CERN

Michal Simon

What's up with the XRootD client

Outline

- xrdcp primer
- Declarative API
- Lifting File API limitations
- Record & replay

xrdcp primer: TLS support

- Triggering TLS with **roots://**

```
$ xrdcp roots://src.cern.ch//path//file roots://dst.cern.ch//path/dir
```

- Encrypting only the **control** channel

```
$ xrdcp --tlsnodata roots://src//file roots://dst//dir
```

- **Backwards compatibility** with old servers

```
$ xrdcp --notlsok roots://src//file roots://dst//dir
```

- Use encryption with **metalinks**

```
$ xrdcp --tlsmetalink roots://src//file roots://dst//dir
```

xrdcp primer: ZIP archives

- Copy file from a ZIP archive

```
$ xrdcp --zip file roots://src//arch.zip roots://dst//dir
```

- Append file into a ZIP archive

```
$ xrdcp --zip-append roots://src//file roots://dst//dir/arch.zip
```

- Use checksum from the Metalink

```
$ xrdcp --cksum zcrc32 --zip file --zip-mtln-cksum \
        root://src//arch.zip root://dst//dir
```

xrdcp primer: continue & retry

- **Continue** timed out transfer

```
$ xrdcp --continue root://src//file root://dst//dir
```

- **Retry** errors

```
$ xrdcp --retry roots://src//file roots://dst//dir/
```

- Retry + **force**

```
$ xrdcp --retry --retry-policy force \
        roots://src//file roots://dst//dir
```

- Retry + **continue**

```
$ xrdcp --retry --retry-policy continue \
        roots://src//file roots://dst//dir
```

xrdcp primer: transfer rate

- Limit the maximum transfer rate

```
$ xrdcp --xrate 150m root://src//file root://dst//dir
```

- Ensure the transfer rate does not drop below given threshold

```
$ xrdcp --xrate-threshold 50m root://src//file root://dst//dir
```

xrdcp primer: miscellaneus

- Support two checksums

```
$ xrdcp --cksum adler32 --cksum md5:print \
        root://src//file root://dst//dir
```

- Cleanup the file on bad checksum

```
$ xrdcp --cksum adler32 --rm-bad-cksum \
        root://src//file root://dst//dir/
```

- Multiple sources (extreme copy)

```
$ xrdcp -y 8 root://src//file root://dst//dir
```

- Preserve extended attributes

```
$ xrdcp --xattr root://src//file root://dst//dir
```

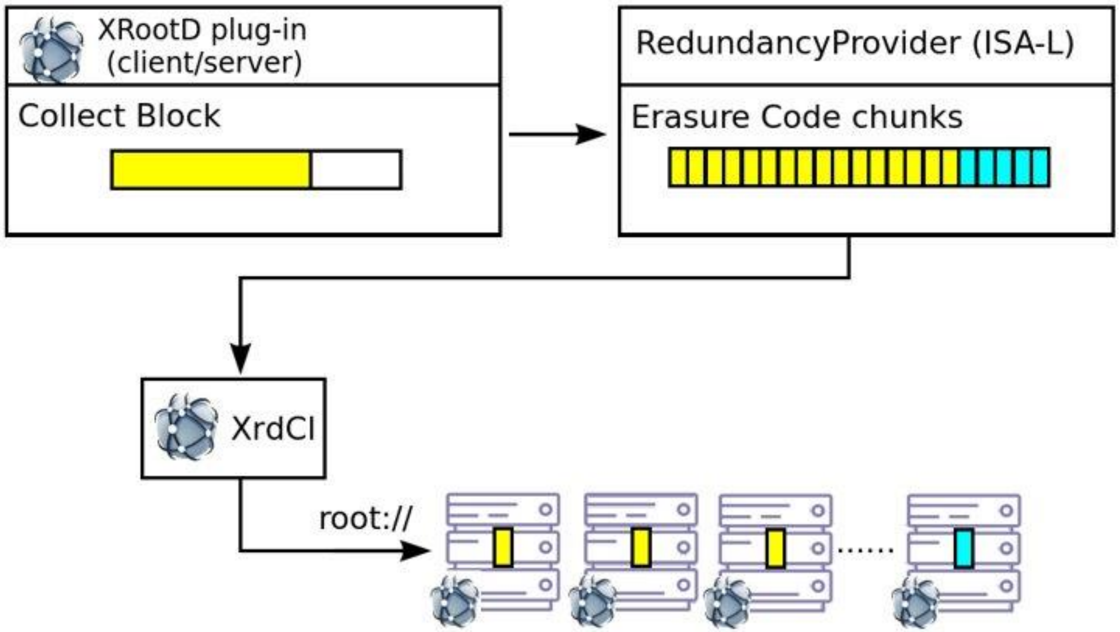
Declarative API: Motivation

- Use case: erasure coding plug-in for EOS
  - Executing multiple operations on multiple **remote** files (stripes) in parallel
- Problem with **asynchronous operation composability and code readability**
  - Asynchronous `Open() + Write() + Close()` in the code is only visible as an `Open()` (rest of the workflow is in the callbacks)

Declarative API: case study

We would like to implement a ECWrite() method based on XRootD client API

- Write one block **striped to *n* data chunks and *m* parity chunks**



Declarative API: case study

- We need to **open** all stripes, **write** to all stripes, **set extended attributes** on all stripes (e.g. checksum), **close** all stripes
- Ideally, for performance we would like to use only **asynchronous** APIs
- The **write** operation **and setting extended attributes** should be done **in parallel**

Declarative API: case study

Update of a single stripe/chunk with standard XrdCl API ...

```
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);
}
```

Declarative API: case study

… also all this boilerplate code is needed!

