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

Robert Poenaru
DFCTI
NIPNE-HH
Magurele, Romania
robert.poenaru@protonmail.ch

Michał Simon
Information Technology Department
CERN
Genève, Switzerland
michal.simon@cern.ch

***Abstract*—A brief description of the Xrootd architecture and its purpose within the WLCG group, together with an overview of the server- and client- sides of the Xrootd framework are discussed within the present work. The client side of Xrootd has a relatively new implementation called Declarative API. Its main purpose is to provide the user an asynchronous interface that is more in line with the modern C++ paradigm. A focus on the development workflow for this API is given. The client API is put into use in terms of the command line interface that is supports, by testing the file copying feature.**

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 then 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 plus 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. Even though LHC has proven to be able moving data at the necessary throughput [2], only by adopting a so-called federated regional storage using Xrootd will be able to avoid any potential latency issues for physicists when accessing the data and the eventual complexity of the tools involved. A discussion on the storage federation will be given in the following sections.

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 could be possible through the hierarchical deployment of redirectors that allow site discovery that have available data in real-time. Although, Xrootd supported multi-storage deployments for a long time, the addition of a feature which allowed its proper functionality within a global, multi-site environment was in fact the core idea of AAA.

In order 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 Xrootd framework, with some additional features. Experiments within LHC (e.g. ATLAS, CMS, LHCb, ALICE) but also smaller experiments (e.g. AMS) have EOS as a native solution for data storage/access for the users. 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 WLCG community.

II. Xrootd Framework

The main objective of any scientific project that is based on experiments which ran at CERN (but also to the other places within WLCG) is the access to the compute resources which are used for submitting jobs that aim to solve a particular task. An old model of such a workflow is described in diagram below (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 induces a lot of extra-time (overhead) that can lead to a relatively slow progress into tackling the actual tasks which have to be performed by the physicist with the required data. 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 which are cooperating with each other (but also independent) and also are accessible via a common namespace. By using multiple dedicated Xrootd servers at each site and a

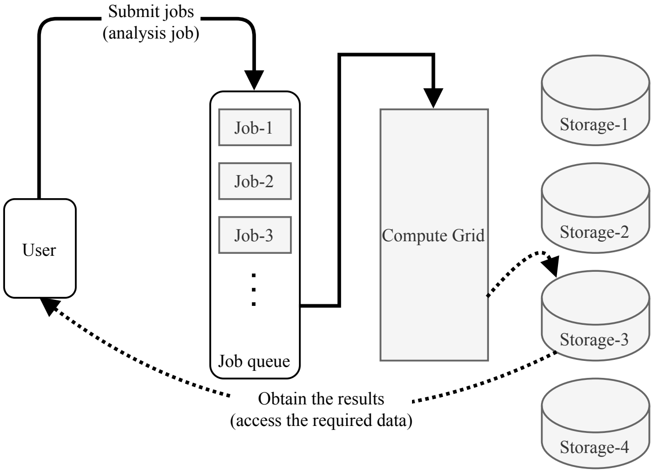

Fig. 1. Process of accessing data in the "jobs go to data" mode. User submits an analysis job that is sent through the compute grid to the sites that host the data.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

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 a direct contact with the central endpoint and is redirected to a site which can provide the necessary data.

The change between the local analysis and Grid analysis is removed within a federated storage system and on top of that, data can be accessed independent on the location, which also reduces the latency between data request and data access.

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. In fact, a tremendous work (progress) has been done in the recent years, especially for extending the scalability features (e.g., [1], [8]), caching [2] and many more. In fact, 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 into 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]).

B. Client-side Xrootd

The client part (XrdClient) of the Xrootd framework is built as a ROOT and POSIX compliant system [1], and its core functionality provide implementations like:

- A communication protocol: this allows for requesting access to an Xrootd server through an authentication process and also giving access to the desired data resources which.
- Handler for any communication errors. This is done through a development of high-level communication policies at the client side.
- Connections to a server need to support multiple and independent data streams.

Xrootd client is made up of three layers, each with a different characteristic [3].

1) The interface layer.
2) High-level communication layer.
3) Low-level communication layer.

The client is taking care of data caching, pipelining, parallelizing and aggregating requests which will provide benefits in terms of latency and throughput. This layered implementation of the XrdClient object assures data accessing methods (e.g. access files, create files, manipulate the data) but also the optimization of the data accessing process - request aggregation [3]. The POSIX interface of the XrdClient is implemented through a shared library which takes any POSIX specific call and routes it to the XrdClient.

