
CERN

Michal Simon

XRootD5: encryption and beyond

Outline

- Short introduction to XRootD5
- Secure root/xroot, why and how?
- Is there anything else?
- Declarative API
- Plans & Summary

XRootD5 in few words

- **Major release**, with the most important new feature being **encryption**
- Protocol and API level backwards compatibility, it is not ABI compatible – plugins will require recompilation
- Released in July, followed by 3 bugfix releases
  - Release in OSG repo and EPEL

Secure root/xroot protocol

- roots/xroots is the old good **root/xroot protocol plus TLS**
- Based on OpenSSL
  - Version 1.0.0 and above
  - Custom hostname verification added to cover the older versions
- Encrypted and unencrypted version of root/xroot protocol run on the same port (by default 1094)

Why do we need encryption?

- Allows for authorization token handling (e.g. SciToken)
  - Prerequisite for **replacing proxy delegation with access tokens** in WLCG
- Encrypt confidential data
  - Encryption ‘in transit’ especially for **CERNBox**
- Encrypt possibly destructive metadata operations (could replace in the future request signing)
- Improves data integrity and allows for further evolution of Third-Party-Copy

What triggers encryption?

- On the client side the **roots/xroots** protocol;
  - --**notlsok** options allows to proceed without encryption if the server is too old to support it
  - --**tlsmetalink** option allows to apply encryption to all URLs in a metalink file

- On the server side the **xrootd.tls** configuration directive, with few compatibility options:
  - by default it is **off**
  - enforce encryption only for clients that support it (**capable**)
  - do encryption only at client discretion (**none**)

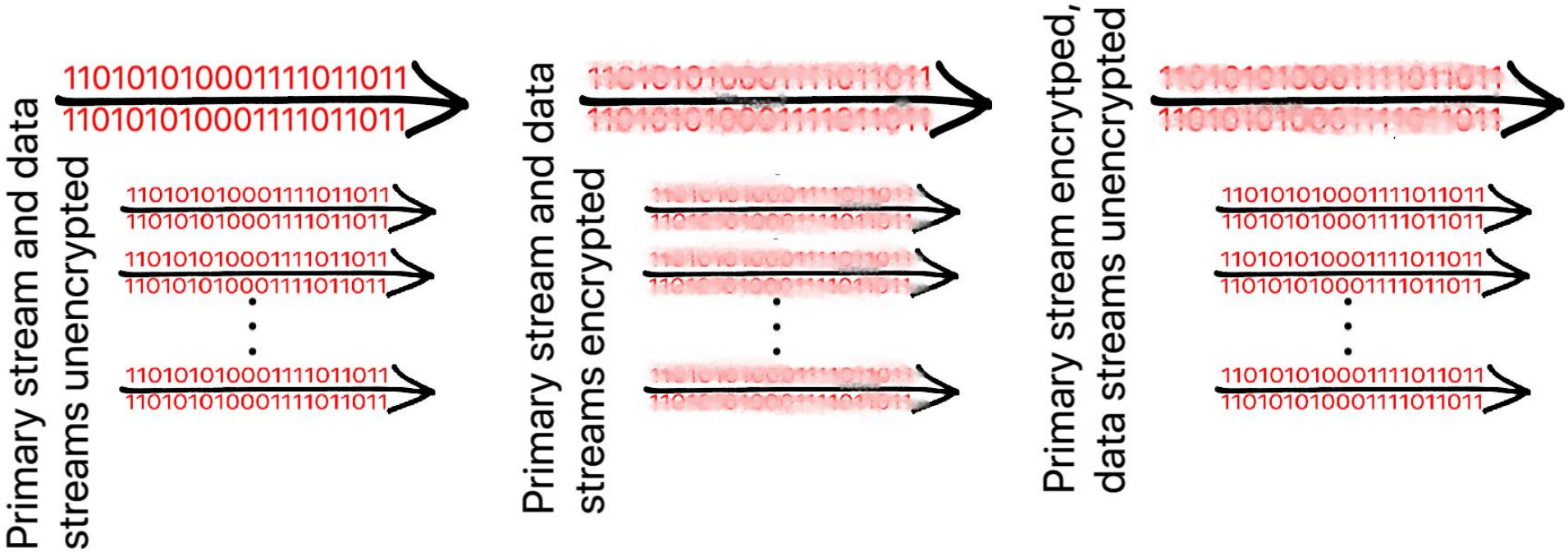
How flexible is it?