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

Declarative API: case study

What do we have so far:

- We updated **only one chunk**
- Write and SetXAttr happen **sequentially** (we would need **yet another handler-class t**o aggregate the result of parallel execution)
- The amount of **boilerplait code is SIGNIFICANT!!!**
- To update all data stripes and parity stripes we will need **yet another handler-class** to cope with parallel execution
- The boilerplait code is very repetitive!

Declarative API: case study

We extracted the repeating patterns, applied significant amount of template meta-programming and got a new declarative API:

- **Asynchronous operation composability**
- **Code readability**
- **Clear workflow**
- **In line with modern c++** (ranges v3 inspired, support for *`lambdas`*, `std::futures`)
- Released in 4.9.0 but more complete set of features available only in 5.0.0

Declarative API

```
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);});
}
```

Declarative API

```
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
  Async(Parallel
       [userHa                t)
       {userHa                new XRootDStatus(st),0);});
}
```

Compose operations
with | operator!

Declarative API

```
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](X
       {userHandler->H                    tDStatus(st),0);});
}
```

Parallel execution of operations!

Declarative API

```
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          uffer for each stripe/chunk
    // ...
    File *file=new F
    Pipeline p=Oper
              |Par                  ,chsize,chbuff),
                                 xrdec.cksum",checksum))
              |Close(file)>>[file](XRootDStatus&){delete file;}
  }
  // Execute the workflow!
  Async(Parallel(wrts)>>
        [userHandler](XRootDStatus& st)
        {userHandler->HandleResponse(new XRootDStatus(st),0);});
}
```

Parallel execution of a container of operations

Declarative API

```
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)
  {
    //         offset, size and buffer for each stripe/chunk

                    e();
                    e, url, flags)
                    (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);});
}
```

Specify async
callback with
>> operator

Declarative API

```

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)
  {
    /              offset, size and buffer for each stripe/chunk

                   e();
                   e,url,flags)
                   (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);});
}

```

Use lambdas (or std::future) as callbacks

Declarative API

```
using namespace XrdCl;

// Write erasure cod
void ECWrite(uint64_
             uint3
             const
             Respon                  ler)
{
  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);});
}
```

First prepare the workflow

Declarative API

```
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= [illegible] , flags)
                [illegible] (file, choff, chsize, chbuff),
                [illegible] (file,"xrdec.cksum",checksum))
                [illegible] ](XRootDStatus&){delete file;}
  }
  // Execute [illegible]
  Async( Parallel(wrts) >>
        [userHandler](XRootDStatus& st)
        {userHandler->HandleResponse(new XRootDStatus(st),0);});
}
```

Execute later!

Declarative API

```
using namespace XrdCl;

// Write erasure code
void ECWrite(uint6
             uint3
             const
             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);});
}
```

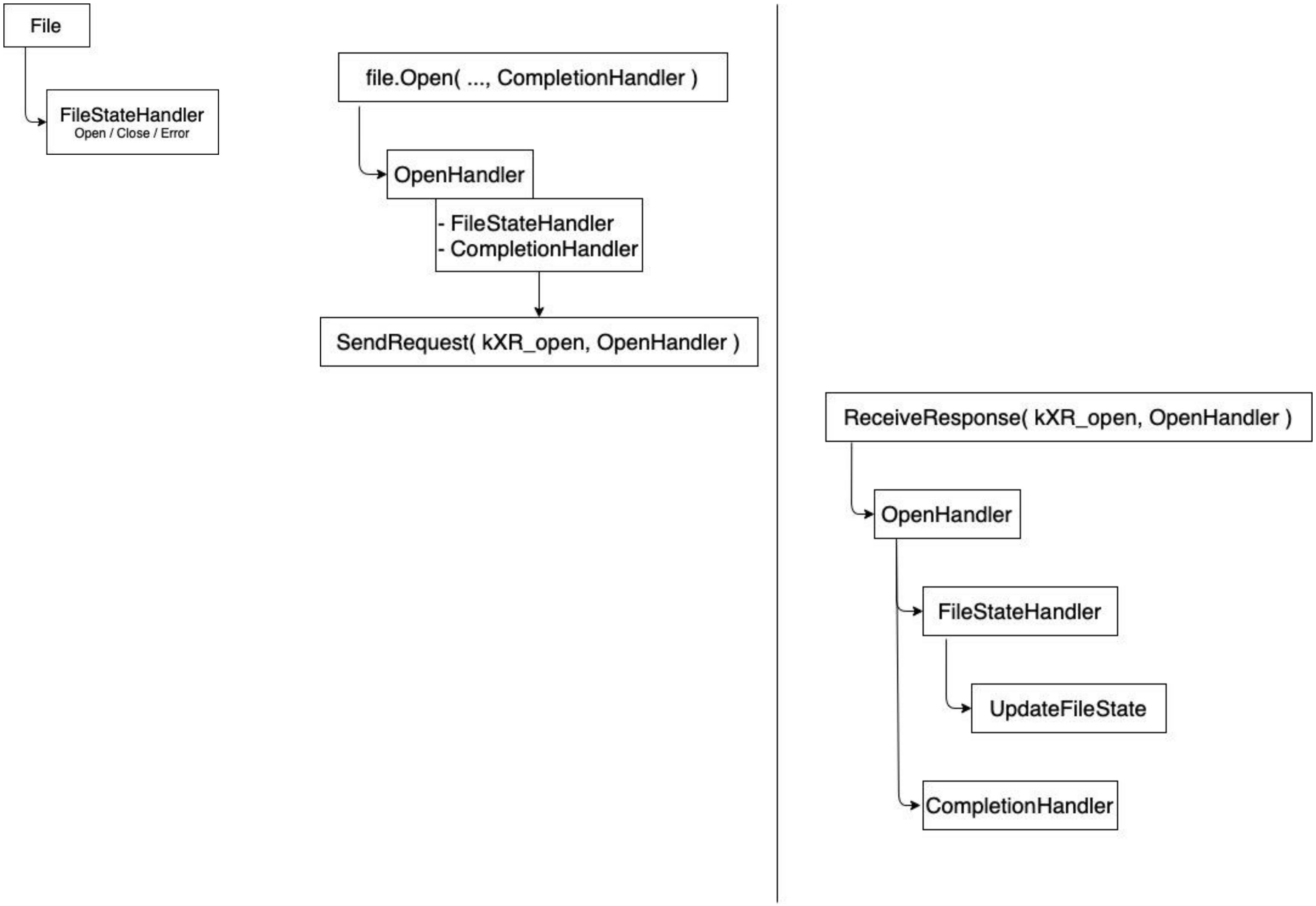
only ~15 lines of code,
no boilerplate code!

