C++ Declarative API

Implementation Overview Within the XRootD Framework

Robert Poenaru
Research Assistant @ IFIN-HH
robert.poenaru@protonmail.ch

Outline

- Aim & Motivation
- XRootD - framework overview
  - Server-side
  - Client-side
- Standard C++ API of the XRootD client: File and FileSystem APIs
- New C++ API for the XRootD client
  - Development of an Erasure Coding plug-in
- Conclusions

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

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

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

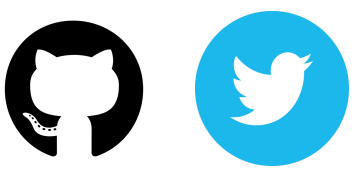
***Abstract*—A brief description of the XRootD architecture and its purpose within the Worldwide Large Hadron Collider Computing Grid (WLCG), alongside an overview of the server- and client- sides of the XRootD framework are discussed in the present work. The client-side of XRootD has a relatively new 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.**

In terms of its functionality, the XRootD framework is composed of a server-side and a client-side. Each component will be described in detail in the following sections; however, it is worth mentioning that the interaction of any end-user with the XRootD framework (in the process of accessing stored data) will be through the client interface (or shortly XrdCl). Making sure that the stored data from any of the facilities which run experiments is available to the user and assuring a constant transfer bandwidth even when the distributed storage system is accessed by a multitude of clients represent a real challenge, especially when considering that, as in any kind of data storage facility, equipment may fail (for example disks that stop working could lead to entire storage blocks to be unresponsive). Nowadays, with the larger capacity disk

About me

- **Research Assistant** @ Department of Theoretical Physics (IFIN-HH)
  - Part-time R.A. @ Department of Computational Physics and Information Technology (IFIN-HH)
- **Ph.D. student** @ Faculty of Physics (University of Bucharest)
  - Topic: Nuclear Structure (Theoretical Physics)
  - Studying the structure of the triaxially deformed nuclei
    - Developing (phenomenological) models that describe the rotational effects in deformed nuclear isotopes



@basavyr

Some disclaimers...

- **xrd**: refers to the XRootD term
- **client**: referring to the XRootD client
- **server**: referring to the XRootD server-side
- **standard C++ API**: referring to the (pre-existing) XRootD client-based API, that is developed in C++ (both asynchronous and synchronous implementations)
- **new/Declarative API**: referring to the newly developed C++ API for the client

Aim

- Provide an overview of the XRootD framework that is used within **WLCG**
  - server-side
  - client-side
- Overview of the standard C++ API for the XRootD client
- C++ Declarative API for the XRootD client
  - Structural overview (in terms of implementation)
  - Case-study: Erasure Coding plug-in mechanism

Motivation

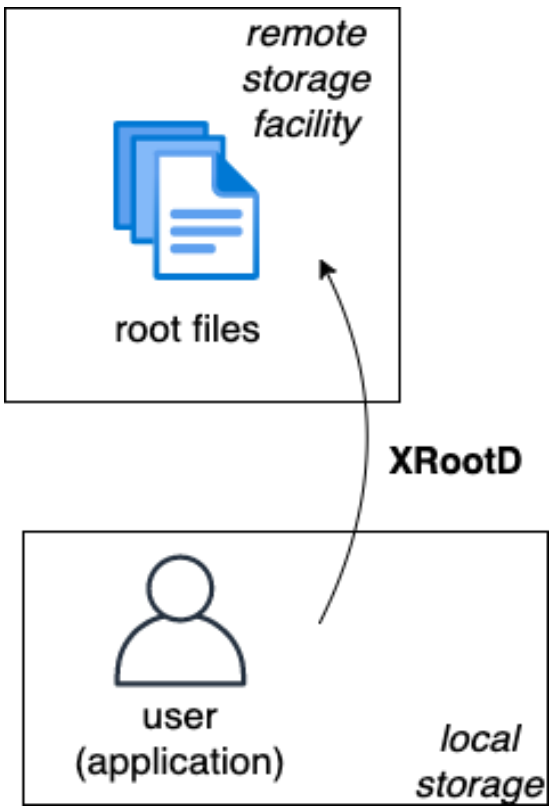
- The standard C++ API can become quite *cumbersome* in:
  - workflows that involve parallel operations
  - containerized operations that need to be executed asynchronously
- Simplification of the codebase could be beneficial for the end-user (when using the client) -> a *Declarative* approach
  - reduce code complexity
  - makes a workflow more readable
  - proves to be more in line with modern C++ coding techniques