- It is pretty flexible ;-) – not everything needs encryption and (at the beginning) not everyone will support encryption

- One can configure the server to encrypt:
  - only the **third-party-copy** orchestration
  - **control channel** after login (handy for GSI auth)
  - control channel before login
  - **data streams**
  - everything

- On the client side:
  - **--tlsnodata** allows to apply roots/xroots only to the control stream

How flexible is it?



What lies beneath the flexibility?

- Handshake negotiation
  - All connections are **initially non-encrypted**
  - The connection is being **upgraded to TLS on client or server request**

- If only control channel should be encrypted we open a second (or multiple) physical connection for the raw data

- **Encrypted and unencrypted traffic uses the same port** number (not like http vs https) to ease operators lives

Is roots/xroots widely available?

- **GFAL2 has been ported to XRootD5** (rebuilt with XRootD5 in EPEL)
- **EOS has been ported to XRootD5** (successful encrypted transfer executed in PPS)
- **DPM has been ported to XRootD5** (rebuilt with XRootD5 in EPEL, passed all site's and dev's tests)
- **dCache** devel team (with our help) implemented roots/xroots support (in Java!!!)

Certificates, certificates, …

- XRootD server needs a host certificate in order to enable encryption
  - configurable with **xrd.tls** directive

- If roots/xroots is being used client will **enforce host verification**
  - the hostname must match the one in the host certificate (or one of the SAN extensions)

Certificates, certificates, …

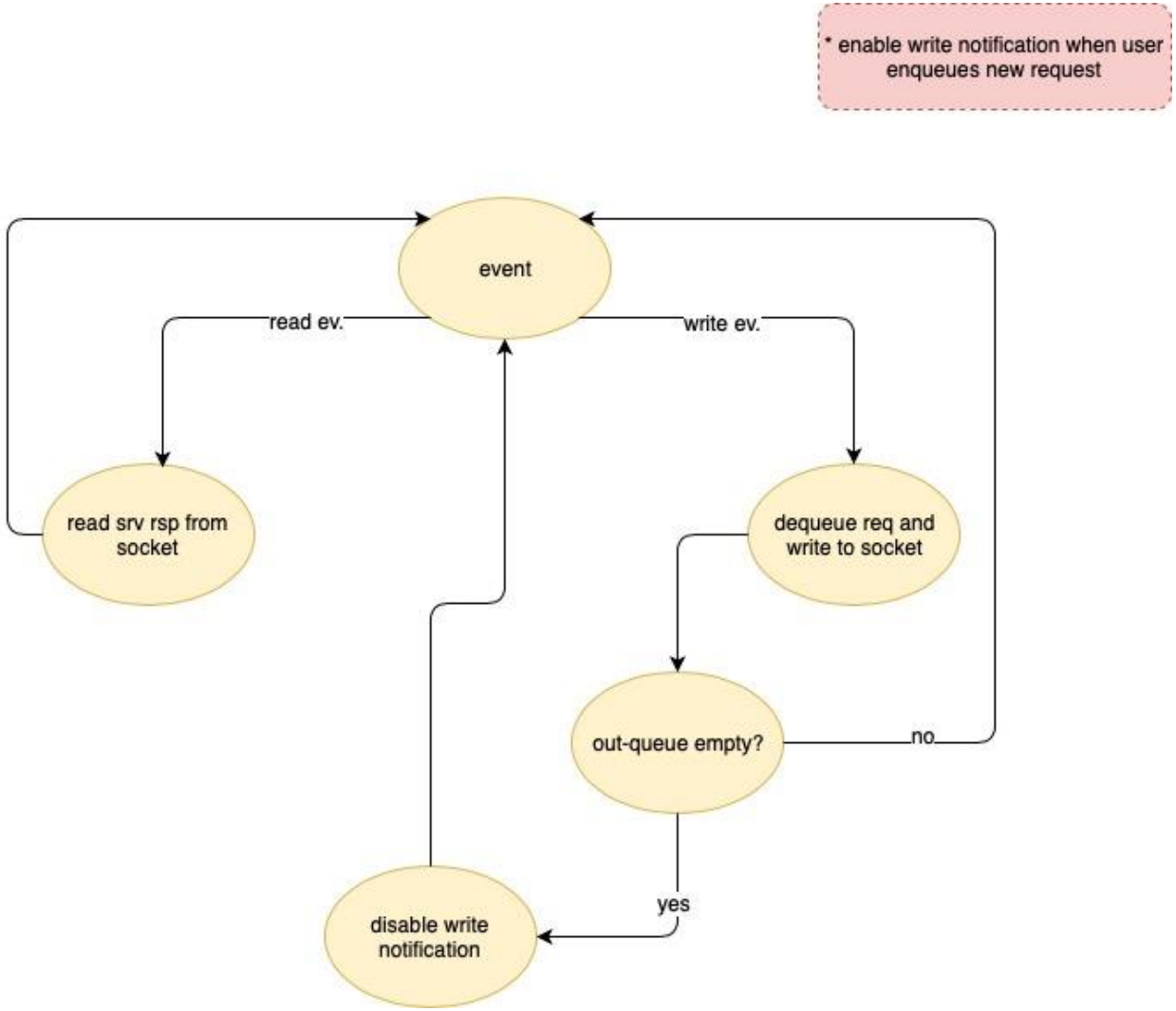
- The client does not need to have a certificate
  - the user **may use his proxy certificate** in order to establish a TLS connection
  - server can be configured to enforce client certificate verification with: **xrd.tlsca**

