EPJ Web of Conferences **251**, 02063 (2021) https://doi.org/10.1051/epjconf/202125102063
CHEP 2021

Exploring the virtues of XRootD5: Declarative API

Michal Simon[1,*] and *Andrew* Hanushevsky[2,**]

[1]CERN
[2]SLAC

Abstract. Across the years, being the backbone of numerous data management solutions used within the WLCG collaboration, the XRootD framework and protocol became one of the most important building blocks for storage solutions in the High Energy Physics (HEP) community. The latest big milestone for the project, release 5, introduced multitude of architectural improvements and functional enhancements, including the new client side declarative API, which is the main focus of this study. In this contribution, we give an overview of the new client API and we discuss its motivation and its positive impact on overall software quality (coupling, cohesion), readability and composability.

1 Introduction

The XRootD [1] project aims at providing low latency and scalable data access for large scientific data sets and is based on a scalable, plug-in centric architecture and a communication protocol. It has been designed with particular emphasis for geographically distributed, file-based repositories. The software suite allows the deployment of federated data clusters and provides important features like access control and WAN data distribution. For almost 10 years now, the XRootD framework has been very successful at facilitating data management of LHC experiments and grew into one of the most important storage technologies in the High Energy Physics (HEP) community. It comes with no surprise that XRootD development is largely driven by the use cases coming from the WLCG project, as it is the backbone of numerous software defined storage solutions (like EOS [2] and DPM [3]) used to accommodate the vast amount of data registered by the LHC experiments at CERN, most notably Atlas [4], CMS [5], LHCb [6] and Alice [7]. One of the key components of the XRootD framework is the C++ client, which is fundamental not only to the command line utilities like *xrdcp* and *xrdfs*, but also to XCache (a XRootD file-based caching proxy) [8] [9], XrootdFS (a FUSE based mountable file system) [10] and EOS (the storage service of choice at CERN). In addition, the XRootD client is employed to provide remote data access in many physics analysis frameworks like ROOT [11] and in data movers like FTS [12]. With latest major release (5.0.0) the XRootD framework brought multitude of architectural improvements and functional enhancements, including the new client side declarative API, which is the main focus of this study. This paper first outlines the motivation for introducing the declarative API into the XRootD client library, and discusses the property of composability. As case study we consider a ZIP archive metadata parser. Subsequently, the syntax and

*e-mail: michal.simon@cern.ch
**e-mail: abh@slac.stanford.edu

fundamental concepts required to understand and use the new API are explained. Finally, we conclude the paper with results of applying some common software metrics (cohesion, coupling, cyclomatic complexity) to the software developed using the new declarative API and we provide a short summary.

2 Motivation

2.1 The Object Oriented APIs

Before XRootD5 has been released, there were just two types of APIs available in the client, an Object Oriented (OO) synchronous API (see List. 1) and an OO asynchronous API (see List. 2). The OO synchronous API is easy to use, the code readability is very good, but there is virtually no composability [13]. In other words, it is not possible to chain the remote access operations together. Let us consider now an example where the user would like to read in parallel from multiple files. A major limitation of the synchronous API is that concurrent access could be achieved only by adding more threads to take advantage of core parallelization, which will increase context management overhead.

Listing 1. Synchronous OO API

```
1  File f;
2  f.Open( url, OpenFlags::Read );
3  f.Read( offset, length, buffer );
4  f.Close();
```

The OO asynchronous API allows to chain the remote access operations however code readability is poor (the flow control is not clearly expressed). Moreover, it requires significant amount of boilerplate code, because each of the asynchronous operations requires a special custom callback object.

