PyXRootD PyPI distribution and new declarative file access API for XRootD Client

AUGUST 2018

AUTHOR:
Krzysztof Jamróg

CERN IT-ST-AD

SUPERVISORS(S):
Michał Simon
Lars Nielsen

Abstract

The project described in this report is related to XRootD framework development. It was divided into two parts. First part was about publishing XRootD python bindings called PyXRootD to Python Package Index. This makes PyXRootD installation much easier and resolves problem with versioning. Second part was about creating new API for file operations, which are one of mainly used components of XRootD framework. Introduced API provides more readable and convenient way of using asynchronous functions available in XRootD.

This report describes the motivation behind both parts of the project and their implementation.

Acknowledgements

First of all, I would like to thank my supervisors Michał and Lars for giving me the opportunity to work on this project, for their great support and a lot of knowledge I got from them.

I would also like to thank the whole CERN Openlab team for the organisation of this programme and a lot of help with addministrative issues.

Last but not least I would like to thank all summer students for making this summer a great experience.

Contents

Contents iv

1 Introduction 1

2 PyXRootD PyPI distribution 2
2.1 The previous installation process and its problems . . . 2
2.2 New approach . . . 2
2.3 Integration with GitLab CI . . . 3

3 New file access API 4
3.1 File and FileSystem - file access API in XRootD . . . 4
3.2 Asynchronous API problems . . . 5
3.3 New solution . . . 7
3.3.1 Declarative syntax . . . 7
3.3.2 Compile time checking of workflow declaration . . . 8
3.3.3 Error handling . . . 8
3.3.4 Passing arguments between operations . . . 8
3.3.5 Application and tests . . . 10

4 Conclusions 11

Bibliography 12

1. Introduction

One of the main challenges for high-energy physics community working at CERN is dealing with huge amount of data generated each day. The data is distributed across multiple data servers or even multiple sites, therefore there is a need for software allowing users to access remote data in easy and efficient way. XRootD is a low latency file access framework of choice for High Energy Physics community and the backbone of the EOS project, the main storage solution used at CERN.

The framework is plugin based what makes it very customizable and therefore allows to meet even very specific requirements.

It has client-server architecture, where the former is responsible for aggregating multiple data storages and providing API to access them, and the latter is designed to use this exposed API to access remote data from the local machine.

The software is written in C++, but from the end user's perspective one of the key components of XRootD client are python bindings which allow to use all of its functionalities from Python scripts. These bindings are called PyXRootD.

The project described in this report is related to XRootD client and its python bindings. The first chapter describes new PyXRootD installation process and the motivation to introduce it. The second chapter is related to new file access API. It describes its key functionalities with some implementation details and usage examples.

2. PyXRootD PyPI distribution

This chapter describes the motivation of publishing PyXRootD to Python Package Index as well as current distribution and installation process

2.1 The previous installation process and its problems

Until now there were two ways of installing PyXRootD. First way was to use package management utilities. There are two of them, where this package is available:

- Official XRootD YUM repository
- EPEL (Extra Packages for Enterprise Linux)

However, PyXRootD is sometimes installed in environments, in which these tools are not available. In these cases, the second way of installation is used, which involves manually downloading and compiling the source code. The typical procedure used to installing PyXRootD in this way is shown below:

```
git clone https://github.com/xrootd/xrootd.git
mkdir build
cd build
cmake ../xrootd/. -DCMAKE_INSTALL_PREFIX=/installation/path
cd bindings/python
make
make install
python setup.py install
```

Listing 2.1: PyXRootD manual installation steps

There are several problems with both approaches described above. First of all, there is no support for python virtual environments, which are commonly used not only in testing environments, but also in production. It is user's responsibility to copy package files to appropriate folder so that it could be visible in a given virtual environment. The second problem is related to versioning. Especially in the second approach, the only way to install specific version is to change a branch after cloning the repository, to one that contains desired version. This requires some additional knowledge about branches naming convention, which is problematic for users who do not belong to the development team. The last problem is that these installation ways are different than classic python package installation approach. That means this process cannot be easily automated so the package needs to be installed separately which might sometimes be inconvenient.