- Allowing the client to establish the TLS connection based on user X509 proxy certificate opens door to a new **less complex implementation of gsi authentication** in the future

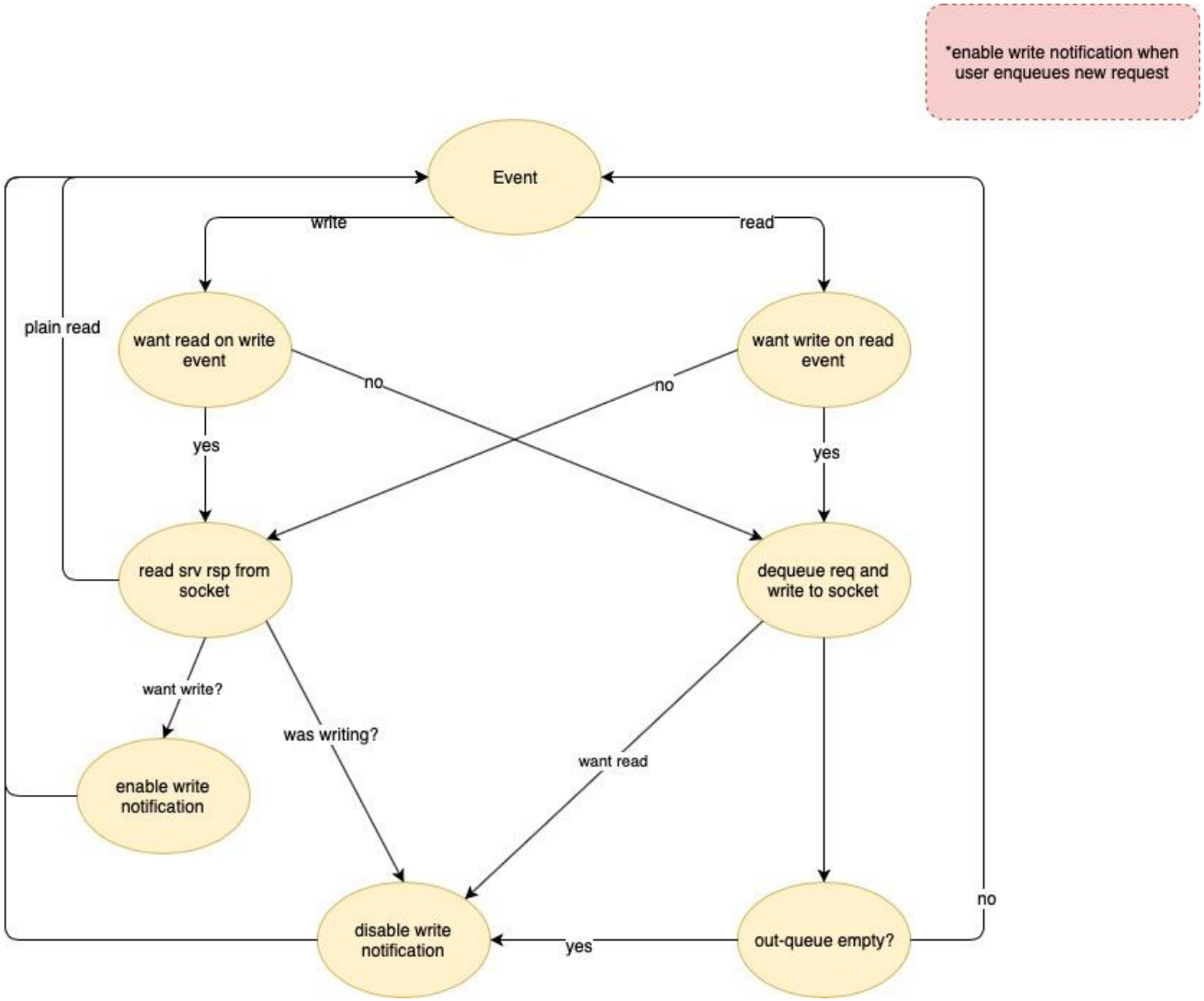
Implementation

- roots/xroots implementations is based on OpenSSL
  - for better performance, **asynchronous APIs and socket BIOs** were used
- **All TLS actions are logged** (e.g. when connection is upgraded to TLS, what version of TLS is being used)
- We are aiming at **isolating OpenSSL in the XrdTls** component
  - should facilitate migration from OpenSSL in the future (e.g. to NSS)

Implementation: event loop (no TLS)



Implementation: event loop (with TLS support)



Anything else?

SecEntity re-mastered:

- X509 capabilities, key-value attributes
- **Credential forwarding**, Multi-VO credentials
- Easily **extensible without breaking API**

Universal (both **root/xroot and http**) **VOMS attribute extractor** plugin

- Gerri's xrdvoms plugin was the starting point
- Now shipped as a sub-package of XRootD in **EPEL**
- Obsoletes several packages (vomsxrd, xrootd-voms-plugin and xrdhttpvoms)

Anything else?

General purpose new features

- **Extended file attributes**
- Extended stat (sets stages for proper **uid/gid tracking**)
- Hardware assisted **CRC32C**
- gstream (monitoring stream optimized to deliver periodic medium-level info)
- Server side **plug-in stacking** with `++` directive
  - User plugin gets a pointer to the level-up plugin so it can call it’s implementation

Client Declarative API

Motivation

- Use case: erasure coding plug-in for EOS ALICEO2
  - AliceO2: 2000 data sources exporting 2GB file (time frame) every 40 seconds
  - Aggregate throughput of 100GB/s
  - 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)