Listing 2. Asynchronous OO API

```
1  File f;
2  // read & close operations are called from the callback
3  f.Open( url, OpenFlags::Read, custom_callback );
```

2.2 Use cases

The development of the XRootD client declarative API has been driven by two major use cases: the client erasure coding (EC) [14] plugin for the Alice O2 project [15] and a ZIP archive metadata parser. The former requires extensive parallel remote access to multiple devices for every implemented operation. The later needs to issue consecutive reads in order to parse the ZIP archive metadata. In order to illustrate the difficulties of using the OO asynchronous API, we will consider the logical open operation of an erasure coded file. In our implementation, the first component of a logical EC open are parallel open requests to n data and p parity stripes (in total $n+p$ requests), of which n have to be successful. The second component, is an open, followed by a read, and then followed by a close of a metadata file. Any error happening during metadata retrieval can be recovered at a different location holding redundant copy of the metadata file. To implement this logic using the asynchronous OO API, five custom callbacks have to be provided in order to ensure chaining of the asynchronous operations, which is a significant amount of boilerplate code (see List. 3). Moreover, the program is difficult to understand as significant fraction of the logic is hidden in the callback objects. For instance, other than for the comment, it is not clear from inspecting the *EcOpen* function that the metadata are being actually read.

Listing 3. Logical open operation of an erasure coded file

```
/*CUSTOM CALLBACKS DEFINITIONS*/
struct ecopen_cb_t{ /* implementation */ };
struct paropen_cb_t{ /* implementation: notify ecopen callback */ };
struct openmd_cb_t{ /* implementation: issue read */ };
struct readmd_cb_t{ /* implementation: issue close */ };
struct closemd_ccb_t{ /* implementation: notify ecopen callback */ };

/*OPEN FOR READING*/
void EcOpen( /*arguments*/ )
{
  /*Allocate callbacks*/
  auto ecopen_cb  = new ecopen_cb_t( /*arguments*/ );
  auto paropen_cb = new paropen_cb_t( ecopen_cb, /*arguments*/ );
  auto openmd_cb  = new openmd_cb_t( ecopen_cb, /*arguments*/ );
  auto readmd_cb  = new readmd_cb_t( ecopen_cb, /*arguments*/ );
  auto closemd_cb = new closemd_cb_t( ecopen_cb, /*arguments*/ );
  /*Open stripes*/
  for( auto &f : stripes )
    f.Open( /*arguments*/, paropen_cb );
  /*Read the metadata file*/
  metadata.Open( /*arguments*/, openmd_cb );
}
```

After analysing the extra code that had to be written in order to chain the OO operations, we have extracted the common patterns (e.g. callback classes), applied significant amount of template meta-programming and flavored it with a pinch of operator overloading. As a result we got a new declarative API that is in line with the modern C++ programming practices (ranges v3 inspired, support for lambdas and *std::futures*), offers greater code readability and genuine composability.

3 The declarative API

Let's consider what would be a good model for asynchronous remote I/O programming. As in case of any other software engineering problem, we would like to be able to decompose larger tasks into smaller ones. For instance, we would like to be able to decompose a file update operation into an open, a write and a close. Afterwards, we can finally program each of the primitive I/O operations. However, there is one more critical step, we have to be able to compose those smaller operations back together into the original, bigger problem. The most important thing is that those operations have to be trivially composable, if the programmer has to know the internals of an operation implementation in order to be able to compose it with another one then all is lost [13]. In addition, it is critical that the operations are lazy evaluated, meaning one can first declare the operations, compose them into a pipeline and only then execute the whole pipeline. Finally, the remote I/O operations have to be associative in order to be really composable (meaning that object *updt1* and *updt2* in listing 4 are equivalent).

Listing 4. Operations associativenes (pseudo code, operator| is used as composition)

```
auto opn_wrt = Open|Write;
auto wrt_cls = Write|Close;
auto updt1   = opn_wrt|Close;
auto updt2   = Open|wrt_cls;
```

3.1 Rules of pipelining

Now, let us dive into the new XRootD declarative API and let us start the tour with the most important topic, which is composability. In XRootD client all the I/O operations can be composed with the | operator. We will call all the basic operations (remote counterparts of POSIX open, read, write, etc.) as *primitive* and all the results of composing two or more operations as *composed*. Another useful concept that we will need is the Pipeline utility class, it is a general-purpose polimorphic I/O operation pipeline wrapper. In other words, instances of Pipeline can store any I/O operation (*primitive* or *composed*). The last thing we need to know to create our first pipeline (see List. 5) is that each I/O operation needs a context (or a handle if you will), in most cases this will be the *File* or *FileSystem* object.