XRootD Framework

- Service that provides *remote access* to **root** files.
  - root: a framework for data analysis, developed at CERN.
- Developed firstly at Stanford Linear Accelerator Center, now maintained together with CERN
- It is the standard remote file access within the WLCG.
- Currently, the framework is used in:
  - data access
  - data management
  - data analysis

XRootD Framework

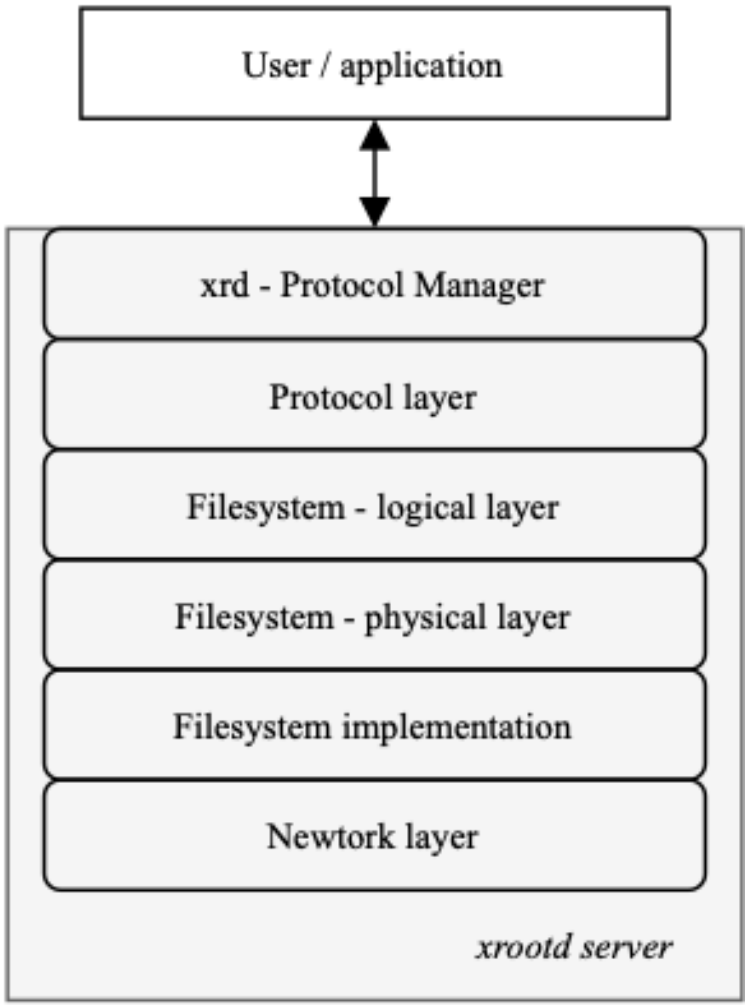
- Any user (e.g. scientist) that wants access to remote files interacts with the XRootD framework via the XRootD *client*.
- Furthermore, the XRootD *server* implementation will search for the required data (files), through a *federated storage system,* by using a so-called *redirector.*



Bauerdick, L., et al. "Using xrootd to federate regional storage." Journal of Physics: Conference Series. Vol. 396. No. 4. IOP Publishing, 2012.