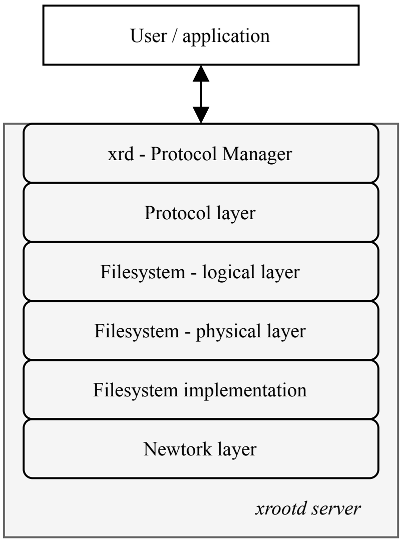

Fig. 2. Xrootd server architecture.

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 synchronous call has been implemented into the corresponding asynchronous 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 slight increase in code complexity).

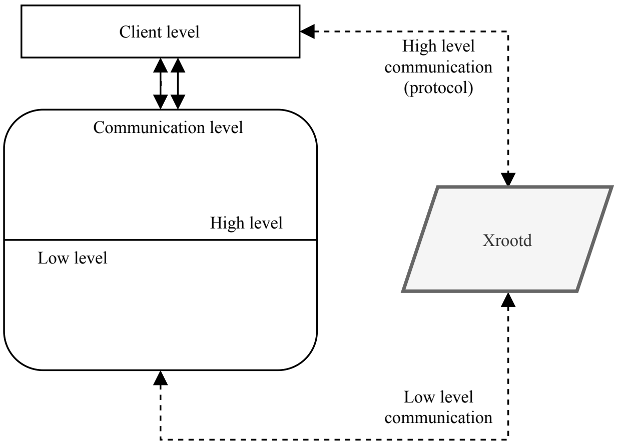

Fig. 3. Xrootd client architecture.

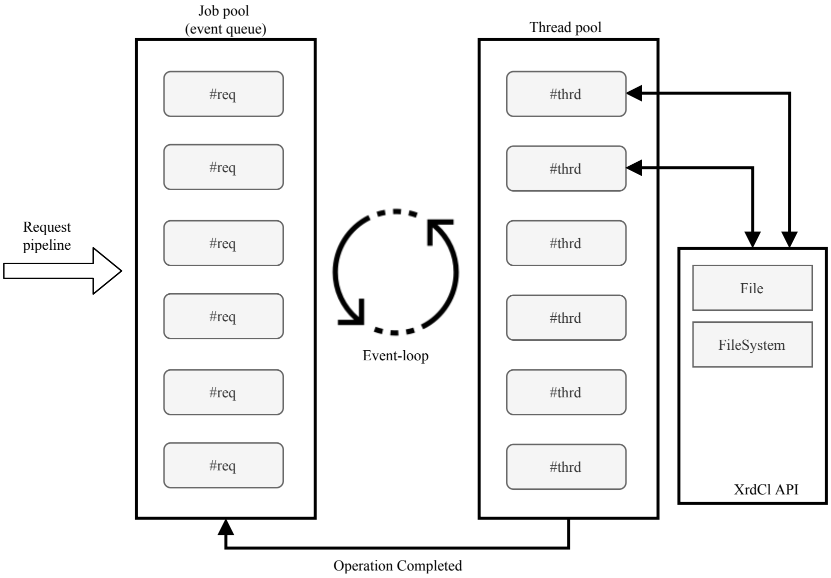

Fig. 4. The XrdCl event-loop.

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 on a single-threaded manner – sequentially. Although, there is a possibility to increase performance by upping the number of event-loops. 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 a default utility.

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 utilization until 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 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 reestablish the connection with new credentials. In fact, the server could make an authentication request to the client (when the connection between client and server is being established).

Before you begin to format your paper, first write and save the content as a separate text file. Complete all content and organizational editing before formatting. Please note sections A-D below for more information on proofreading, spelling and grammar.

A. The new Xrootd client

Firstly, released with the third major version of Xrootd, it provides the new client library libXrdCl.so and new command line utilities. The old XrdClient was completely deprecated with the launch of the fourth major release of Xrootd.