Listing 5. Simple pipeline example: open, read and close

```
1  File f;
2  Pipeline p = Open(f, url, OpenFlags::Read)
3             | Read(f, off, len, buf)
4             | Close(f);
```

Pipelines obey certain rules. First of all, as we mentioned before, the operations within a pipeline are associative. Secondly, defining a pipeline does not trigger it (in a sense, it is lazy evaluated). Thirdly, once executed, operations in the pipeline are performed strictly from left to right (in our example it is first the *Open*, then the *Read*, and then the *Close*). Finally, if during the execution an operation fails the pipeline stops, and subsequent operations are ignored.

3.2 Executing a pipeline

A pipeline can be executed either using the *Async* or *WaitFor* functions. The *Pipeline* object needs to transfer the ownership of the underlying operations pipeline to the executing routine. The *Async* function (as the name suggests) triggers asynchronous execution of the pipeline and returns a *std::future* to the final status that is the outcome of carrying out the I/O operations in the given pipeline (see List. 6).

Listing 6. Execute pipeline with Async

```
1  auto ftr_status = Async(std::move(p));
```

The *WaitFor* function triggers synchronous (blocking) execution of the pipeline and returns the final status that is the outcome of carrying out the I/O operations in the given pipeline (see List. 7).

Listing 7. Execute pipeline with WaitFor

```
1  auto status = WaitFor(std::move(p));
```

3.3 Handlers

Any operation can (but does not have to) be assigned with a handler using the » operator. In a simple case the result of an operation can be directed to an instance of *std::future* (see List. 8).

Listing 8. Direct operation response to a std::future

```
1  File f;
2  std::future<ChunkInfo> rsp;
3  Async( Open(f, url, OpenFlags::Read) | Read(f, off, len, buf) >> rsp );
```

```
4
5     // later on use the future
6     try
7     {
8       // use the response
9       ChunkInfo chunk = rsp.get();
10    }
11    catch(PipelineException &ex)
12    {
13      // if the read request failed we will get an exception
14      XRootDStatus &status = ex.GetError();
15    }
```

In more complex cases, an I/O operation can be handled by a *function*, *functor* or *lambda* (see List. 9).

Listing 9. Handle operation with lambda

```
1   File f;
2   WaitFor(Open(f, url, OpenFlags::Read) >>
3               [](XRootDStatus &st)
4               {
5                 // handle response
6               });
```

Finally, an I/O operation can be handled also with a *packaged_task* and for backwards compatibility also with an instance of *ResponseHandler* (an XRootD4 style handler).

3.4 Forwarding values between handlers and operations

In order to facilitate forwarding values between handlers and operations the Fwd class has been provided. Fwd is an argument wrapper accepted by any I/O operation. Initially, a Fwd instance contains no value and can be assigned with one for example in an operation handler. To illustrate the usefulness of forwarding values, let us consider following example. Let us assume that there is a file (not to big) of unknown size and that there is a need to read the file with a single read request. In order to implement this scenario, we will forward two values from the handler of an *Open* operation that has the stat information of the given file as arguments to the *Read* operation (see List. 10).