Limitations of XrdCl::File: #1

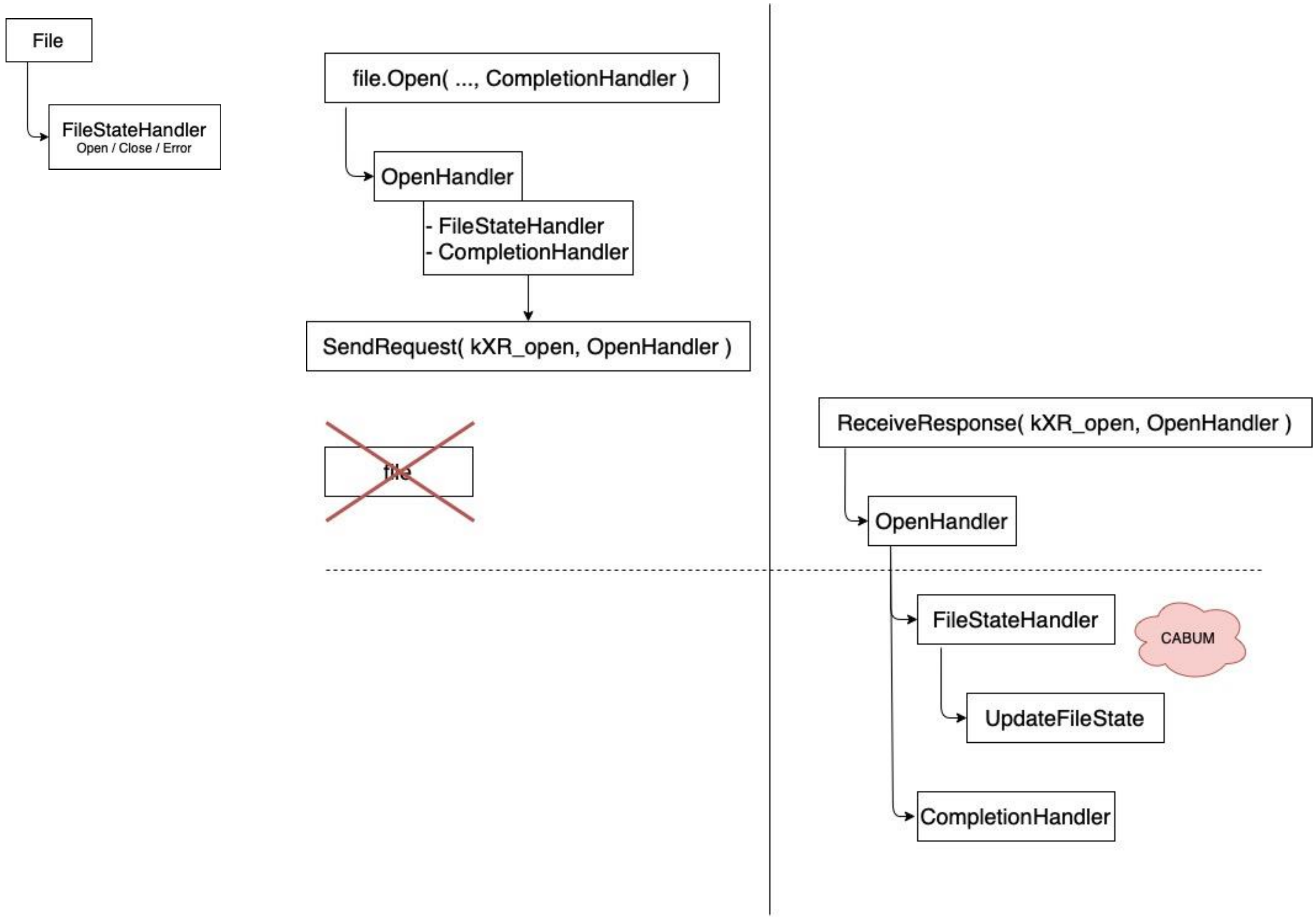
- In order to handle internal state the **XrdCl::File** uses an internal *FileStateHandler* object that is being called whenever a async operation completes