The new client supports virtual streams, multiple requests to the same server will be handled by the server in the order it so choses. In terms of handling responses, the client does not have a specific order in which it handles them: handling takes place as soon as the responses come, calling the user call-back function. The XrdCl has two implementations that deal with I/O and file specific operations (e.g. methods for opening, creating files, writing data streams to a file, directory creation and so on), namely File and FileSystem. With the new XrdCl, these two objects can be accessed from multiple execution threads safely. This can be done because the XrdCl implementation is using a worker thread pool which can handle multiple function call-backs. Even when any I/O operations are in progress, the XrdCl can support thread forking without any issues or impediments within the workflow (File and FileSystem remain valid once the forking has been done in both parent and child threads, operations inside the parent threads will continue after the fork while in the child case a recovery procedure will be executed, just like in the case of a broken connection).

B. The XrdCl event-loop

As it was already mentioned, the client implementation supports the handling of multiple executions in the background: asynchronous runtime. In general, an event loop within a programming paradigm has the following features:

- It is an endless loop which waits for tasks, executes them and then it sleeps until more tasks are incoming.
- The event loop executes tasks only when there is no other ongoing task in the pipeline (only when the call stack is empty).
- The event loop will handle function callbacks and promises (asynchronous behavior of the C++ implementation).
- The event loop executes tasks starting from the oldest to first: sequentially.

In Fig. 4 the event queue is schematically represented: when the thread pool completes a task/job, a callback function is called, which does error handling (if there are any) or some other operations (e.g. file, I/O specific). The callback function is sent to the event queue, and when the call stack is empty, the event goes through the event queue and sends the callback to the call stack.

The entire XrdCl API stack is represented as a block diagram in Fig. 5, where each of the components are organized in three main categories: Xrootd-Core, Xrootd and External.

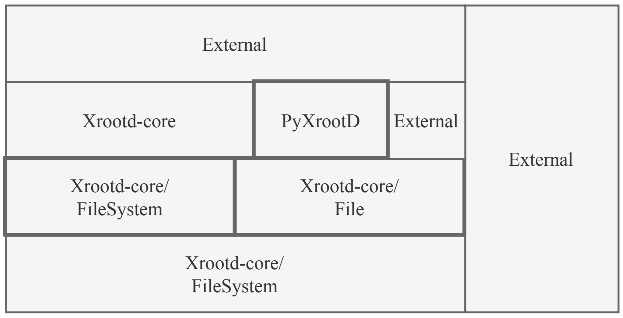

Fig. 5. Structure of the XrdCl API stack.

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 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).

C. *Asyncrhonous 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, that contains both single file as well as file system implementations. 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 within File and FileSystem objects take one extra argument: the so-called handler objects (see listings 1 and 2 for the core difference).

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

Listing 1: Synchronous version of the Open method.

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.

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

Listing 2: Asynchronous version of the Open method.

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.

```
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.

A usual execution flow which 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. In Fig. 6 a flow with only three operations that are used to write some data into a file, each with its corresponding operation handler.

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

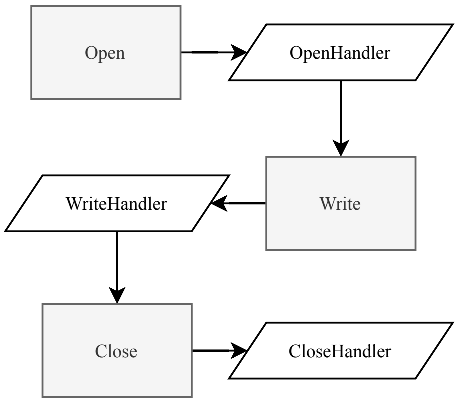

Fig. 6. A simple three file specific operations workflow for writing data to a file.

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 for facilitating the usage of XrdCl API by the users. It has been built on top of 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 for describing the main parts and characteristics of the implementation.

Its key features that actually 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.
- Syntax is declarative-centric, meaning that users should focus on the actual choice of operation rather than paying much attention (effort) on the execution flow.
- Proper signaling for user of any incorrect declarations and configurations during compilation phase.
- Error handling for the workflow is done in a consistent manner, 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: 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 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; 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 between 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 C++ programming language.

The syntax also supports a parallel execution of multiple flows of operations (XrdCl::Parallel implementation is part of the Operation Utilities within the Declarative API toolset). The Parallel utility aggregates several operations (those might be compound operations) for parallel execution. It also accepts a variable number of operations. 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 &&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.

```
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: 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 the Listing 8. The first line is for 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.