2.2 New approach

Before moving to the description of new way of PyXRootD installation, it is worth to have a general overview of what PyPI and PIP are. PyPI is the official repository for Python libraries. At

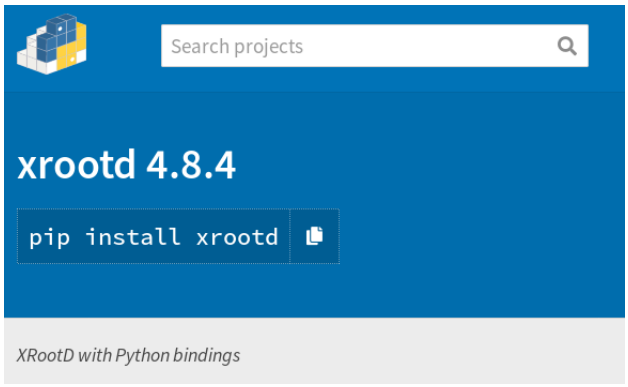

Figure 2.1: PyXRootD uploaded to PyPI

the time of writing this report it contains over 150 thousands of different packages. All of those packages can be installed by PIP which is a package management system used to install and manage Python packages.

The solution to the problems described in the previous section was to upload PyXRootD to PyPI (Fig. 2.1). After that the installation is very easy and can be done using PIP:

```
pip install xrootd==4.8.4
```

There is also no problem with versioning anymore. Version can be specified in installation command as shown above. PIP allows to automate installation of entire group of packages by creating properly formatted requirements.txt file and installing it as following:

```
pip install -r requirements.txt
```

That allows to install PyXRootD automatically with other Python packages used in the given project.

2.3 Integration with GitLab CI

The process of publishing PyXRootD to PyPI has been integrated with GitLab CI/CD (Continuous Integration & Deployment) tool. This was done by adding job which builds required archives whenever new version of XRootD framework is published.

3. New file access API

3.1 File and FileSystem - file access API in XRootD

As XRootD is mainly used with file-based repositories, one of its most important components is the file access API. This utility contains both single file and file system functions.

There are two versions of each of those functions: synchronous and asynchronous. The main difference from the API perspective is that asynchronous functions take one more argument which is a handler object. This object implements specific interface and its methods are called when the response from asynchronous function call arrives. The example of two versions of Write method are shown in Listings 3.1 and 3.2. Listing 3.3 shows example handler implementation.

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

Listing 3.1: Synchronous version of Write method

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

Listing 3.2: Asynchronous version of Write method

```
class SimpleHandler: public ResponseHandler {
    public:
        void HandleResponse(XrdCl::XRootDStatus *status, XrdCl::AnyObject *response){
            //------------------------------------------------
            //! Perform operations using response object
            //------------------------------------------------
            delete status;
            delete response;
            delete this;
        }
};
```

Listing 3.3: Example handler class

3.2 Asynchronous API problems

Asynchronous API in many cases can be much more efficient than synchronous code. As well known, in case of asynchronous functions, program does not wait for the function execution end and it is the handler which is responsible to control the execution flow. To perform specific operation on a file, usually there is a need to call at least few functions in specific order. For example reading a file usually consists of the following steps:

- Opening file
- Reading file content
- Closing file

To ensure proper execution flow, each next function needs to be called from the previous function's handler. The typical flow of the case described above is shown in Figure 3.1. At first *Open* operation is called. After its execution is finished, provided handler is run. Inside the handler, the second operation (*Read*) is called with another handler in which the third operation (*Close*) is called.

Defining that relatively simple flow requires quite a lot of work. At first, each operation call requires different arguments so each handler needs to have different implementation. That means for each handler user needs to define separate class. Furthermore, in each handler the operation status needs to be checked and potential errors should be handled. This process becomes even more difficult in case of more complex flow including parallel operations, as shown in Figure 3.2.

