

The CERN Tape Archive: CTA-EOS Interface

Eric Cano

Michael Davis

Steven Murray

September 5, 2017

Contents

Contents	1
1 Introduction	2
2 CTA Operations	3
2.1 Archive file (CLOSEW)	3
2.2 Update	3
2.3 Explicit retrieve (PREPARE)	3
2.4 Implicit retrieve (open for read)	5
2.5 User-triggered disk copy removal	5
2.6 Garbage collection of disk copies	5
2.7 Complete deletion of files	5
2.8 File (re-)injection	6
2.9 Slow reconciliation	6
2.10 Operations summary	6
3 EOS-CTA Protocol	7
3.1 Data serialization	7
3.2 CTA front end extension	9
3.3 Move to Xrootd's SSIv2	10
3.4 EOS support of tape notions	10
4 Constraints	11
4.1 Performance requirements	11
4.2 Operational constraints	11
4.3 Synchronous and Asynchronous Calls	11
4.4 Data Integrity	11
4.4.1 After-the-fact check on archive from EOS	11
4.5 Security	11
A Questions and Issues	13
A.1 API	13
A.2 Communication Layer	13
A.3 Return value	13
A.4 <code>xattr</code>	13
A.5 CTA Failure	14
A.6 File life cycle	14
A.7 Handling of immutability	14
A.8 Slow reconciliation interface	14
A.9 Restoring Deleted Files	14
A.10 Storage Classes	14
A.11 Request Queue	14
A.12 Catalogue	14

B EOS-CTA Reconciliation Strategy 15

 B.1 Reconciling EOS file info and CTA disk file info 15

 B.2 Reconciling EOS deletes which haven't been propagated to CTA 15

C EOS-CTA Authorization Rules 16

Chapter 1

Introduction

This document summarizes the full control chain between the User, EOS and CTA. It is intended to be used for CTA-EOS interface design. The file lifecycle includes:

Archive File:

- File create: write until initial close (CLOSEW)
- File update: denied for files with copy on tape. This can be achieved by administrators by adding an immutable flag (ACL) to the directories configured to go to tape in EOS, or as rule in EOS.

Retrieve File:

- Explicit retrieve (PREPARE)
- Implicit retrieve (allowed or not, this choice should probably be left to the operator)

Delete File:

- User-triggered disk copy removal (allowed or not, optional)
- Garbage collection of disk copies
- Complete deletion of files

Update Metadata:

- Update metadata on EOS side, propagate to CTA, rate limited:
 - Generic metadata update (just involving a catalog entry update)
 - Storage class change, involving a migration to different tape pools (most probably postponed to repack)
- Inject metadata of missing files from CTA to EOS:
 - Injection of files from CASTOR during initial data migration
 - Re-injection of files from CTA to EOS during disaster recovery

In addition, in order to make sure no changes were lost, implicit operations are needed:

Reconciliation:

- Fast reconciliation: in-flight archive requests for sure, maybe retrieve requests as well
- Full or slow reconciliation: complete name space scan
- Reconcile Storage Classes: Synchronize the list of valid tape storage classes between EOS and CTA

Chapter 2 describes the use cases in more detail. Chapter 3 defines the protocol for the EOS-CTA API. Chapters 4 describes the performance and operational constraints on the system. Appendices A, B and C detail issues which need to be agreed on or resolved.

Chapter 2

CTA Operations

2.1 Archive file (CLOSEW)

Figure 2.1 shows the sequence of a client writing a file to EOS, with the optional check of the storage class on open and synchronous archive request queuing on close. This second option allows to signal problems to the user, at least on close and possibly as well on open. Note that:

- disk copies cannot be deleted before they are archived on tape (pinning)—the full file could still be deleted (potentially leading to issues to be handled in the tape archive session).
- the files with an archive on tape should be immutable in EOS (raw data use case), or a delayed archive mechanism should be devised for mutable files (CERNBox archive use case).
- synchronous calls allow reporting of CTA failures to the original client (especially in the case of custodial data/raw data case). The request will be sent to CTA as a notification message (see 3.1).
- reporting metadata in “tape replica” (checksum and size) in addition to archive completion will allow EOS to detect discrepancies (like happened when requests got mixed up in initial tests).
- the workflow will both trigger the synchronous archive queuing and post a second delayed workflow job that will check and re-issue the request if needed (in case the request gets lost in CTA). This event-driven reconciliation acts as a fast reconciliation. The criteria to check the file status will be the EOS side status (see below) which CTA reports asynchronously to EOS (see 3.1).