Listing 10. Forwarding values

```
1   File f;
2   Fwd<uint32_t> len; // we will assign values later
3   Fwd<char*>    buf; // on in the open handler
4   Pipeline p = Open(f, url, OpenFlags::Read) >>
5                  [len, buf](XRootDStatus &st, StatInfo &inf)
6                  {
7                    if(!st.IsOK()) return;
8                    // forward the length and buffer
9                    len = inf.GetSize();
10                   buf = new char[inf.GetSize()];
11                 }
12              | Read(f, 0, len, buf) // use forwarded arguments
13              | Close(f);
```

3.5 Control directives

There are four control directives that allow to alter pipeline execution: *Pipeline::Stop*, *Pipeline::Repeat*, *Pipeline::Ignore* and *Pipeline::Replace*. All of those directives must be

called from within an operation handler body. The *Pipeline::Stop* forces the pipeline to be stopped, the user may optionally provide the final status as an argument (defaults to success). The *Pipeline::Repeat* forces the current operation to be repeated (e.g. repeat read operation until the end-of-file, see List. 11). The *Pipeline::Ignore* makes the pipeline ignore an operation failure and resumes execution at the operation next in turn (e.g. ignore a read error and proceed to *Close*, see List. 11).

Listing 11. Control directives: print file content to stdout

```
File f;
Fwd<uint64_t> off = 0;
uint32_t      len = 1024;
char*         buf = new char[len+1];

Pipeline p = Open(f,url,OpenFlags::Read)
           | Read(f,off,len,buf) >>
               [off](XRootDStatus &st, ChunkInfo &ch)
               {
                 if(!st.IsOK())
                   Pipeline::Ignore(); // proceed to close
                 if(ch.length == 0) return; // EOF
                 auto len = ch.length;
                 auto buf = ch.buffer;
                 buf[len] = 0;
                 std::cout << buf;
                 // adjust the offset
                 off = *off+1024;
                 // repeat until EOF
                 Pipeline::Repeat();
               }
           | Close(f);
```

The *Pipeline:Replace* is overloaded and can be used either to replace the current operation with a different one or to replace the whole pipeline. This facility allows to define composed I/O operations and whole pipelines in a recursive way (e.g. recursively try opening redundant file replicas, see List. 12).

Listing 12. Recursively try replicas.

```

auto TryOpen(File &f,const std::vector<std::string> &urls,size_t i=0)
{
  return Open(f,urls[i],OpenFlags::Read) >>
         [&f,&urls,i](XRootDStatus &st)
         {
           if(st.IsOK) return;           // we found a valid replica
           if(i+1>=urls.size()) return; // there are no more replicas to try
           // recover error at next replica
           Pipeline::Replace(TryOpen(f,urls,i+1));
         };
}
```

3.6 Special operations: Parallel and Final

The XRootD client declarative API provides two special operations *Parallel*, which allows parallel execution of asynchronous component I/O operations and *Final* that provides a standard place for resource deallocation. The *Parallel* operation accepts multiple I/O operations as arguments or a container of I/O operations (see List. 13). Moreover, one of the three available policies might be chosen for a *Parallel* operation: *Any* - it is enough if just one of

the component operations is successful for the *Parallel* operation to be successful, *Some* - it is enough if just *N* of the component operations are successful for the *Parallel* operation to be successful, and *AtLeast* - same as *Some* and in addition it is guaranteed that the completion handler of the *Parallel* operation will be called only when all component operations are resolved.

Listing 13. Parallel execution.

```
1
2   // Issue two read requests at offsets:
3   // `off1` and `off2` of size `len`
4   auto ParReadArgs(File     &file, uint64_t  off1, uint64_t  off2,
5                    uint32_t  len,  void     *buf1, void     *buf2)
6   {
7     return Parallel(Read(file,off1,len,buf1),  // first read request
8                     Read(file,off2,len,buf2)); // second read request
9   }
10
11  // Issue a read request for every buffer that
12  // has been provided by the user
13  auto ParReadCont(File                    &file,
14                   std::vector<uint64_t> &offs,
15                   std::vector<buffer_t> &bufs)
16  {
17    std::vector<Pipeline> reads; reads.reserve(bufs.size());
18    for(size_t i=0;i<bufs.size();++i)
19      // add another read request
20      reads.emplace_back(Read(file,offs[i],bufs[i].size(),bufs[i].data()));
21    return Parallel(reads);
22  }
```

The *Final* operation is always guaranteed to be executed even if the pipeline has been stopped prematurely due to an error. Moreover, *Final* MUST always be the last operation in the pipeline. The *Final* utility has been introduced in order to facilitate resource management (see List. 14).