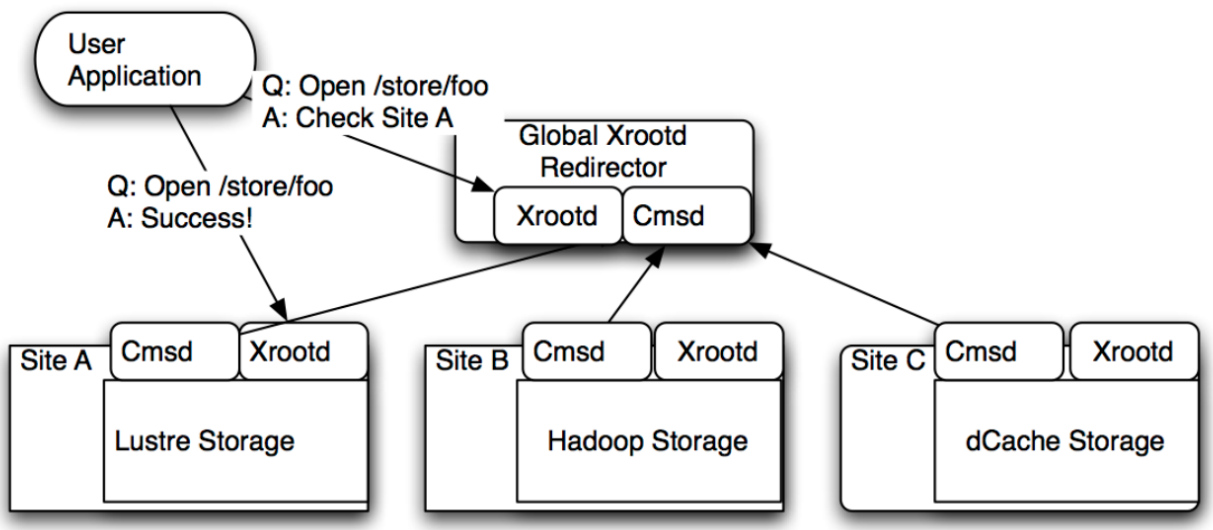
server-side xrd

- At its core, xrd-server acts like any remote data server.
- It is innovative in terms of:
  - its scalability
  - robustness
  - fault-tolerance
  - job recovery (in case of job failing during an execution)
  - cache mechanism



server-side xrd: workflow for file access

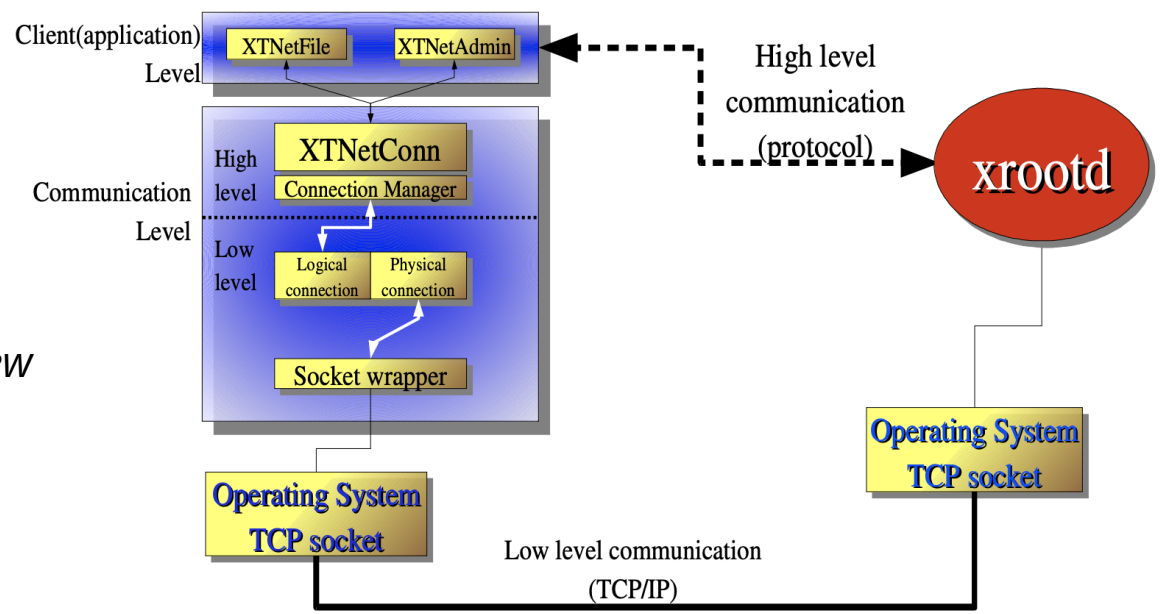
- The client contacts a centralized endpoint with a request to open a file.
  - This Xrootd server called a *redirector* is connected to a daemon: Cluster Management Service Daemon (*cmsd*).
  - The redirector's *xrootd* daemon queries for a file's location
  - If the location is not known in the local memory cache, the *cmsd* will subsequently query all sites (storage facilities) subscribed to the redirector.