- As the *FileStateHandler* needs to exist at the moment a response arrives, hence **XrdCl::File** object **MUST NOT** be destroyed if there are any requests in-the-flight

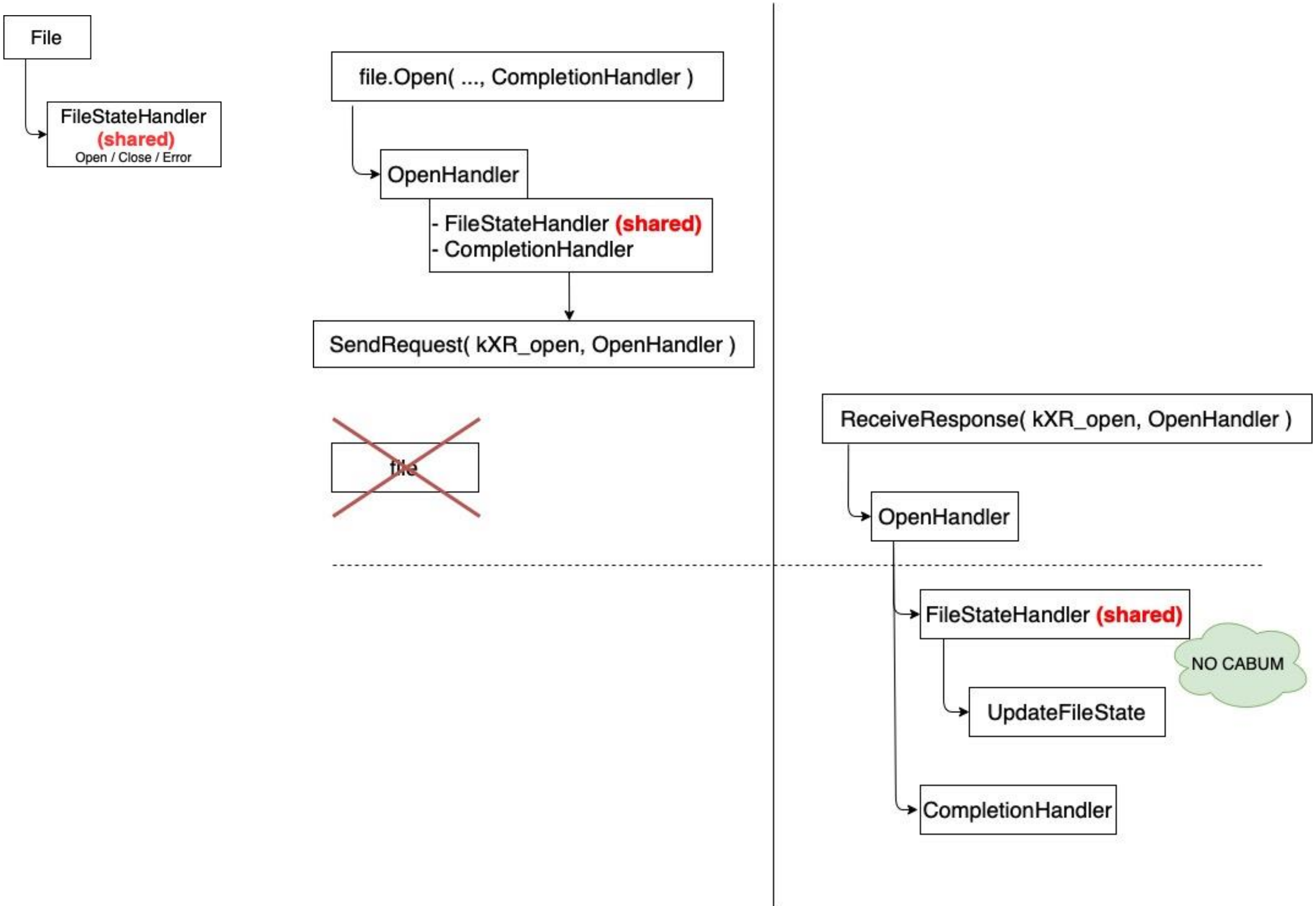
Limitations of XrdCl::File: #1



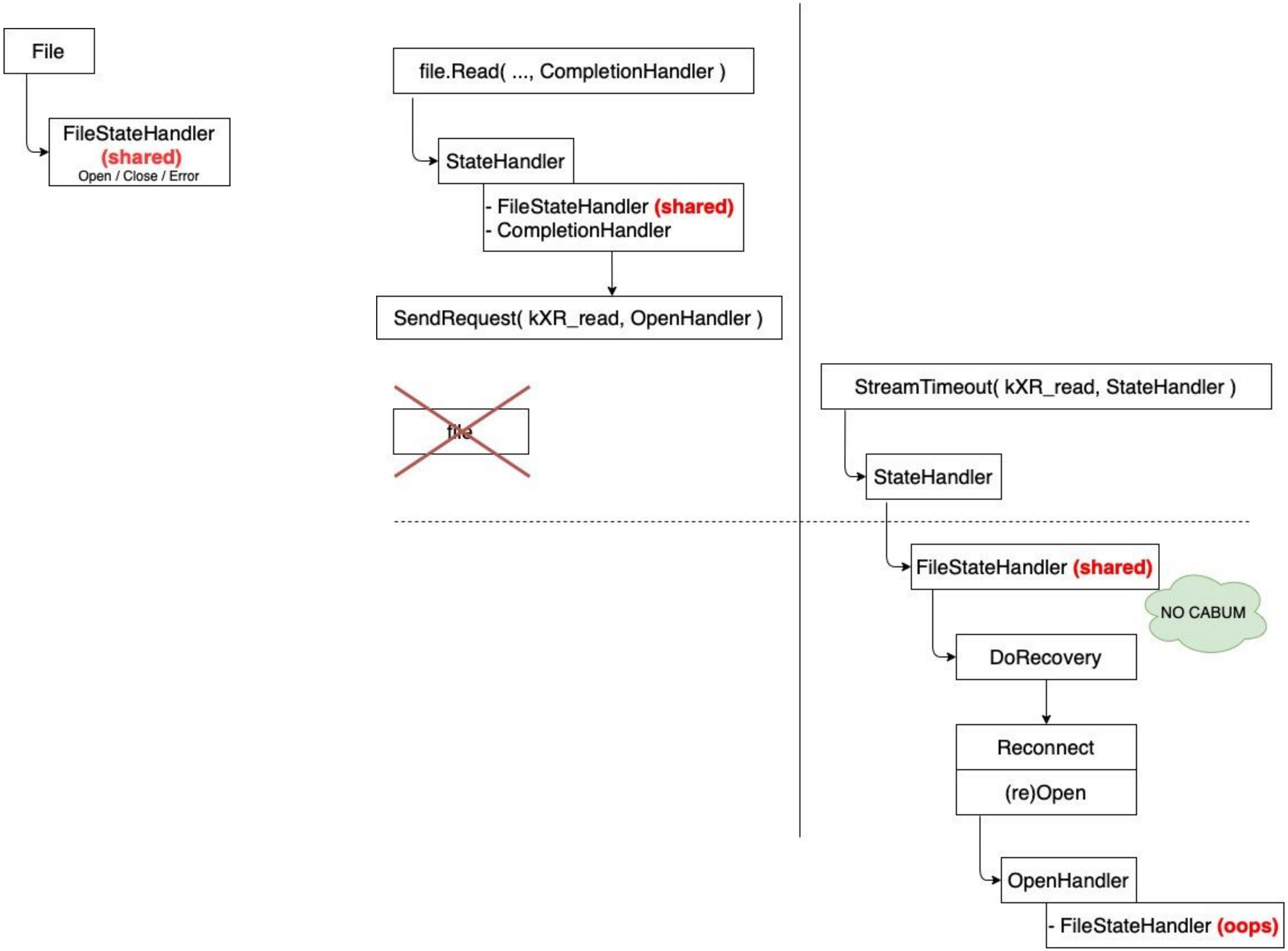
Limitations of XrdCl::File: #1



Limitations of XrdCl::File: #1



Limitations of XrdCl::File: #1



Limitations of XrdCl::File: #2, part 1

- All synchronous operations are implemented in terms of asynchronous operations
  - By providing a completion handler that syncs issuing the request with receiving the response using a semaphore

