)),

SetXAttr(file, "xrdec.chksum",checksum))
                       | Close(file) >>
[file](XRootDStatus &){delete file};}
    }
    //execute the workflow asynchronoulsy
    Async(Parallel(writes))>>
[userHandler](XRootDStatus & status)

{ userHandler->HandleResponse(new
XRootDStatus(status),0);});
}
```

Listing 9: Implementation of the Erasure Coding plug-in with Declarative API

V. Conclusions

In the present work, a detailed overview of the XRootD framework was given, together with its major importance within the WLCG group and the High Energy Physics community. A short description of the architecture for both the server-side as well as the client-side was discussed. The asynchronous behavior of the XrdCl API which is written in C++ has been reviewed, with the latest features and release. Attention was focused on the File and FileSystem objects within the XRootD client. The asynchronous API's importance in terms of usage has been mentioned and also the drawbacks in terms of code complexity. The next topic was devoted to the Declarative API, which is built on top of the existing XrdCl asynchronous API and its main feature is the ease of use from a code-logistic standpoint. The Declarative API was also put into usage with the implementation of an Erasure Coding plug-in inside the XRootD client. It is showed that Declarative API is an efficient tool in providing an asynchronous C++ interface for the user while keeping a clear and concise workflow.

Acknowledgments

The first author is very grateful to the entire IT department from CERN, especially Michał, who had the time and patience for providing help whenever required and many clarifications throughout the collaboration. Special thanks also go to the Department of Computation Physics from Magurele (i.e. Mihnea Dulea – head of the department, and Ionut Vasile who provided the computational resources that were required). This work was possible through the CONDEGRID project.

Appendix

The asynchronous workflow for a chain of operations was discussed in Section III. The diagram in Fig. A1 aims at giving a schematic representation of the pipeline, including the concept of response handler.

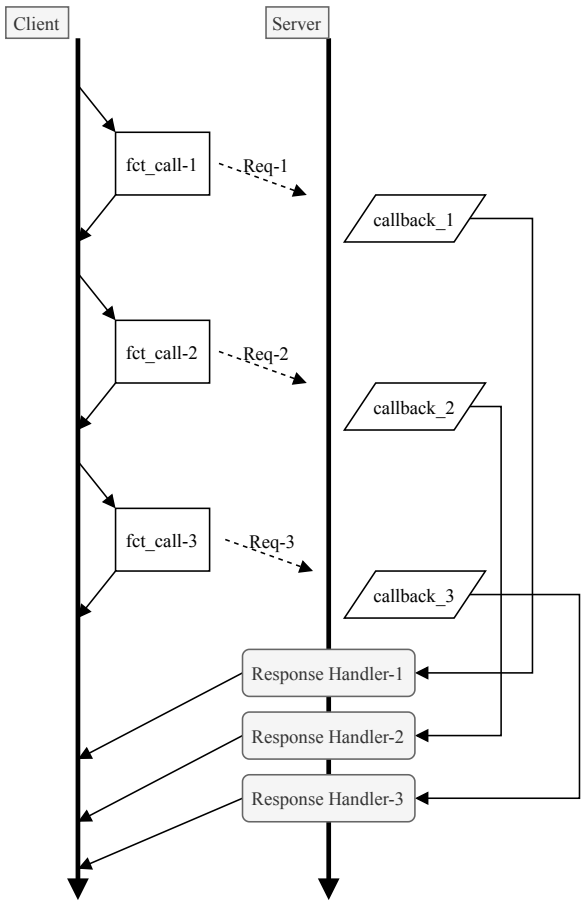

Fig. A1: Asynchronous workflow in terms of response handlers.

REFERENCES

[1] Dorigo, A., Elmer, P., Furano, F., & Hanushevsky, A. (2005, March). XROOTD/TXNetFile: a highly scalable architecture for data access in the ROOT environment. In Proceedings of the 4th WSEAS International Conference on Telecommunications and Informatics (p. 46). World Scientific and Engineering Academy and Society (WSEAS).

[2] Bauerdick, L., Benjamin, D., Bloom, K., Bockelman, B., Bradley, D., Dasu, S., ... & Lesny, D. (2012, December). Using XRootD to federate regional storage. In Journal of Physics: Conference Series (Vol. 396, No. 4, p. 042009). IOP Publishing.

[3] Boeheim, C., Hanushevsky, A., Leith, D., Melen, R., Mount, R., Pulliam, T., & Weeks, B. (2006). Scalla: Scalable cluster architecture for low latency access using XRootD and olbd servers. Technical report, Stanford Linear Accelerator Center.

[4] Fajardo, E., Tadel, A., Tadel, M., Steer, B., Martin, T., & W√ºrthwein, F. (2018, September). A federated XRootD cache. In Journal of Physics: Conference Series (Vol. 1085, No. 3, p. 032025). IOP Publishing.

[5] Gardner, R., Campana, S., Duckeck, G., Elmsheuser, J., Hanushevsky, A., H√∂nig, F. G., ... & Yang, W. (2014, June). Data federation strategies for ATLAS using XRootD. In Journal of Physics: Conference Series (Vol. 513, No. 4, p. 042049).

[6] Simon, M. (2019, March 08). XRootD Client Configuration & API Reference. Retrieved November 03, 2020, from https://XRootD.slac.stanford.edu/doc/xrdcl-docs/www/xrdcldocs.html

[7] The Worldwide LHC Computing Grid (WLCG), http://wlcg.web.cern.ch/

[8] De Witt, S., & Lahiff, A. (2014). Quantifying XRootD scalability and overheads. In Journal of Physics: Conference Series (Vol. 513, No. 3, p. 032025). IOP Publishing.

[9] Pyxrootd: Python bindings for XRootD. Retrieved November 03, 2020, from https://XRootD.slac.stanford.edu/doc/python/XRootD-python-0.1.0/

[10] XRootD: The central GitHub repository, [Source Code available on November 03, 2020]: https://github.com/XRootD/XRootD.

[11] XRootD: The official documentation [available on November 04, 2020]: https://XRootD.slac.stanford.edu/doc/doxygen/current/html/annotated.html.

[12] Andrew Hanushevsky (2018, February 13) Scalable Service Interface: The official documentation [available on November 04, 2020]: https://XRootD.slac.stanford.edu/doc/dev49/ssi_reference-V2.htm#_Toc50632342