Apart from complexity, there is also another problem with current asynchronous API, which is code readability. The example usage of asynchronous function is shown in Listing 3.4. As it can be seen, at the first sight only first operation call can be seen and further execution is done in handler. That means to understand whole execution flow user would need to go through set of handlers what can be very inconvenient and time consuming.

```
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
```

Listing 3.4: Asynchronous usage example

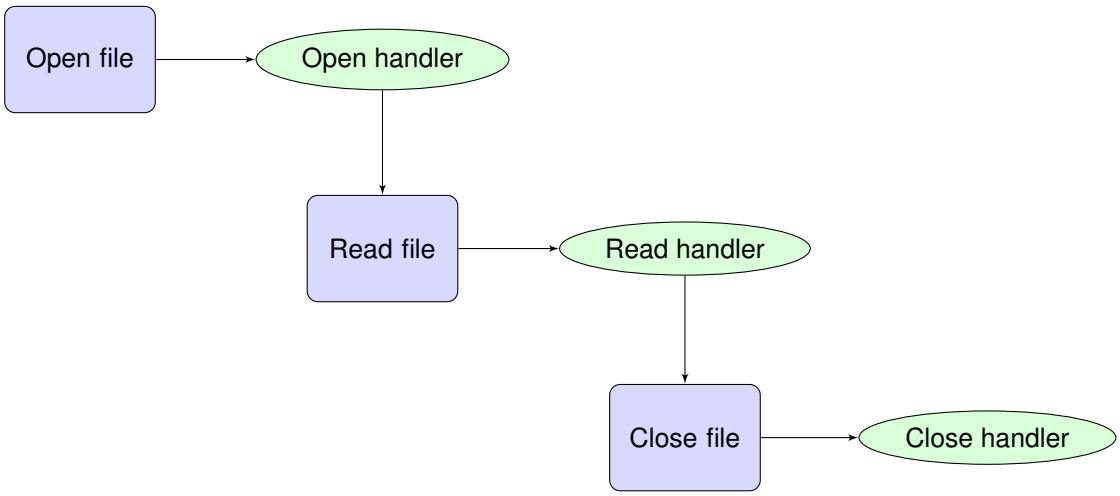

Figure 3.1: File reading operations flow

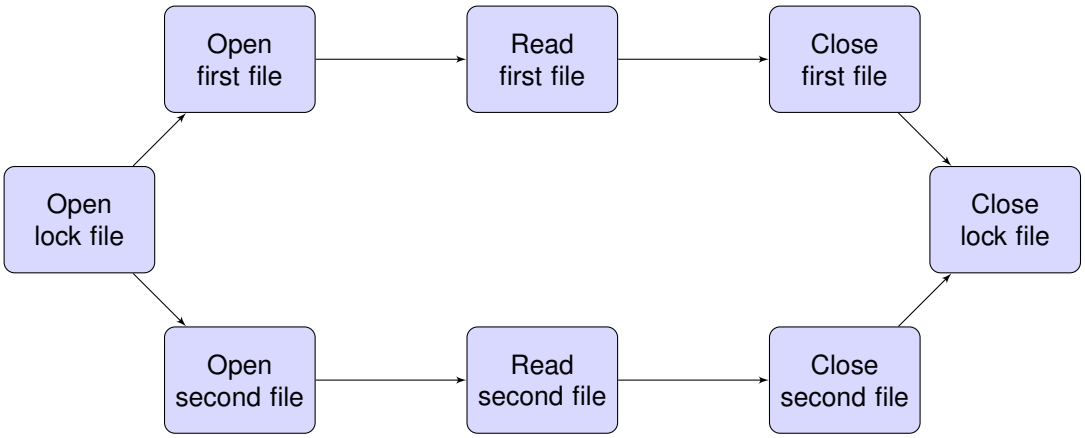

Figure 3.2: More complex file operations flow (handlers are skipped for readability)

3.3 New solution

The goal of that part of the project was to make usage of asynchronous API easier and thus mitigate the problems described in previous section. For this purpose, new declarative API has been designed and implemented. It has been built on top of existing API and provides additional level of abstraction for the end user. This section describes the main parts and characterictics of introduced mechanism as well as its usage examples and some implementation details.