Listing 14. Final utility.

```
1   auto f = std::make_shared<File>();
2   Pipeline p = Open(*f,url,OpenFlags::Read)
3              | Read(*f,off,len,buf)
4              | Close(*f)
5              | Final([f](XRootDStatus&)
6                      {
7                        // make sure `f` is deallocated only after
8                        // all the pipeline's operations have been
9                        // executed, no matter if there was an error
10                       // or not
11                     });
12  Async(std::move(p));
```

4 Case study and results

As a case study we will consider the ZIP archive class of the XRootD client. In particular, we will focus on the ZIP archive open routine that aims at opening the file and then at parsing the ZIP metadata [16]. The routine estimates the offset of the metadata based on the sizes of individual metadata records. However due to a variable-size user added comment the chosen offset might be invalid and as a result several reads might be needed. The ZIP archive open

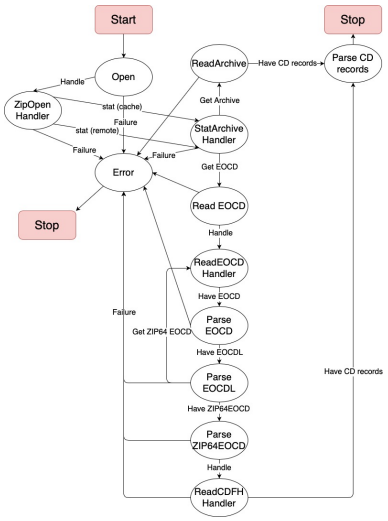

Figure 1. OO API impl: control flow graph

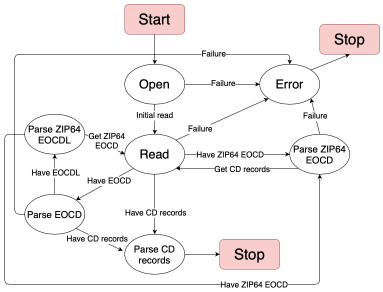

Figure 2. Declarative API impl: control flow graph

routine has two implementations: first that uses the old XRootD4 OO asynchronous API (*ZipArchiveReader*) and second based on the new XRootD5 declarative API (*ZipArchive*) [17]. In order to compare the two implementations we applied some standard software metrics to both software routines.

4.1 Cyclomatic complexity

Cyclomatic complexity is a metric that allows to determine the number of linearly independent paths within a section of source code. Two paths are considered as linearly independent if and only if an edge exists that belongs to only one of those paths. [18] Cyclometric complexity ***M*** is resolved based on a control flow graph of the given section of source code. The cyclometric complexity of the ZIP archive open routine implemented using the XRootD4 OO API and XRootD5 declarative API have been determined respectively to ***M=14*** based on the control flow graph shown in Figure 1 and to ***M=8*** based on the control flow graph shown in Figure 2. The significantly lower cyclometric complexity of the implementation that employed the new declarative API implies easier code maintainability and is due to greatly simplified error handling. Finally, lower cyclometric complexity means there are fewer execution paths that need to be accounted for in the test suite.

4.2 Cohesion

Cohesion is an ordinal software metric that reflects the degree to which a software module or a class is unified around a central concept it serves. It is a measure of how strongly the encapsulated data are related and of how much the functionalities embedded in a single software module have in common. High cohesion is a desirable property because it is associated with several important characteristics of software like robustness, reusability and understandability. [19][20]

The fundamental goal of the ZIP archive class is to extract the information about data layout in the archive from the metadata and to serve the data itself to the end user. In the

implementation employing the XRootD4 OO API, the ZIP metadata parsing functionality, the custom asynchronous callbacks providing operation chaining and core functionality of the class are strongly interleaved. One can easily notice a recurring pattern: an asynchronous operation is issued, then its result is interpreted as ZIP metadata, and then subsequently the custom completion handler chains another asynchronous operation. As a result, the XRootD4 OO API based implementation can be classified as having ***sequential cohesion***. On the other hand, in the implementation based on XRootD5 declarative API it has been possible to extract ZIP metadata parsing from the ZIP archive class into a separate module providing this functionality. Moreover, there is no need for custom completion handlers that provide operation chaining as this functionality is by default available in the new API. As a result, the ZIP archive class is focused only at its core role and hence the implementation can be classified as having ***functional cohesion***.