Bauerdick, L., et al. "Using xrootd to federate regional storage." Journal of Physics: Conference Series. Vol. 396. No. 4. IOP Publishing, 2012.

xrd client

- The implementation that allows users to store, retrieve, and edit remote data (i.e., root files) that are located remotely on the storage grid at CERN.
- Developed in C++:
  - Newer versions of the XRootD framework support *Python Bindings.*
  - Most important APIs: File and FileSystem
  - Shortly named **XrdCl.**



Architecture overview of the xrd client

Dorigo, Alvise, et al. "XROOTD-A Highly scalable architecture for data access." WSEAS Transactions on Computers 1.4.3 (2005): 348-353.

xrd client

- The **client** is composed of three layers:
  - the **interface** layer (methods dealing with file access are defined)
  - the **high-level** communication layer (policies related to fault tolerance, read caching)
  - the **low-level** communication layer:
    - allows for connection multiplexing
    - synchronous/asynchronous communication with the xrd server
    - socket error handling
    - takes care of like read/write, socket polling to handle read/write timeouts, connection, and disconnection
    - detects connection timeouts

xrd client

- Within a workflow that involves file accessing, the client provides two main APIs:
  - XrdCl::File
  - XrdCl::FileSystem
- Both APIs are implemented in C++, with sync/async implementations
- List of operations (specific to each class of APIs)
  - File: Open(),Read(),Write(),Close(),Truncate(),VectorRead(),VectorWrite() and so on.
  - File-System: Locate(), Mv(), Truncate(), Rm(), Mkdir(), RmDir() and so on.
- Each operation has a well defined set of arguments

sync vs. async

- Two versions implemented in the standard API for File and FileSystem functions:
  - synchronous
  - asynchronous
- The main difference from the API perspective: asynchronous functions take one more argument (i.e., the *handler object).*

sync Write() function

```
XRootDStatus Write( uint64_t        offset,
                    uint32_t        size,
                    const void      *buffer,
                    uint16_t        timeout = 0 );
```

async Write() function

```
XRootDStatus Write( uint64_t        offset,
                    uint32_t        size,
                    const void      *buffer,
                    ResponseHandler *handler,
                    uint16_t        timeout = 0 );
```

async within the standard xrd client API

- For a simple workflow involving file-based operations, one can consider the following operation pipeline:
  - Opening a file
  - Read file content
  - Close the file
- For a proper execution flow, each next function needs to be called from the previous function's handler.
- At first ***Open*** operation is called -> Inside the handler, the second operation (***Read***) is called with another handler -> third operation (***Close***) is called.

