
CERN

Michal Simon

# Exploring the virtues of XRootD5: Declarative API

# Outline

- Motivation & Use Cases
- Declarative API overview
- Control directives
- Summary

# Motivation

- OO **synchronous API:** easy to use, good code readability, but **no composability**
  - Concurrent I/O only through multithreading
- OO **asynchronous API:** operation chaining, **poor readability,** huge amount of **boilerplate**
- Driven by two major use cases
  - client **erasure coding plugin for Alice O2** project
    - Requires parallel remote I/O for every operation
  - **ZIP archive** metadata parser
    - Needs to chain consecutive reads to parse metadata

# Declarative API overview

- Every two or more **I/O operations can be composed with | operator** forming a pipeline
- Every operation optionally may be **assigned with a completion handler using >> operator**
  - ***std::future***, **lambda**, *ResponseHandler*, etc.
- *Pipeline* class: polymorphic wrapper for I/O operations
  - Instance of ***Pipeline* can store any I/O operation**
- Triggered either with ***WaitFor*** or ***Async*** utility functions
- *Fwd* class: utility for **forwarding values** between completion handlers and I/O operations
  - Think of it as *std::shared_ptr* to *std::optional*

# Rules of pipelining

- Operations within a pipeline are **associative**
- Defining a pipeline does not trigger it (in a sense, it is **lazy evaluated**)
- Once executed, operations in the pipeline are **performed strictly from left to right**
- **On operation failure the pipeline stops**, any subsequent operations are ignored

```
1
2 std::future<ChunkInfo> ftr;
3 Pipeline p = Open(file, url, OpenFlags::Read)
4            | Read(file, off, len, buf) >> ftr
5            | Close(file);
6
```

# Control directives

- Control directives have to be **invoked from within a completion handler** and allow to **alter pipeline execution**
  - ***Stop***: stop the pipeline immediately
  - ***Repeat***: repeat current operation
  - ***Ignore***: ignore error and proceed with execution
  - ***Replace***: replace current operation or whole pipeline

# Loop: print remote file

```
1
2  std::shared_ptr<File> file=std::make_shared<File>();
3  Fwd<uint64_t> off = 0; // forwardable!!!
4  uint32_t      len = 1024;
5  char*         buf = new char[len+1];
6  Pipeline p = Open(file,url,OpenFlags::Read)
7             | Read(file,off,len,buf) >>
8               [off](auto& status,auto& chunk)
9               {
10                if(!status.IsOK())
11                  Pipeline::Ignore(); // proceed to close
12                if(chunk.length == 0) return; // EOF
13                std::cout << std::string(chunk.buffer,chunk.length);
14                // adjust the offset
15                off = *off+1024;
16                // repeat until EOF
17                Pipeline::Repeat();
18              }
19            | Close(file) >> [file](auto& st){};
20 Async(std::move(p));
21
```

# Recursion example

- Recursively try opening one of the redundant file replicas

```
1
2  auto TryOpen(File &f,const std::vector<std::string> &urls,size_t i=0)
3  {
4    return Open(f,urls[i],OpenFlags::Read) >> [&f,&urls,i](auto &status)
5    {
6      if(status.IsOK()) return; // we found valid replica
7      if(i+1>=urls.size()) return; // there are no more replicas to try
8      // recover error  next replica
9      Pipeline::Replace(TryOpen(f,urls,i+1));
10   };
11 }
```

# Parallel operation

- ***Parallel*** operation allows to aggregate several component operations for parallel execution

```
1
2 Pipeline p = Open(file,url,OpenFlags::Read)
3            | Parallel( // also works with containers
4                Read(file,off1,len1,buf1),
5                Read(file,off2,len2,buf2)
6              )
7            | Close(file);
8
```

# Final operation

- ***Final*** operation is always guaranteed to be executed and MUST be the last operation in the pipeline

```
1
2  std::shared_ptr<File> file=std::make_shared<File>();
3  uint64_t off = 0;
4  uint32_t len = 1024;
5  char*    buf = new char[len+1];
6  Pipeline p = Open(file,url,OpenFlags::Read)
7             | Read(file,off,len,buf) >> [](auto &status, auto &chunk)
8               {
9                 if(!status.IsOK()) return;
10                std::cout << std::string(chunk.buffer,chunk.length);
11              }
12            | Close(file)
13            | Final([file,buf]{delete[] buf;}); // deallocate resources
14
```

# Summary

- The declarative API
  - allows for **better-structured code** (confirmed by software metrics)
  - gives a standard way of **composing asynchronous I/O operations**
  - provides a set of **control directives** for dynamically altering running I/O pipelines
  - is in line with **modern C++** programming practices

# Questions?

# Declarative API vs coroutines

- **Why don’t we just use coroutines?!?**
  - They are available **only in C++20** (and we just managed to move to C++14)
  - Coroutines don’t really provide operation chaining (although they are great for async programming)
  - For remote I/O, we can **optimize RTTs by bundling requests** (consider open + write + close)
  - For local I/O, once io_uring is available we can **reduce the number of syscalls**
- If there’s interest we can provide an `awaitable` interface for our users