Client Declarative API

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

```
1
2  using namespace XrdCl;
3
4  /*
5   * Write to a single chunk
6   */
7  void ECWrite(uint64_t           offset,
8               uint32_t           size,
9               const void        *buff,
10              ResponseHandler *userHandler)
11 {
12   // translate arguments to chunk specific parameters
13   // ...
14   File *file=new File();
15   OpenHandler *handler=
16     new OpenHandler(file,userHandler,/*long list of arguments*/);
17   // although we do a write in here we only see an open call,
18   // all the logic is hidden in the callback and the workflow
19   // is unclear
20   file->Open(url,flags,handler);
21 }
22
```

Client Declarative API

… 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
  // ...
}
```

Client Declarative API

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!

Client Declarative API

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

Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure coded block
5  void ECWrite(uint64_t          offset,
6               uint32_t          size,
7               const void       *buffer,
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // calculate offset, size and buffer for each stripe/chunk
14     // ...
15     File *file=new File();
16     Pipeline p=Open(file,url,flags)
17              |Parallel(Write(file,choff,chsize,chbuff),
18                       SetXAttr(file,"xrdec.cksum",checksum))
19              |Close(file)>>[file](XRootDStatus&){delete file;}
20   }
21   // Execute the workflow!
22   Async(Parallel(wrts) >>
23         [userHandler](XRootDStatus& st)
24         {userHandler->HandleResponse(new XRootDStatus(st),0);});
25 }
26
```

Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure coded block
5  void ECWrite(uint64_t          offset,
6               uint32_t          size,
7               const void       *buffer,
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // calculate offset, size and buffer for each stripe/chunk
14     // ...
15     File *file=new File();
16     Pipeline p=Open(file,url,flags)
17               |Parallel(Write(file,choff,chsize,chbuff),
18                        SetXAttr(file,"xrdec.cksum",checksum))
19               |Close(file)>>[file](XRootDStatus&){delete file;}
20   }
21   // Execute the
22   Async(Parallel
23         [userHa                                    t)
24         {userHa                              new XRootDStatus(st),0);});
25 }
26
```

Compose operations with | operator!

Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure coded block
5  void ECWrite(uint64_t          offset,
6               uint32_t          size,
7               const void       *buffer,
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // calculate offset, size and buffer for each stripe/chunk
14     // ...
15     File *file=new File();
16     Pipeline p=Open(file, url, flags)
17               |Parallel(Write(file,choff,chsize,chbuff),
18                         SetXAttr(file,"xrdec.cksum",checksum))
19               |Close(file)>>[file](XRootDStatus&){delete file;}
20   }
21   // Execute the workflow
22   Async(Parallel(wrts) >>
23         [userHandler](X
24         {userHandler->H                tDStatus(st),0);});
25 }
26
```

Parallel execution of operations!

Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure coded block
5  void ECWrite(uint64_t          offset,
6               uint32_t          size,
7               const void       *buffer,
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // calculate offset, re     uffer for each stripe/chunk
14     // ...
15     File *file=new F
16     Pipeline p=Oper
17                | Par                    chsize,chbuff),
18                                        xrdec.cksum",checksum))
19                | Close(file)>>[file](XRootDStatus&){delete file;}
20   }
21   // Execute the workflow!
22   Async( Parallel(wrts) >>
23         [userHandler](XRootDStatus& st)
24         {userHandler->HandleResponse(new XRootDStatus(st),0);});
25 }
26
```

Parallel execution of a container of operations

Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure coded block
5  void ECWrite(uint64_t          offset,
6               uint32_t          size,
7               const void       *buffer,
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // ... offset, size and buffer for each stripe/chunk
14
15     ...e();
16     ...e,url,flags)
17     ...(Write(file,choff,chsize,chbuff),
18                 SetXAttr(file,"xrdec.cksum",checksum))
19               |Close(file)>>[file](XRootDStatus&){delete file;}
20   }
21   // Execute the workflow!
22   Async(Parallel(wrts) >>
23         [userHandler](XRootDStatus& st)
24         {userHandler->HandleResponse(new XRootDStatus(st),0);});
25 }
26
```

Specify async callback with >> operator

Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure coded block
5  void ECWrite(uint64_t          offset,
6               uint32_t          size,
7               const void       *buffer,
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // ... offset, size and buffer for each stripe/chunk
14
15     ...();
16     ..., url, flags)
17     ...(Write(file, choff, chsize, chbuff),
18                 SetXAttr(file, "xrdec.cksum", checksum))
19               | Close(file) >> [file](XRootDStatus&){ delete file; }
20   }
21   // Execute the workflow!
22   Async( Parallel(wrts) >>
23         [userHandler](XRootDStatus& st)
24         {userHandler->HandleResponse(new XRootDStatus(st),0);});
25 }
26
```

Use lambdas (or std::future) as callbacks

Client Declarative API

Using declarative API:

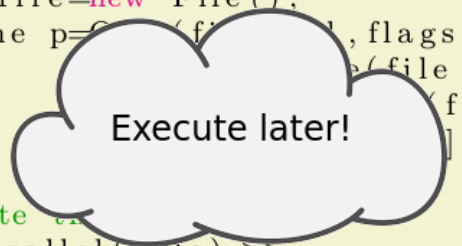
```
1
2  using namespace XrdCl;
3
4  // Write erasure cod
5  void ECWrite(uint64_
6               uint3
7               const
8               Respon                    ier)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // calculate offset, size and buffer for each stripe/chunk
14     // ...
15     File *file=new File();
16     Pipeline p=Open(file,url,flags)
17               |Parallel(Write(file,choff,chsize,chbuff),
18                         SetXAttr(file,"xrdec.cksum",checksum))
19               |Close(file)>>[file](XRootDStatus&){delete file;}
20   }
21   // Execute the workflow!
22   Async(Parallel(wrts) >>
23         [userHandler](XRootDStatus& st)
24         {userHandler->HandleResponse(new XRootDStatus(st),0);});
25 }
26
```

First prepare the workflow

Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure coded block
5  void ECWrite(uint64_t          offset,
6               uint32_t          size,
7               const void       *buffer,
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // calculate offset, size and buffer for each stripe/chunk
14     // ...
15     File *file=new File();
16     Pipeline p=C...(fi...l, flags)
17                                 ...e(file, choff, chsize, chbuff),
18                                 ...(file, "xrdec.cksum", checksum))
19                                 ...](XRootDStatus&){delete file;}
20   }
21   // Execute t...
22   Async( Parallel(wrts) >>
23         [userHandler](XRootDStatus& st)
24         {userHandler->HandleResponse(new XRootDStatus(st),0);});
25 }
26
```