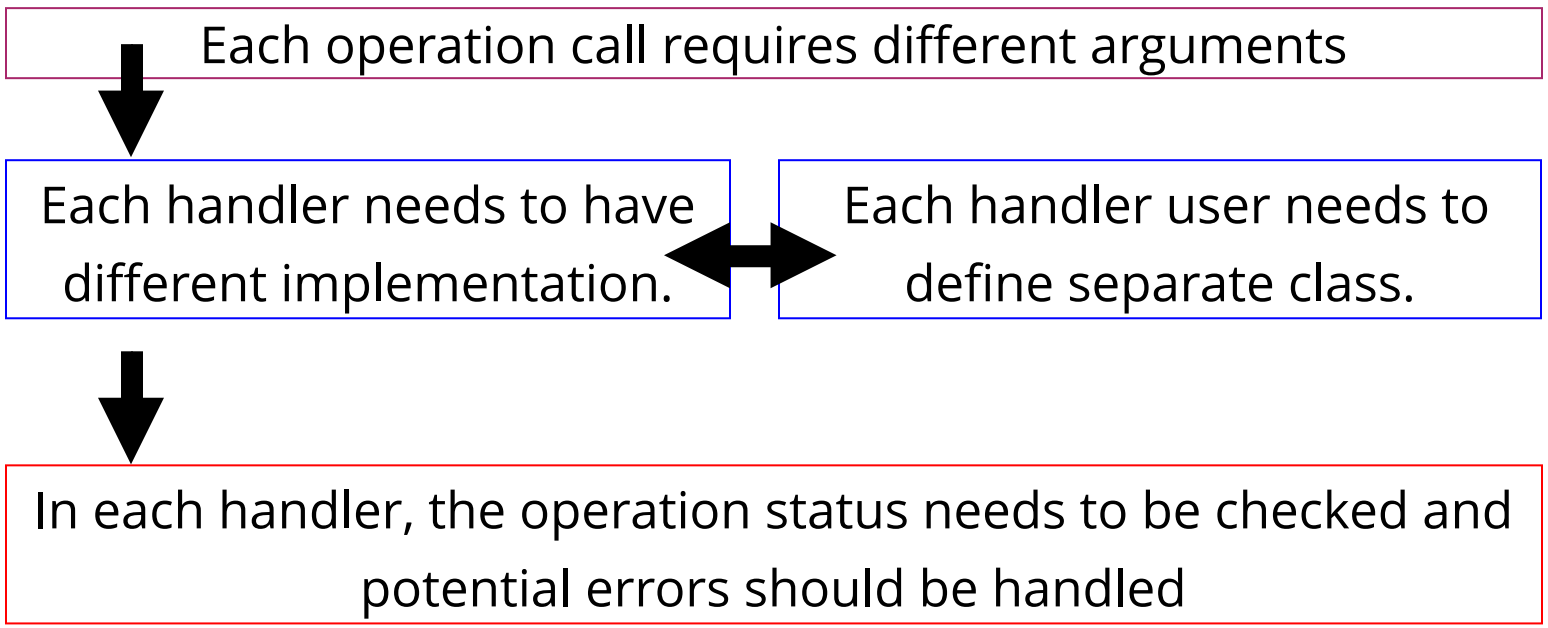
```
const string path = "/tmp/testfile.txt";
const OpenFlags::Flags flags = OpenFlags::Read;
const Access::Mode mode = Access::None;

auto openHandler = new CustomOpenHandler();

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

async within the standard xrd client API

Drawback nr1 - code complexity



⚠ **This process becomes even more difficult in case of more complex flow including parallel operations**

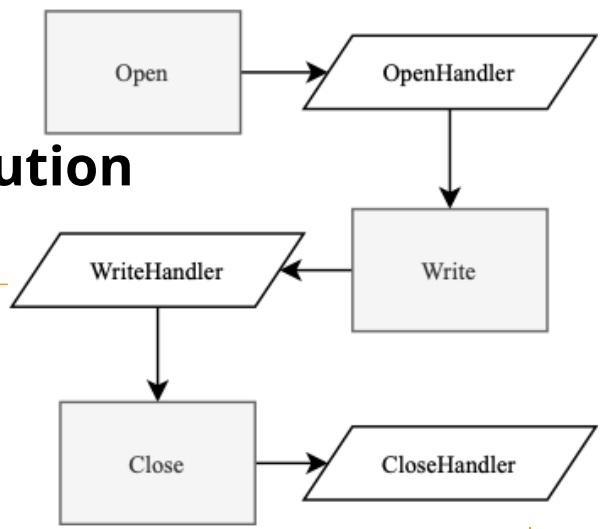
async within the standard xrd client API

Drawback nr2 - code readability

- Only the first function call can be seen in the "chain" of operations.
- Further execution is done in the handler.

- To understand the entire execution flow, the user would need to go through a whole set of handlers what can be very inconvenient and time-consuming.

- ⚠️ **Even worse in the case of parallel execution**




```
const string path = "/tmp/testfile.txt";
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
```

xrd client -> C++ Declarative API

🚀 NEW ALTERNATIVE

⚙️ *Developed as a potential tool which might mitigate the standard API drawbacks*

- Allows File and FileSystem operations.
- Its sole focus is on facilitating:
  - the asynchronous programming model
  - chaining of operations
- Designed to be more in line with modern C++ programming practices (e.g., lambdas, futures, packaged tasks).
- The syntax was created through the usage of the *operator overloading* feature utility available in C++.

Declarative API

Pipeline: a class that can wrap any kind of operation (including compound operations).