It is worth noticing that higher cohesion of the implementation employing the declarative API results in better reusability (extracted ZIP parsing functionality in separate module), more concise codebase (37% shorter compared to its counterpart) and enhanced code readability and maintainability (its counterpart needed 5 custom completion handler classes).

4.3 Coupling

Coupling is a software metric indicating how closely connected two (or more) routines or modules are. Low coupling often implies good software design, and if accompanied by high cohesion, leads to high readability and maintainability. [19]

The problem of tight coupling of the ZIP archive class and the ZIP metadata parser that occurred in the implementation based on XRootD4 OO API (*control* coupling) has been fully resolved in the refactored version of the software that uses the new declarative API (*data* coupling).

5 Conclusions

The new declarative API introduced in XRootD5 facilitates functional software design and allows for a better-structured code, as confirmed by applying several software metrics. It gives a standard way of composing asynchronous I/O operations, which comes in handy especially when implementing complex remote data access schemes like erasure-coding. In addition, it provides a set of control directives that make it possible to dynamically alter a running I/O pipeline. The XRootD client declarative API is in line with modern C++ programming practices and facilitates use of valuable utilities like lambdas and *std::futures* in the context of the framework.

The new API provides a convenient and efficient way of doing asynchronous I/O in the present day without the need of falling back to slightly more heavyweight technologies like stackfull *boost::fibers* or having to wait for compiler support for C++20 (introduces coroutines) on target platforms.

References

[1] A. Hanushevsky and D. L. Wang, IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, 1168-1175 (2012)
[2] AJ Peters et al, J. Phys.: Conf. Ser. **664**, 042042 (2015)
[3] A Manzi et al, J. Phys.: Conf. Ser. **898**, 062011 (2017)
[4] The ATLAS Collaboration et al, JINST **3**, S08003 (2008)

[5] The CMS Collaboration et al, JINST **3**, S08004 (2008)
[6] The LHCb Collaboration et al, JINST **3**, S08005 (2008)
[7] The ALICE Collaboration et al, JINST **3**, S08002 (2008)
[8] E Fajardo et al, J. Phys.: Conf. Ser. **1085**, 032025 (2018)
[9] R W Gardner et al, J. Phys.: Conf. Ser. **898**, 062017 (2017)
[10] Wei Yang and Andrew Bohdan Hanushevsky, J. Phys.: Conf. Ser. **898**, 062046 (2017)
[11] Danilo Piparo, J. Phys.: Conf. Ser. **664**, 062049 (2015)
[12] A A Ayllon et al, J. Phys.: Conf. Ser. **513**, 032081 (2014)
[13] Bartosz Milewski, Category Theory for Programmers **0464825083, 9780464825081** (2018)
[14] I. S. Reed and G. Solomon, Journal of the Society for Industrial and Applied Mathematics **8**, 300–304 (1960)
[15] Ananya et al, J. Phys.: Conf. Ser. **513**, 012037 (2014)
[16] .ZIP File Format Specification, https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT (2020)
[17] A. Hanushevsky et al, XRootD, GitHub repository, https://github.com/xrootd/xrootd/tree/v5.1.1
[18] McCabe, IEEE Transactions on Software Engineering **4**, 308–320 (1976)
[19] E. Yourdon and C. L. LeRoy, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design **978-0-13-854471-3, ISBN 0-13-854471-9** (1975)
[20] J. Ingeno, Software Architect's Handbook **178862406-8** (2018)