There are several requirements which needed to be met to make this API the easiest and the most useful. At first, it should be possible to define whole operations workflow in one place without a need to spread the logic into many classes. Secondly, the syntax should be as easy and declarative as possible so that users could focus on what operations they want to perform without caring about proper execution flow. What is more, as the syntax can be very specific, user should be notified about potentially incorrect configuration during compilation time. The next important thing is error handling. As the whole workflow declaration is supposed to be done in one place, it should also be possible to check for potential errors in the same place. The challenge was to meet all of these requirements and at the same time have all functionality which was there before. So for example, it should be still possible to create communication between operations so that the result of one operation could be used to compute arguments for the next operation.

3.3.1 Declarative syntax

The syntax for workflows declaration was created with usage of operators overloading utility available in C++. The example of its usage is shown in Listing 3.5.

```
uint64_t offset = 0;
uint32_t size = 50;
char* buffer = new char[size]();
const string path = "/tmp/testfile.txt";
const OpenFlags::Flags flags = OpenFlags::Read;
const Access::Mode mode = Access::None;

File *file = new File();

auto readHandler = new ResponseHandler();

auto &pipeline = Open(file)(path, flags, mode)
                 | Read(file)(offset, size, buffer) >> readHandler
                 | Close(file)();

Workflow workflow(pipeline);
workflow.Run().Wait();
```

Listing 3.5: Workflow declaration syntax

First few lines of this code snippet contain operations parameters declarations. Then the operations flow is declared and assigned to the *pipeline* variable. The declaration consists of the following elements: for each operation the correcponding object is created. Then the () operator is used to pass arguments to the operation. The number of arguments and their types are directly dependant on the operation. After that the >> operator can be used to provide a custom handler for the given operation. However this is optional and right now the handler should be used only to perform some additional actions with operation result. It is no longer responsible for controlling the flow. This mechanism also has support for lambda functions which can be used instead of defining separate class. In this example the handler is specified only for *Read* operation. Operations defined in this way can be connected by | operator. After defining the pipeline the *Workflow* object is created. It is a wrapper responsible for running operations, storing the result of the execution and controlling the flow using semaphores mechanism. After creation of this

object the *Run()* method is called to execute defined operations. *Wait()* method stops current thread until the whole workflow will be executed.

Provided syntax allows also to define parallel execution of two operations flow. For that purpose, dedicated *Parallel* class is used. The example usage of this functionality is shown in Listing 3.6. It implements the flow shown in Figure 3.2.

```
auto &firstPipe = Open(firstFile)(firstFilePath, flags)
                | Read(firstFile)(offset, size, firstBuffer)
                | Close(firstFile)();

auto &secondPipe = Open(secondFile)(secondFilePath, flags)
                 | Read(secondFile)(offset, size, secondBuffer)
                 | Close(secondFile)();

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

Workflow workflow(pipe);
workflow.Run().Wait();
```

Listing 3.6: Workflow with parallel operations (parameters declarations are skipped for readability)

As it can be seen it is used in the same way as normal operation, the only difference is that its constructor accepts the collection of operations and there is no need to use $()$ operation as this is not an operation thus does not need to take arguments.

3.3.2 Compile time checking of workflow declaration

Workflow declaration syntax is very readable and easy to understand. However it has very specific syntax which needs to be followed. That is why the compile time declaration checking has been introduced so that user will be notified about malformed declaration during compilation. This has been done with usage of template metaprogramming and static assertions. Going into details, just after creation operation object has different template type than the types returned by $()$ and $>>$ operators and different methods are available. Therefore for example adding handler to the operation which is not configured will not be possible and will end up with the following error message:

static assertion failed: Operator >> is available only for type Operation<Configured>

3.3.3 Error handling

As in proposed mechanism it is no longer neccessary to provide handlers, there is a need to have the possibility to handle failed operations statuses in the same place in which the workflow is defined. Therefore in internal workflow implementation status of each operation is checked and if it is not correct, then workflow execution ends and this failed status is saved as a status of workflow. Otherwise the next operation is run and if everything goes correctly the status of the last operation will be saved as the workflow status. It can be accessed by *GetStatus()* method implemented in *Workflow* class.