```
File f;
std::future<ChunkInfo> resp;

// open, read from and close the file
Pipeline p  = Open(file, url, OpenFlags::Read)
            | Read(file, offset, size, buffer) >> resp
            | Close(file);

auto status = WaitFor(p);
```

Async: utility for *asynchronous* execution of pipelines.

```
FileSystem fs(url);
std::future<XRootDStatus> status =
                    Async( Truncate(fs, path, size) );
```

WaitFor: utility for *synchronous* execution of pipelines.

```
FileSystem fs(url);
XRootDStatus status = WaitFor( Truncate(fs, path, size) );
```

Declarative API

```
File f;
std::future<ChunkInfo> resp;

// open, read from and close the file
Pipeline p  = Open(file,url,OpenFlags::Read)
            | Read(file,offset,size,buffer) >> resp
            | Close(file);

auto status = WaitFor(p);
```

The pipe "|" operator is overloaded: defines pipelined operations (each function call will be executed in order)

The operator ">>" is overloaded: provides a custom handler for the given operation.

⚠ The handler is optional (the pipeline will stop in case of occurring errors throughout the execution).

Declarative API

- **XrdCl::Parallel**: aggregates several operations (might be compound operations) for parallel execution.
  - E.g.: "o1" could be a compound operation like the previous one
- As an argument: accepts a variable number of operations or a container of operations.

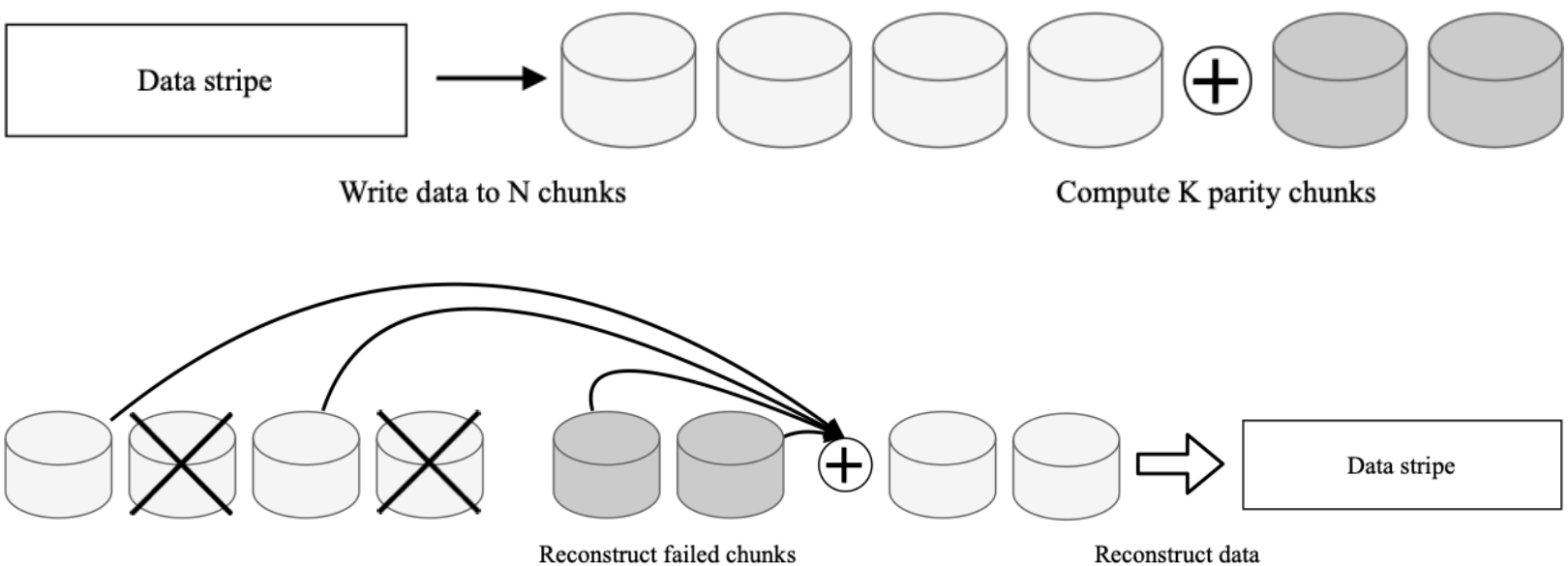
```
auto &&o1   = Open(file1,url1,OpenFlags::Read);
auto &&o2   = Open(file2,url2,OpenFlags::Read);
auto &&o2   = Open(file2,url2,OpenFlags::Read);