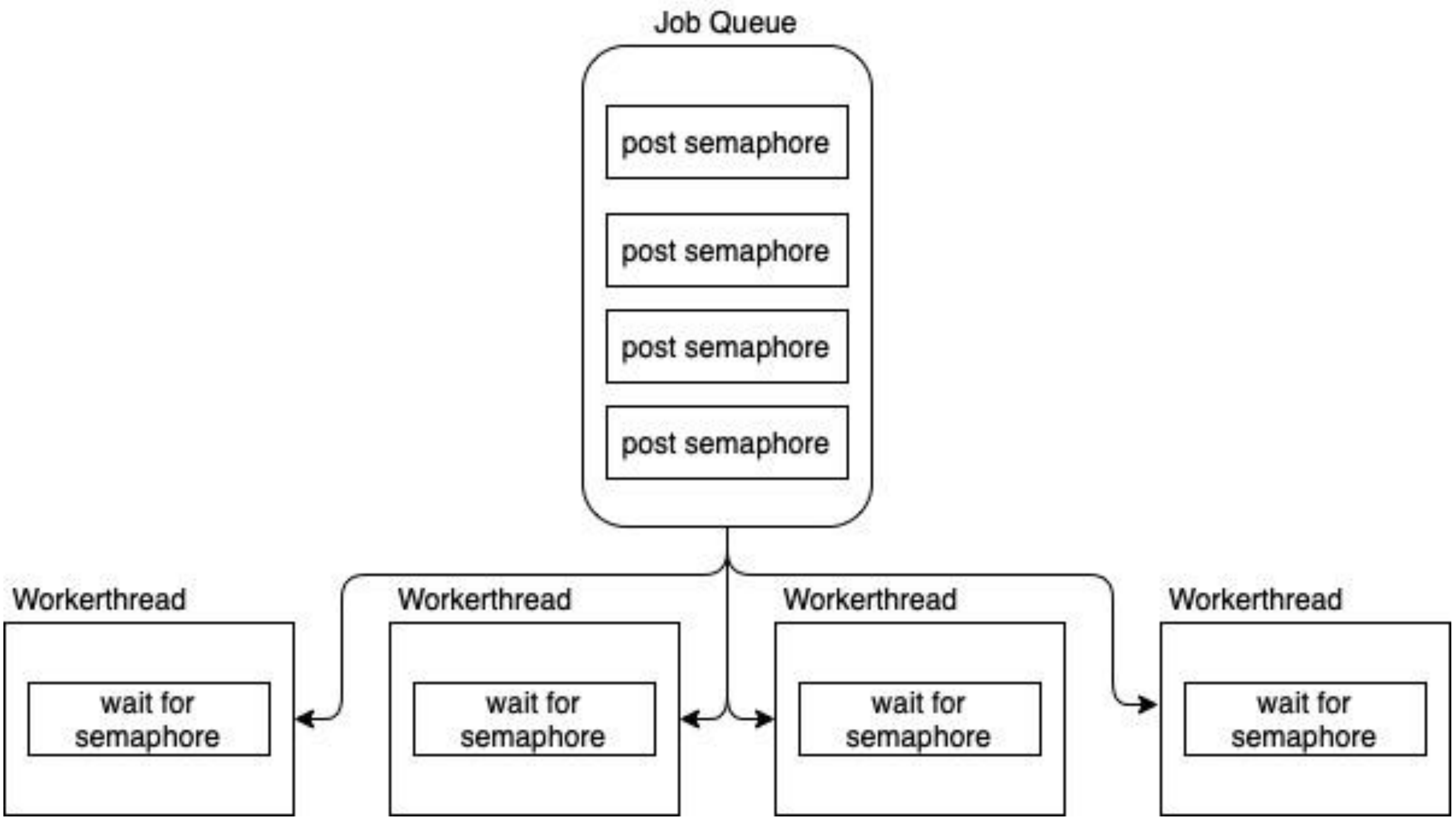
- All completion handlers are called in the (fixed size) thread-pool

Limitations of XrdCl::File: #2, part 2

- One **MUST NOT** mix synchronous operations with asynchronous ones
  - Consider following example:
    - We have a thread-pool of 4 threads
    - We issue 4 asynchronous *opens*
    - In the completion handlers we issue 4 synchronous *closes*
    - This will deadlock the whole thread-pool: each worker thread will wait on a semaphore that will be only posted when a worker thread is available

  - Use declarative API to chain operations!!!

Limitations of XrdCl::File: #2



Limitations of XrdCl::File: #3

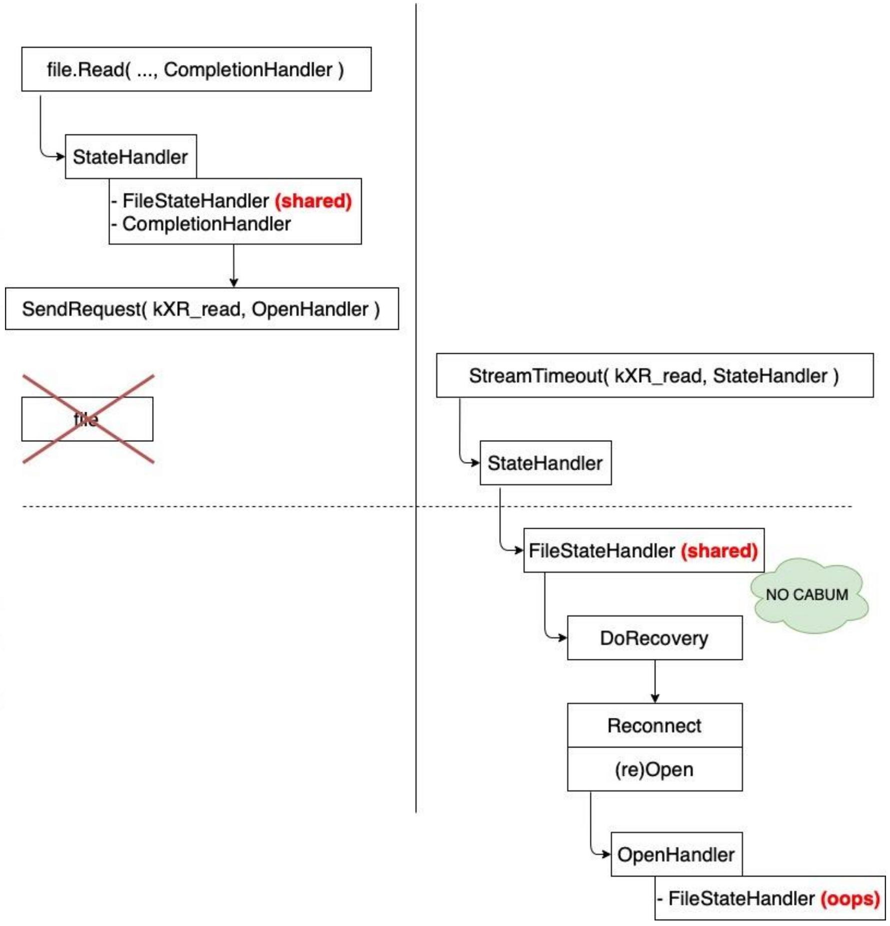
- The **XrdCl::File** destructor will issue a *Close* request
  - The *Close* has to be synchronous
    - As discussed in #1 the XrdCl::File object **MUST** exist when the response comes back
  - Hence we recommend to always do an async close before destroying the XrdCl::File object

```
auto file = std::make_shared<XrdCl::File>();

XrdCl::Pipeline p = XrdCl::Open( *file, "root://host//path", XrdCl::OpenFlags::Read )
                  | XrdCl::Close( *file ) >> // MUST do an asynchronous Close
                      [file]( auto st )
                      {
                        file.reset(); // MUST destroy the file only after it's closed
                      };
```