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 an error status). Using the pipelining semantic makes the control flow clearer and more robust

```
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.

V. XRDCP – TESTING THE COMMAND LINE INTERFACE

The command line interface within the Xrootd Client provides an interface between the user and the actual XrdCl C++ API but within the console. It supports multiple functions to the user, each with its flags/options.

The file copying procedure is also provided within the command line interface, namely as the xrdcp command. According to the documentation [6] - the xrdcp utility copies one or more files from one location to another. The data source and destination may be a local or remote file or directory. Additionally, the data source may also reside on multiple servers.

In the present work, the copy functionality of the Xrootd client is put to the test, by measuring the time it takes to copy files of different size from one location to another. The file will remain on the same disk space. Three text files with different size are selected for the testing phase, and the main interest is to see how is does the length of the copy process change with the increase in file size.

The environment in which these simulated results take place is a personal computer laptop running MacOS 10.15 (Catalina) equipped with a solid-state drive. For each file the copy process duration was averaged across ten runs. The bandwidth for each file is also given by the xrdcp output.

~File size (MB)	Average copy time (s)	N.o. tests
260	0.210	10
560	0.544	10
1770	1.439	10

Table 1: The test results for each file size. Average time it takes to copy a file after 10 consecutive iterations per file.

Results are given in Table 1.

Having the duration necessary to copy a file (based on its size) and keeping the same number of evaluations, one can also compute the bandwidth, and compare it across the three file sizes. Results can be seen in Figure 7.

VI. 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 following 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. Finally, the command line interface was put to the test, with the file-copy process called xrdcp.

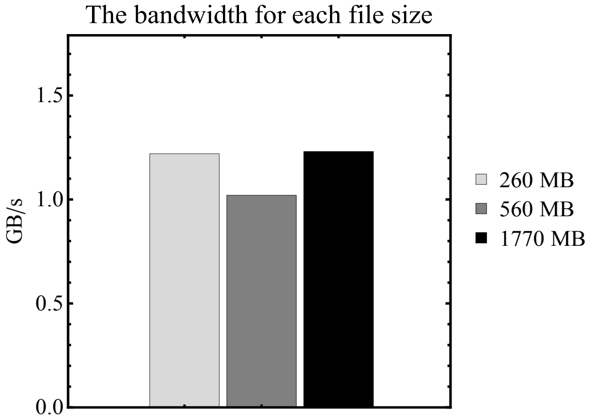

Figure 7: Results of the bandwidth calculation.

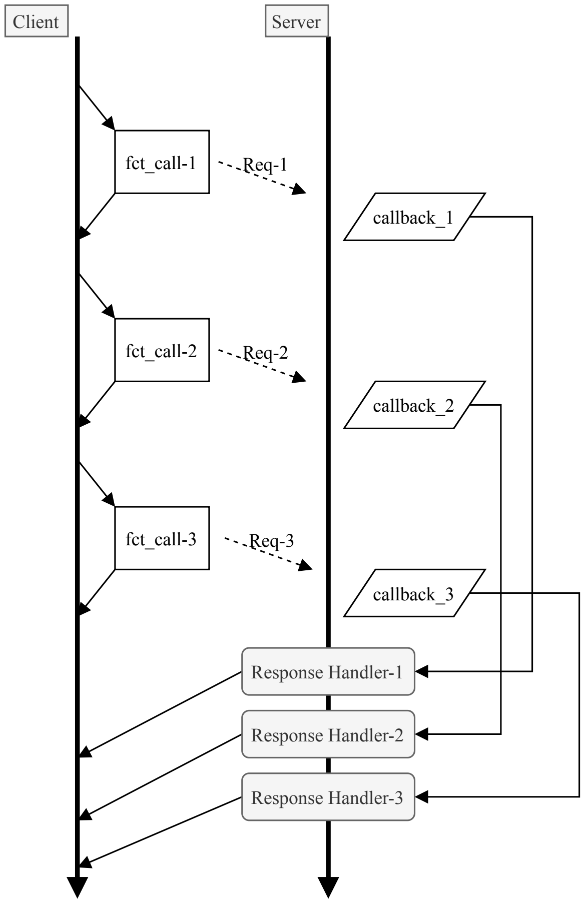

Figure 8: Asynchronous workflow in terms of response handlers.

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 department, and Ionut Vasile who provided the computational resources that were required).

APPENDIX

The asynchronous workflow for a chain of operations was discussed in Section III. The diagram in Figure 8 aims at giving a schematic representation of the pipeline, including the concept of response handler.

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