
CERN

Michal Simon

The new XRootD client declarative API

Outline

- Motivation
- Overview
- Pipelines vs bundling

Motivation

- Use case: erasure coding plug-in for EOS
  - Executing multiple operations on multiple files (stripes) in parallel
- Problem with **asynchronous operation composability and code readability**
  - Asynchronous `Open() + Read() + Close()` in the code is only visible as an `Open()` (rest of the workflow is in the callbacks)
- **In line with modern C++** (ranges v3 inspired, support for *`Lambdas`*, `std::futures`)

Overview: example

Example: `open | read | close`

```
1
2   File f;
3   std::future<ChunkInfo> resp;
4
5   // open, read from and close the file
6   Pipeline p  = Open(file,url,OpenFlags::Read)
7               | Read(file,offset,size,buffer) >> resp
8               | Close(file);
9
10  auto status = WaitFor(p);
11
```

Overview: synchronous vs asynchronous

Synchronous execution

```
1
2  FileSystem fs(url);
3  XRootDStatus status = WaitFor( Truncate(fs,path,size) );
4
```

Asynchronous execution

```
1
2  FileSystem fs(url);
3  std::future<XRootDStatus> status =
4                  Async( Truncate(fs,path,size) );
5
```

Overview: Fwd utility

```
1
2   Fwd<uint32_t> size; // size of the file
3   Fwd<void*>    buff; // buffer for the data
4
5   std::future<ChunkInfo> resp; // server response
6
7   auto &&p = Open(file,url,OpenFlags::Read) >>
8              [size,buff](XRootDStatus &status,StatInfo &info)
9                {
10                 if(!status.IsOK()) return;
11                 size = info.GetSize(); // forward size and
12                 buff = new char[info.GetSize()]; // buffer
13               }
14         | Read(file,0,size,buff) >> resp
15         | Close(file);
16
17  auto status = WaitFor( p );
18
```

Overview: Parallel execution

```
1
2   auto &&o1    = Open(file1,url1,OpenFlags::Read);
3   auto &&o2    = Open(file2,url2,OpenFlags::Read);
4   auto &&o2    = Open(file2,url2,OpenFlags::Read);
5
6   // open 3 files in parallel
7   Pipeline p   = Parallel(o1,o2,o3);
8
9   auto status = WaitFor( p );
10
```

Overview: *XrdCl::ResponseHandler*

```
1
2   class ExampleHandler : public ResponseHandler
3   {
4     public:
5       void HandleResponse(
6               XRootDStatus *status, // status of the operation
7               AnyObject *response   // server response (type erased)
8             )
9       {
10        // handle the operation here
11      }
12  };
13
14  ...
15
16  FileSystem fs(url);
17  ExampleHandler hndl;
18
19  auto status = WaitFor( Stat(fs,path) >> hndl );
20
```

Overview: functions / functors / lambdas

```
1
2   void ExampleHandler(
3           XRootDStatus &status, // status of the operation
4           StatInfo     &info    // server response (explicit type)
5         )
6   {
7     // handle the operation here
8   }
9
10  ...
11
12  FileSystem fs(url);
13  // could also be a lambda or function object !!!
14  auto status = WaitFor( Stat(fs,path) >> ExampleHandler );
15
```

Overview: *std::future*

```
1
2   File file;
3   std::future<ChunkInfo> resp;
4
5   Async( Read(off,size,buff) >> resp );
6
7
8   ...
9
10  // later on process the future
11  try
12  {
13    ...
14    // if everything went OK we will get the ChunkInfo,
15    // otherwise it will throw
16    ChunkInfo chunk = resp.get();
17    ...
18  }
19  catch( PipelineException &ex )
20  {
21    // we will learn the reason for failing from
22    // the status object
23    XRootDStatus &status = ex.GetError();
24    ...
25  }
26
```

Overview: *std::packaged_task*

```
1
2   using namespace std;
3
4   packaged_task<uint64_t(XRootDStatus &st,StatInfo &info)> parse =
5       [](XRootDStatus &st, StatInfo &info)
6         {
7           if(!st.IsOK) throw PipelineException(st);
8           return info.GetSize();
9         };
10
11  FileSystem fs(url);
12  future<uint64_t> size = parse.get_future();
13
14  Async( Stat(fs,path) >> parse );
15
16  // later on use size the same way as
17  // the future from previous example
18
```

Overview: final example

Example: `lock | open | read | close | unlock`

```
1
2   File lock, file;
3   FileSystem fs(url);
4   std::future<ChunkInfo> resp; // server response
5
6   auto &&p = Open(lock, "root://host//path/to/.lock", OpenFlags::New)
7            | Close(lock)
8            | Open(file, "root://host//path/to/file.txt",OpenFlags::Read)
9            | Read(file,0,1024,buff) >> resp
10           | Close(file);
11           | Rm(fs, "root://host//path/to/.lock");
12
13  // we can already pass resp to an algorithm for processing
14
15  // we wait for the pipeline to complete
16  auto status = WaitFor( p );
17
```

XrdCl Pipelines vs bundling

- Once bundling is implemented on the protocol level it could be exposed to the users e.g. by overloading ‘+’ operator
- For example:

```
open + read + close
```

Questions?