Can we do better?

- Can we work around limitation #1 and #3?
- Both are really error prone!!!
- If only the ***this*** pointer would be reference counted!!!



Let's be more Pythonic :-)

- Chage the implementation from:

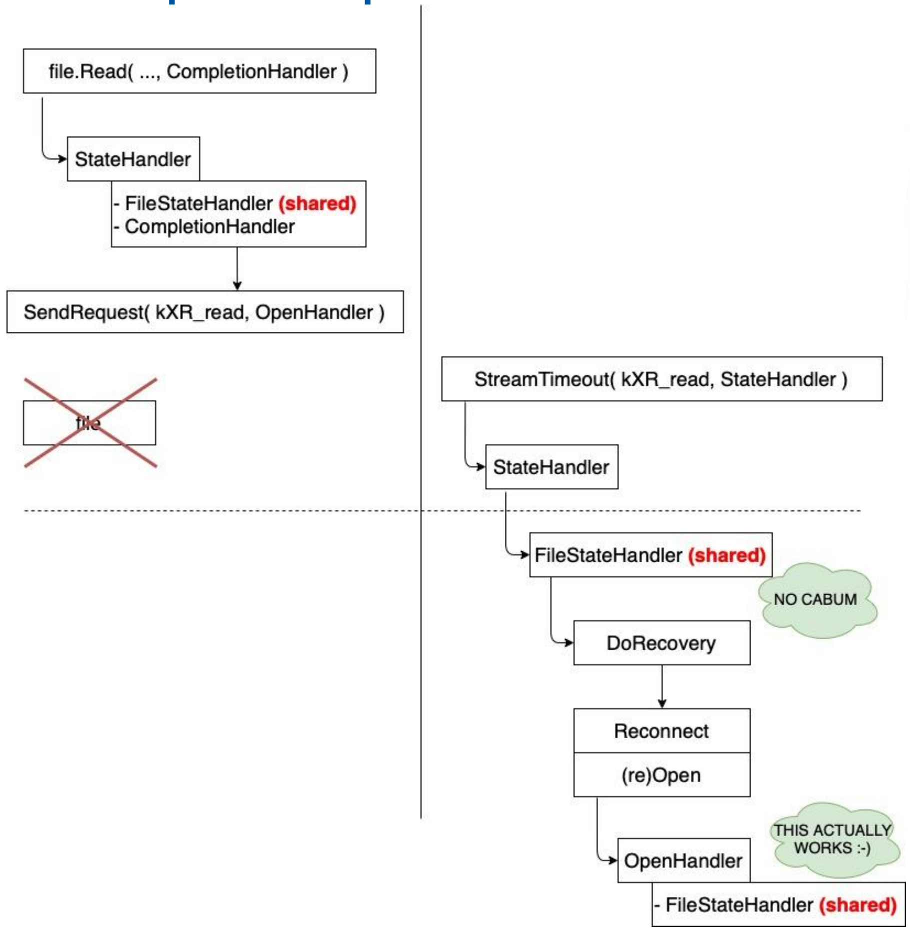
```
XRootDStatus Open( const std::string                   &url,
                   uint16_t                             flags,
                   uint16_t                             mode,
                   ResponseHandler                     *handler,
                   uint16_t                             timeout  = 0 );
```

to:

```
static XRootDStatus Open( std::shared_ptr<FileStateHandler> &self,
                          const std::string                 &url,
                          uint16_t                           flags,
                          uint16_t                           mode,
                          ResponseHandler                   *handler,
                          uint16_t                           timeout  = 0 );
```

Let's be more Pythonic :-)

- This required quite some refactoring but now this works:



Limitation #1 is lifted!