EOS will need to represent and handle part the tape status of the files. This includes the fact that the file should be on tape, the name of the CTA storage class, and the mutually exclusive statuses indicated by CTA: not on tape, partially on tape, fully on tape. The report from CTA will use the “tape replica” message (see 3.1).

2.2 Update

EOS should filter and forbid updates to files located on tape. Otherwise, a policy should be devised to migrate at a reasonable rate (once per day?), to allow a backup-type behavior (CERNBox files).

2.3 Explicit retrieve (PREPARE)

Explicit retrieve will face several use cases. The way we deal with them should be decided. Synchronous and asynchronous (through WFE) behaviors make little difference, but allow (catastrophic) problems to be reported to the user.

- The file is not on disk, no request is queued yet and so we should queue the retrieve request.

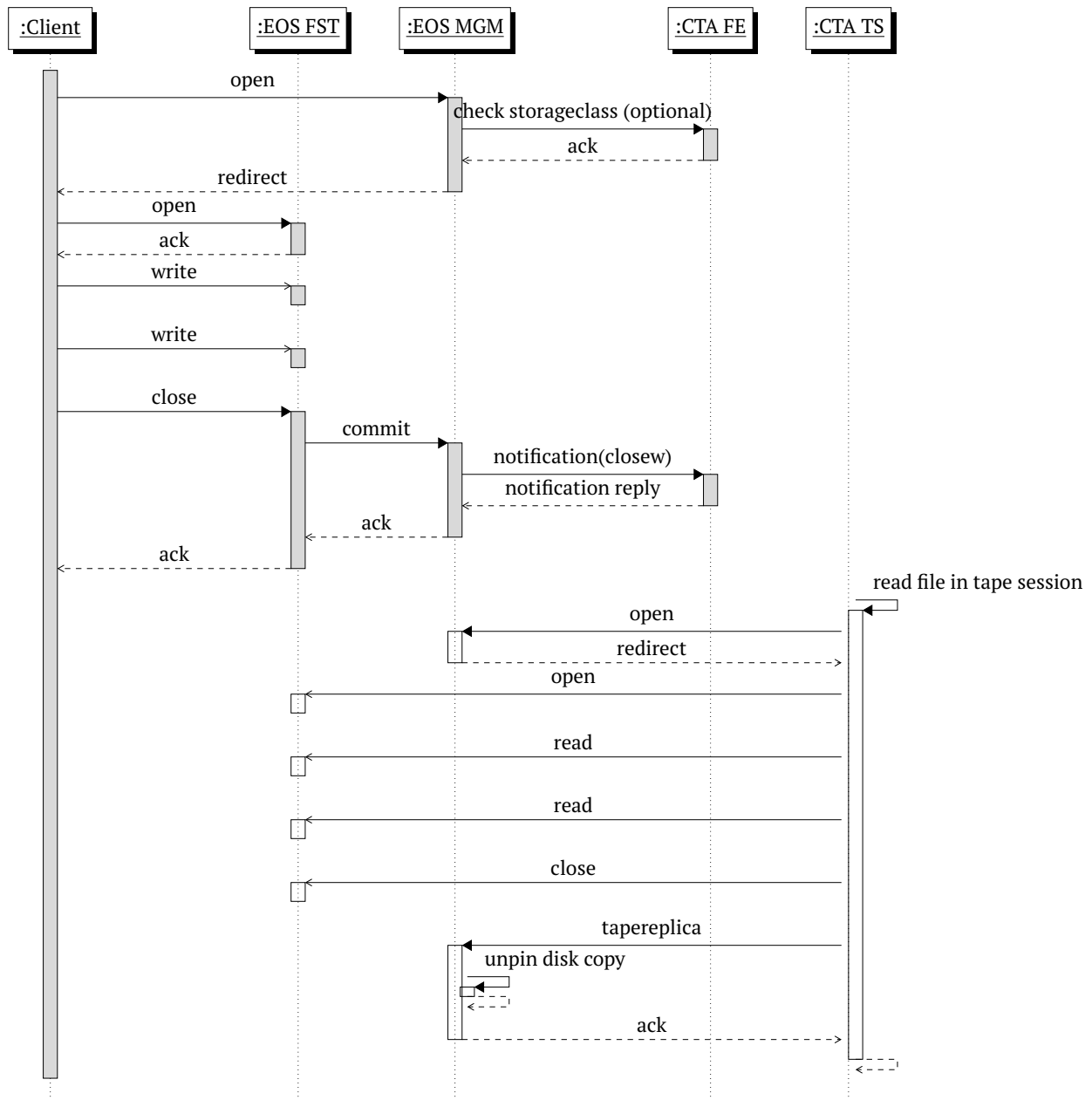


Figure 2.1: File write and archive queuing (synchronous)

- The file is not on disk, but a retrieve request is already queued in CTA. EOS could already know about this request (stateful case) or not. CTA should handle the requeuing in all cases.
- The file is already on disk and EOS should return immediately.

Figure 2.2 describes the enqueueing and subsequent retrieve of a file not present on disk (synchronous scenario).

Failed CTA transfers to and from tape should be reported to the EOS end-user by setting appropriately-named extended attributes on the corresponding EOS namespace entries. For example, a retrieve operation that fails due to a full EOS disk server (reporting ENOSPC to a tape server) could be reported by the tape server setting the value of the file's `last_retrieve_result` attribute to "Disk space full".