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

auto status = WaitFor( p );
```

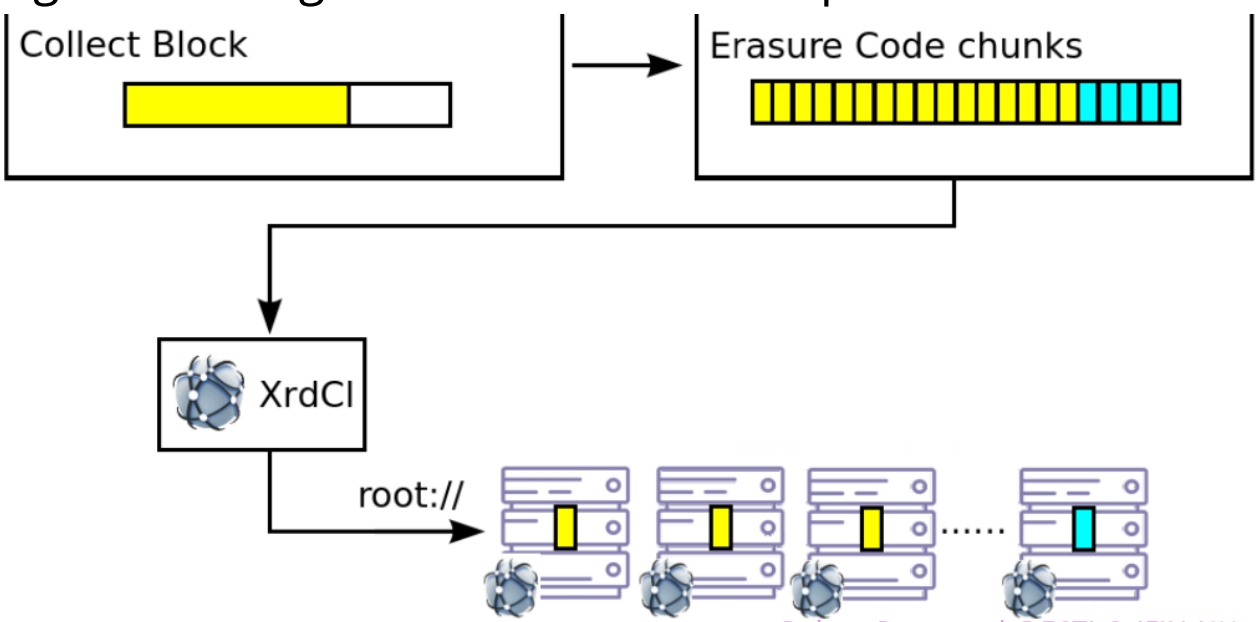
Declarative API: a case study

- For testing that the newly developed API is indeed a more efficient tool in terms of the codebase ➡️ a special mechanism was implemented (as a plug-in for the client): **Erasure Coding (EC)**
- EC is a method of data protection:
  - data is broken into fragments, expanded and encoded with redundant data pieces
  - data is then stored across a set of different locations or storage media.



Erasure Coding

- In terms of development workflow:
  - One needs to open all stripes
  - Write to all stripes
  - Set extended attributes on all stripes (e.g., checksums)
  - Close all stripes.
- For efficiency, the *writing* and *setting extended attributes* operations should be done in **parallel.**



Erasure Coding

If the standard asynchronous API would've been used...