Let's be more Pythonic :-)

- With limitation #1 being removed we can replace the synchronous call to *Close* in **XrdCl::File** destructor with an asynchronous call and hence lift limitation #3

- To summarize, starting with **5.5.0**:

  - It is **OK** to issue an async request and then immediately destroy the **XrdCl::File** object (also true for the **XrdCl::FileSystem** object)

  - It is **OK** to destroy the **XrdCl::File** object in a completion handler without previously closing it

Record & replay: motivation

- **Allow to emulate real applications without running complex runtime environments**
  - Facilitate **benchmarking** and **debugging** of storage systems

- Based on two components:
  - **Recorder plugin**: records all client actions in a *CSV* file
    - One can **load the plugin into any 'black-box' application** that use XRootD client without modifying the source code
  - **Replay tool**: replay all client actions
    - Preserving original timing

Recorder plug-in

- Available in **xrootd-client-recorder** sub-package
  - In order to accommodate older **XRootD4 clients** we also provide a back-ported version as a separate package
  - Released in **5.5.0**

- Example configuration file:

```
url = *
lib = /usr/lib64/libXrdClRecorder-5.so
enable = true
output = /tmp/out.csv # optional
```

Recorder plug-in

- User's actions are stored using ***CSV* file format**
  - We do **support quoting** so it is safe to use comas in URL opaque info

- By default the file is stored at: **/tmp/xrdrecord.csv**
  - This can be overwritten either in the config file **using the *output* key**, or
  - Using an **environment variable:** XRD_RECORDERPATH

- Introduces only **minimal or no overhead**

Replay tool

- To replay the registered actions:

  xrdreplay /tmp/xrdrecord.csv

- Alternatively one can do the replay from *stdin* (e.g. if the *CSV* needs to be unzipped):

  cat /tmp/xrdrecord.csv | xrdreplay

- There are 4 operational modes:
  - **Print mode**: display runtime and IO statistics for given *CSV*
  - **Verify mode**: verify that the required input files exist
  - **Creation mode**: create required input data
  - **Playback** (default): replay given *CSV*

Replay tool: print mode

- To display statistics from recorded I/O pattern without replaying do:

```
xrdreplay -p recording.csv

# =============================================
# IO Summary (print mode)
# =============================================
# Sampled Runtime  : 5.485724 s
# Playback Speed   : 1.00
# IO Volume (R)    : 536.87 MB [ std:536.87 MB vec:0 B page:0 B ]
# IO Volume (W)    : 536.87 MB [ std:536.87 MB vec:0 B page:0 B ]
# IOPS      (R)    : 64 [ std:64 vec:0 page:0 ]
# IOPS      (W)    : 64 [ std:64 vec:0 page:0 ]
# Files     (R)    : 1
# Files     (W)    : 1
# Datasize  (R)    : 536.87 MB
# Datasize  (W)    : 536.87 MB
# ---------------------------------------------
# Quality Estimation
# ---------------------------------------------
# Synchronicity(R) : 4.55%
# Synchronicity(W) : 100.00%
```

- To further inspect details of the recording use long format (-l) and/or the summary option (-s)

Replay tool: verify mode

- To verify availability of all input files do:

```
xrdreplay -v recording.cvs

...
# ---------------------------------------------
# Verifying Dataset ...
# .............................................
# file: root://cmsserver//store/cms/higgs.root
# size: 536.87 MB [ 0 B out of 536.87 MB ]  ( 0.00% )
# ---> info: file exists and has sufficient size
```

- On success the shell return code is 0, if there was a missing, too small, or inaccessible file the shell return code is 251.

Replay tool: creation mode

- Creates the required input files
  - ***-c*** create files reassembling the original
  - ***-t*** create and truncate the file to the required size (will contain 0s)

- *--replace* option allows to modify the input and output path used by xrdreplay
  - Can be used multiple times to overwrite multiple URLs

```
xrdreplay --replace root://cmsserver//store/cms/:=root://mycluster//mypath/ --replace file:/data/:=file:/gpfs/data/ -v
```

Replay tool: playback mode

- Without print, verify or create ***xrdreplay*** will replay the recorded pattern
  - It will try to **preserve the original timings** (this might not be possible if responses are significantly slower)
  - The ***–x*** option allows to tune the replay speed
- After replaying the pattern a summary is given

```
xrdreplay recording.cvs

# =============================================
# IO Summary
# =============================================
# Total   Runtime  : 5.488581 s
# Sampled Runtime  : 5.485724 s
# Playback Speed   : 1.00
# IO Volume (R)    : 536.87 MB [ std:536.87 MB vec:0 B page:0 B ]
# IO Volume (W)    : 536.87 MB [ std:536.87 MB vec:0 B page:0 B ]
# IOPS      (R)    : 64 [ std:64 vec:0 page:0 ]
# IOPS      (W)    : 64 [ std:64 vec:0 page:0 ]
# Files     (R)    : 1
# Files     (W)    : 1
# Datasize  (R)    : 536.87 MB
# Datasize  (W)    : 536.87 MB
# IO BW     (R)    : 97.82 MB/s
# IO BW     (W)    : 97.82 MB/s
```

Summary

- Don't be afraid of **async APIs**
  - Declarative API makes it **much easier and readable**
  - The File object no longer needs to outlive the completion handlers

- Record / replay is **great for debugging and benchmarking** storage systems

- There is **lots of functionalities build into the *xrdcp*** tool
  - Be sure to know its capabilities before enhancing it with scripts