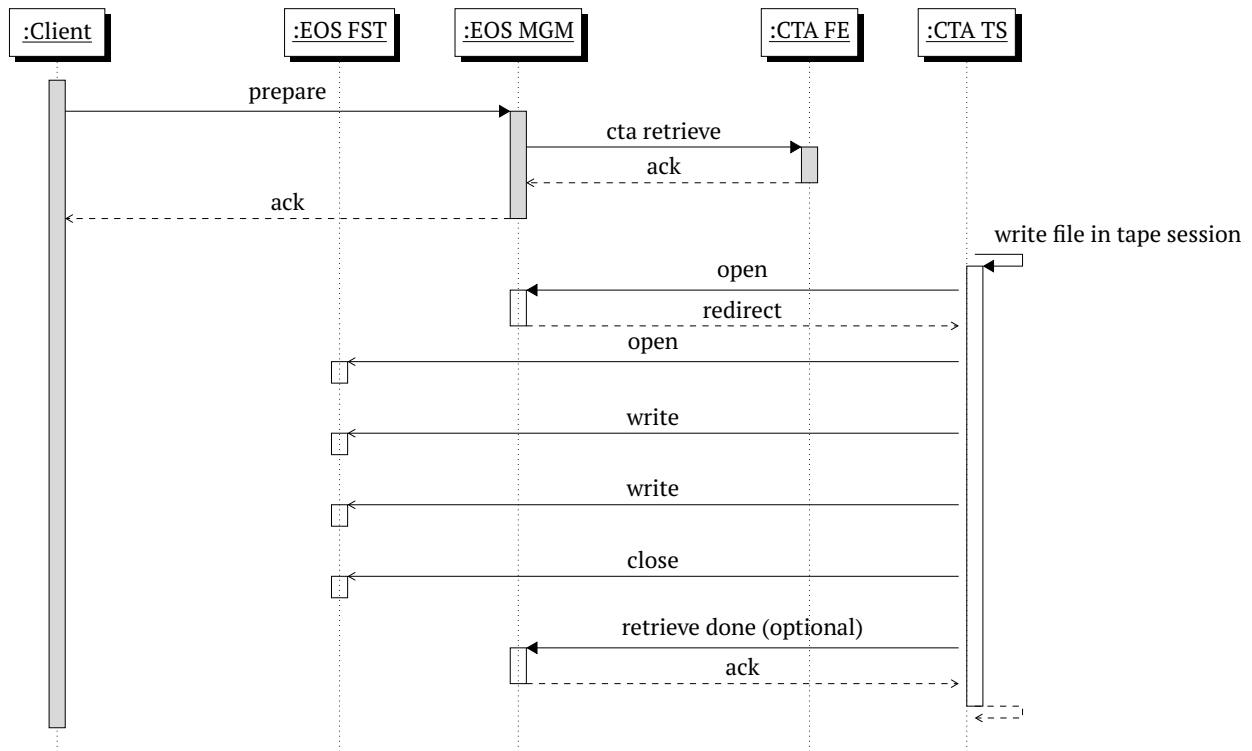


Figure 2.2: File read from tape with explicit prepare (synchronous)

2.4 Implicit retrieve (open for read)

When a user opens a file with copies only on tape, EOS should implicitly generate a retrieve request to CTA. In turn, CTA should return a time estimate for the file retrieve arrival, so EOS can return a try timing to the XrootD client, which will retry the open at that time.

As EOS will not keep track of retrieve requests it issued, CTA should have an idempotent retrieve queuing.

2.5 User-triggered disk copy removal

CASTOR has learned that it is not easy or even possible to implement the exact “garbage collection” policy required by experiments when it comes to deleting disk copies of files safely stored on tape. CASTOR has provided the `stager_rm` command to end users to enable them to manually garbage collect files in their CASTOR disk cache. We currently believe that an equivalent of the `stager_rm` command should be implemented in EOS. Such a command could simply be a request to execute a `stager_rm` workflow action on a specific file.

2.6 Garbage collection of disk copies

A double-criteria garbage collection will probably be necessary to keep free space in disk pools (file age (LRU/FIFO/etc. ...) + pinning).

2.7 Complete deletion of files

There is no interest in reporting failure to delete the file in CTA while the deletion proceeds in EOS, so synchronous and asynchronous implementations are equivalent. The complete deletion of files from EOS raises several race conditions (delete while archiving, delete while retrieving), but all will probably be resolved by failure of data or metadata operations initiated from CTA to EOS, plus slow

reconciliation. The deletion of the file can be represented by a notification message (as any file operations can).

2.8 File (re-)injection

CTA should be able to inject new files into the EOS tree when:

- Files are imported from CASTOR to EOS-CTA.
- Files are re-injected in the EOS tree (using disaster recovery data from the CTA file catalogue).