3.3.4 Passing arguments between operations

Ofter there is a need to use the result of one operation as an argument for another operation. For example *Stat* operation can return information about the size of the file which can be used in *Read* operation. For that purpose the mechanism of passing arguments between operations has been implemented. Parameters passing can be done in handler. For that be possible handler

needs to inherit from *ForwardingHandler* class, which contains *ForwardParam* method dedicated for that purpose. This is a template method in which user needs to specify forwarded argument type as a template parameter and its value as a function argument. The example implementation of such handler is shown in Listing 3.7.

```
class StatHandler: public ForwardingHandler {
    public:
        void HandleResponse(XrdCl::XRootDStatus *status, XrdCl::AnyObject *response) {
            StatInfo *stat = 0;
            response->Get(stat);
            uint32_t size = stat->GetSize();
            char* buffer = new char[size]();
            ForwardParam<Read::BufferArg>(buffer);
            ForwardParam<Read::SizeArg>(size);

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

Listing 3.7: Handler with parameters forwarding

To enable usage of forwarded parameter, dedicated *notdef* object needs to be provided to operation configuration function instead of real value (Listing 3.8).

```
auto statHandler = new StatHandler();

auto &pipe = Open(f)(fileUrl, flags)
           | Stat(f)(true) >> statHandler
           | Read(f)(offset, notdef, notdef)
           | Close(f)();

Workflow workflow(pipe);
workflow.Run().Wait();
```

Listing 3.8: Workflow with notdef params declaration

Parameters forwarding can also be done with lambda functions, as shown in Listing 3.9

```
auto statHandler = [](XRootDStatus &st, StatInfo& stat, ParamsContainerWrapper& params){
    uint32_t size = stat.GetSize();
    char* buffer = new char[size]();
    params.ForwardParam<Read::BufferArg>(buffer);
    params.ForwardParam<Read::SizeArg>(size);
};

auto &pipe = Open(f)(fileUrl, flags)
           | Stat(f)(true) >> statHandler
           | Read(f)(offset, notdef, notdef)
           | Close(f)();

Workflow workflow(pipe);
workflow.Run().Wait();
```

Listing 3.9: Forwarding parameters using lambda function

```
Selected tests/
  Selected tests/WorkflowTest/
    Selected tests/WorkflowTest/WorkflowTest::ReadingWorkflowTest
    Selected tests/WorkflowTest/WorkflowTest::WritingWorkflowTest
    Selected tests/WorkflowTest/WorkflowTest::MissingParameterTest
    Selected tests/WorkflowTest/WorkflowTest::OperationFailureTest
    Selected tests/WorkflowTest/WorkflowTest::DoubleRunningTest
    Selected tests/WorkflowTest/WorkflowTest::ParallelTest
    Selected tests/WorkflowTest/WorkflowTest::FileSystemWorkflowTest
    Selected tests/WorkflowTest/WorkflowTest::MixedWorkflowTest

Running:

OK (8)
........
```

Figure 3.3: Unit tests results

3.3.5 Application and tests

All of the functionalities described above are applied to all file and file system functions available in XRootD client. The mechanism has been tested with unit tests which result is shown in Figure 3.3.

4. Conclusions

In this project PyXRootD packages has been successfully published to Python Package Index. This allows to use PIP to install this package what makes installation process much more convenient and easier to automate. Furthermore, PIP provides support for python virtual environments and proper version management.

Additionaly, new declarative file access API for XRootD client has been introduced. Its clear syntax makes usage of asynchronous file and file system access functions much easier and the code much more readable.

Bibliography

[1] *XRootD repository* https://github.com/xrootd/xrootd

[2] *XRootD Home Page* http://xrootd.org/

[3] *PyXRootD package in PyPI* https://pypi.org/project/xrootd/

[4] *Python Packaging User Guide* https://packaging.python.org/