Client Declarative API

Using declarative API:

```
1
2  using namespace XrdCl;
3
4  // Write erasure cod
5  void ECWrite(uint6
6               uint3
7               const
8               ResponseHandler *userHandler)
9  {
10   std::vector<Pipeline> wrts; wrts.reserve(nbchunks);
11   for(size_t i=0;i<nbchunks;++i)
12   {
13     // calculate offset, size and buffer for each stripe/chunk
14     // ...
15     File *file=new File();
16     Pipeline p=Open(file,url,flags)
17               |Parallel(Write(file,choff,chsize,chbuff),
18                         SetXAttr(file,"xrdec.cksum",checksum))
19               |Close(file)>>[file](XRootDStatus&){delete file;}
20   }
21   // Execute the workflow!
22   Async(Parallel(wrts) >>
23         [userHandler](XRootDStatus& st)
24         {userHandler->HandleResponse(new XRootDStatus(st),0);});
25 }
26
```

only ~15 lines of code,
no boilerplate code!

Client Declarative API

Another example: parsing ZIP using declarative API

- Open remote ZIP archive and read the Central Directory
- Once we have the archive size we can make an educated guess on the offset of the Central Directory
- However it might be that there is a comment at the end of the archive which will invalided our guess
  - In this case we need to adjust the offset we are reading at and reissue the read request

Client Declarative API

Parsing ZIP using declarative API:

```
1
2  File archive;
3  Fwd<uint32_t> rdsize = CD::size; // assume no comment
4  Fwd<uint64_t> rdoff;
5  Fwd<void*>    rdbuff = new char[CD::size];
6  Pipeline open = Open(archive, url, flags) >>
7                  [=](XRootDStatus &status, StatInfo &info)
8                  {
9                    // handle status ...
10                   uint64_t archsize = info.GetSize();
11                   if( archsize == 0 ) Pipeline::Stop();
12                   // calculate offset for 'Read'
13                   rdoff  = archsize - CD::size;
14                 }
15               | Read(archive, rdoff, rdsize, rdbuff)
16                 [=](XRootDStatus &st, ChunkInfo &ch)
17                 {
18                   // handle status ...
19                   ParseCD(ch.buffer, ch.length);
20                   if(need_more)
21                   {
22                     // adjust rdsize, rdoff & rdbuff
23                     Piepeline::Repeat();
24                   }
25                   // ...
26                 };
27
```

Client Declarative API

Parsing ZIP using declarative API:

```
File archive;
Fwd<uint32_t> rdsize = CD::size; // assume no comment
Fwd<uint64_t> rdoff;
Fwd<void*>    rdbuff = new char[CD::size];
Pipeline open = Open(archive, url, flags) >>
                  [=](XRootDStatus &status, StatInfo &info)
                  {
                    // handle status ...
                    uint64_t archsize = info.GetSize();
                    if( archsize == 0 ) Pipeline::Stop();
                    // calculate offset for 'Read'
                    rdoff  = archsize - CD::size;
                  }
                | Read(archive, rdoff, rdsize, rdbuff)
                  [=](XRootDStatus &st, ChunkInfo &ch)
                  {
                    // handle status ...
                    ParseCD(ch.buffer, ch.length);
                    if(need_more)
                    {
                      // adjust rdsize, rdoff & rdbuff
                      Piepeline::Repeat();
                    }
                    // ...
                  };
```