```
using namespace XrdCl;

/*
 * Write to a single chunk
 */
void ECWrite(uint64_t          offset,
             uint32_t          size,
             const void       *buff,
             ResponseHandler *userHandler)
{
  // translate arguments to chunk specific parameters
  // ...
  File *file=new File();
  OpenHandler *handler=
    new OpenHandler(file,userHandler,/*long list of arguments*/);
  // although we do a write in here we only see an open call,
  // all the logic is hidden in the callback and the workflow
  // is unclear
  file->Open(url,flags,handler);
}
```

Plus a lot of boilerplate code for:

1. **handler logic**
2. **parallel execution of the writes**
3. **parallel execution of the xtatrrs**

Erasure Coding

If the standard asynchronous API would've been used...

```
using namespace XrdCl;

class CloseHandler : public ResponseHandler
{
  CloseHandler(File *file,/*other arguments*/){ /*...*/ }

  void HandleResponse(XRootDStatus *st, AnyObject *rsp)
  {
    // 1: validate status and response first
    // ...
    // 2. call the end-user handler
    userHandler->HandleResponse(st,rsp);
  }

  // members
  // ...
}

class XAttrHandler : public ResponseHandler
{
  XAttrHandler(File *file,/*other arguments*/){ //... }

  void HandleResponse(XRootDStatus *st, AnyObject *rsp)
  {
    // 1. validate status and response first
    // ...
    // 2. proceed to the next operation
    CloseHandler *handler = new CloseHandler(file,/*...*/)
    file->Close(handler,/*...*/)
  }

  // members
  // ...
}
```

```
class WrtHandler : public ResponseHandler
{
  WrtHandler(File *file,/*other arguments*/){ //... }

  void HandleResponse(XRootDStatus *st, AnyObject *rsp)
  {
    // 1. validate status and response first
    // ...
    // 2. proceed to the next operation
    XAttrHandler *handler = new XAttrHandler(file,/*...*/)
    file->SetXAttr("xrdec.chsum",checksum,handler);
  }

  // members
  // ...
}
```

```
class OpenHandler : public ResponseHandler
{
  OpenHandler(File *file,/*other arguments*/){ //... }

  void HandleResponse(XRootDStatus *st, AnyObject *rsp)
  {
    // 1. validate status and response first
    // ...
    // 2. proceed to the next operation
    WrtHandler *handler = new WrtHandler(file,/*...*/)
    file->Write(offset,size,buffer,handler,/*...*/)
  }

  // members
  // ...
}
```

What do we have so far:

- **We updated only one chunk**
- **Write and SetXAttr happen sequentially (we would need another handler-class to aggregate the result of parallel execution)**
- **To update all data stripes and parity stripes we will need yet another handler-class to cope with parallel execution.**
- **Boilerplate code -> very repetitive**

Erasure Coding

```
using namespace XrdCl;

// Write erasure coded block
void ECWrite(uint64_t           offset,
             uint32_t           size,
             const void        *buffer,
             ResponseHandler *userHandler)
{
  std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
  for(size_t i=0;i<nbchunks;++i)
  {
    // calculate offset, size and buffer for each stripe/chunk
    // ...
    File *file=new File();
    Pipeline p=Open(file,url,flags)
              | Parallel(Write(file,choff,chsize,chbuff),
                         SetXAttr(file,"xrdec.cksum",checksum))
              | Close(file)>>[file](XRootDStatus&){delete file;}
  }
  // Execute the workflow!
  Async(Parallel(wrts) >>
        [userHandler](XRootDStatus& st)
        {userHandler->HandleResponse(new XRootDStatus(st),0);});
}
```

1. **clean code**
2. **concise workflow**
3. **easy to read**

Conclusions

- An overview of the XRootD framework was provided
  - Server-side
  - Client-side

  } functionality
- Standard C++ API was presented (File and FileSystem APIs)
  - shows some drawbacks in terms of code complexity
- Declarative API -> improve the File and FileSystem APIs:
  - simplifies the workflow in terms of actual code-base.
  - Might be an efficient tool when developing mechanisms that require multi-stage & parallel workflows.