In both cases, CTA must support the fact that EOS will treat this file as a new one and issue it a new EOS file id. The files will remain referenced in CTA as long as they exist on tape (as long as the tape is not rewritten or removed from the catalogue). Every existing tape file should be referenced (even for deleted files, or failed writes if we add support for skipped writes in the future).

2.9 Slow reconciliation

The slow reconciliation would scan the entire list of files existing in one EOS instance. CTA could then detect the files which are missing on its side, and the ones which are not known to EOS anymore. Metadata changes in EOS will as well be propagated to CTA during this process. Extra levels of safety could be added (crossing sizes, checksums, etc. ...) at the cost of a heavier streaming from EOS. We would then need a retransmit request operation (could be triggering the proper workflow in EOS), and possibly another operation allowing the confirmation of non-existence of a file.

The slow reconciliation would be done against the files listed as belonging to a given EOS instance in the CTA catalog. The listing needs to include the metadata or a mean to detect its changes, and re-create archive requests if needed.

An ideal reconciliation rate would be one week.

2.10 Operations summary

CTA-initiated operations:

- read file to archive
- report partial archival (optional)
- report complete archival (with metadata for EOS to cross check: "tapereplica")
- write file to retrieve
- set xattr
- get full file list with metadata

EOS-initiated operations:

- archive file
- retrieve file
- delete file

Chapter 3

EOS-CTA Protocol

3.1 Data serialization

All messages that are sent from EOS to CTA and from CTA to EOS will be serialized using Google protocol buffers. A dedicated type of workflow will be assigned but the administrator to the events that should be propagated to CTA. This will trigger the sending of a `notification` message (through an interface still to be decided). The sending must be synchronous, with error propagation to the user. The option of asynchronous events is possible for the ones not requiring error propagation to the user.