Client Declarative API

Parsing ZIP using declarative API:

```
1
2  File archive;
3  Fwd<uint32_t> rdsize = CD::size; // assume no comment
4  Fwd<uint64_t> rdoff;
5  Fwd<void*>    rdbuff = new char[CD::size];
6  Pipeline open = Open(archive, url, flags) >>
7                  [=](XRootDStatus &status, StatInfo &info)
8                  {
9                    // handle status ...
10                   uint64_t archsize = info.GetSize();
11                   if( archsize == 0 ) Pipeline::Stop();
12                   // calculate offset for 'Read'
13                   rdoff  = archsize - CD::size;
14                 }
15               | Read(archive, rdoff, rdsize, rdbuff)
16                 [=](XRootDStatus &st, ChunkInfo &ch)
17                 {
18                   // handle status ...
19                   ParseCD(ch.buffer, ch.length);
20                   if(need_more)
21                   {
22                     // adjust rdsize, rdoff & rdbuff
23                     Piepeline::Repeat();
24                   }
25                   // ...
26                 };
27
```

set args in handler

Client Declarative API

Parsing ZIP using declarative API:

```
1
2   File archive;
3   Fwd<uint32_t> rdsize = CD::size;
4   Fwd<uint64_t> rdoff;
5   Fwd<void*>    rdbuff = new char[CD::size];
6   Pipeline open = Open(archive, url, flags) >>
7                   [=](XRootDStatus &status, StatInfo &info)
8                   {
9                     // handle status ...
10                    uint64_t archsize = info.GetSize();
11                    if( archsize == 0 ) Pipeline::Stop();
12                    // calculate offset for 'Read'
13                    rdoff  = archsize - CD::size;
14                  }
15                | Read(archive, rdoff, rdsize, rdbuff)
16                  [=](XRootDStatus &st, ChunkInfo &ch)
17                  {
18                    // handle status ...
19                    ParseCD(ch.buffer, ch.length);
20                    if(need_more)
21                    {
22                      // adjust rdsize, rdoff & rdbuff
23                      Piepeline::Repeat();
24                    }
25                    // ...
26                  };
27
```

On the horizon (5.1.0)

Ensure data integrity in XCache; significantly reduce transfer failures due to checksum errors

- **Paged read**: read request with **CRC32C** (hardware assisted) **per 4KB block**

New features for EOS

- **Write recovery at MGM** (allows to **recover 99% of I/O errors** for *xrdcp* transfers to EOS)
- **Collapse redirect from passive to active** MGM in xrootd client
  - Facilitate FUSE interaction with passive-active MGM deployment
- **Simplify buffer management & avoid copying data between kernel and user space**
  - Using **splice/vmsplice** syscalls
  - Speed up data transfer: for **slow medium ~3-5%**, for **fast (like ramdisk) ~40%**; reduces CPU usage **by a factor of 3-4**

2021 plans

- Follow up and support **XRootD5 deployment**; **backporting critical bugfixes** to XRootD4
- Pushing **XRootD to Debian** distribution (big thanks to Mattias!)
- High priority new developments
  - **Finalize client EC plugin for Alice O2** (Hook it up to EOS)
  - **ZIP append** (initial work done by a summer student, needs checkpoint support on server side)
  - **Paged Write** (Further boost XCache data integrity)
- Other new developments
  - **uid/gid** tracking; connect **control and data streams on different interface**; recursive delete (driven by webdav semantics)
  - Get/put file (new TPC); channel level plug-ins; RDMA support; Extending testing infrastructure (mock event-loop)

Summary

- We have a working and fully functional **secure roots/xroots protocol**
- Many backwards compatibility ‘features’ to **facilitate forward migration path**
- Plenty new features and enhancements facilitating **development against XRootD** framework

Questions?