The error back-propagation protobuf is still TBD.

The protocol buffer specification used by the EOS fuse mount is not be shared with the EOS/CTA interface in order to keep the development of the fuse mount and EOS/CTA interface completely decoupled.

The communication channel used for EOS-CTA communication is still being decided. The main technical constraint is the threading model: the XrootD server should allow maximum parallelism of the requests serving in the CTA front end. This is used inside the front end to internally pack requests together to improve database and object store bandwidth, in order to meet the performance requirements described in [4.1](#).

The EOS to CTA and CTA to EOS interface must be designed to handle the versioning issues of deployments of EOS and CTA where the interface has been upgraded. Such upgrades would start with a new version of CTA being installed first that can talk both the old and new interface protocol(s). Then each EOS instance would be upgraded, one experiment at a time.

The workflows will no longer set any file properties on the EOS side, as those will be set by CTA via “tape replica” and “xattr” calls.

```
syntax = "proto3";
package eos.wfe;

message Id {
    fixed64 uid      = 1;          //< user identity number
    string username  = 2;          //< user name
    fixed64 gid      = 3;          //< group identity number
    string groupname = 4;          //< group name
}

message Checksum {
    string value     = 1;          //< checksum value
    string name      = 2;          //< checksum name
}

message Clock {
    fixed64 sec      = 1;          //< seconds of a clock
    fixed64 nsec     = 2;          //< nanoseconds of a clock
}
```

```

message Md {
    fixed64 fid      = 1;      ///< file/container id
    fixed64 pid      = 2;      ///< parent id
    Clock ctime      = 3;      ///< change time
    Clock mtime      = 4;      ///< modification time
    Clock btime      = 5;      ///< birth time
    Clock ttime      = 6;      ///< tree modification time
    Id owner         = 7;      ///< ownership
    fixed64 size      = 8;      ///< size
    Checksum cks      = 9;      ///< checksum information
    sfixed32 mode     = 10;     ///< mode
    string lpath      = 11;     ///< logical path
    map<string,string>
        xattr = 12;      ///< xattribute map
};

message Security {
    string host       = 1;      ///< client host
    string app        = 2;      ///< app string
    string name       = 3;      ///< sec name
    string prot       = 4;      ///< security protocol
    string grps       = 5;      ///< security grps
}

message Client {
    Id user           = 1;      ///< acting client
    Security sec      = 2;      ///< client security information
}

message Service {
    string name       = 1;      ///< name of the service
    string url        = 2;      ///< access url of the service
}

message Workflow {
    enum EventType { NONE = 0; OPENR = 1; OPENW = 2; CLOSER = 3; CLOSEW = 4;
                    DELETE = 5; PREPARE = 6; }
    EventType event   = 1;      ///< event
    string queue      = 2;      ///< queue
    string wfname     = 3;      ///< workflow
    string vpath      = 4;      ///< vpath
    Service instance  = 5;      ///< instance information
    fixed64 timestamp = 6;      ///< event timestamp
}

message Notification {
    Workflow wf       = 1;      ///< workflow
    string turl       = 2;      ///< transport URL
    Client cli        = 3;      ///< client information
    Md file           = 4;      ///< file meta data
    Md directory      = 5;      ///< directory meta data
}

message Xattr {
    enum Operation { NONE = 0; GET = 1; ADD = 2; SET = 3; DELETE = 4;}
    fixed64 fid       = 1;      ///< file id
    map<string, string>
        xattrs = 2;      ///< xattribute map
    Operation op      = 3;      ///< operation to execute for this xattr map
}

message Tapereplica {
    enum Status { NONE = 0; OFFTAPE = 1; ONTAPE = 2; ONTAPESAVE = 3;}
    fixed64 fid       = 1;      ///< file id
    Status status      = 2;      ///< state state for file ID
    fixed64 size       = 3;      ///< File size as recorded on tape for cross check
}

```

```

    Checksum cks      = 4;      //< File checksum as computer while writing to tape
}

message Error {
    enum Audience { NONE = 0; EOSLOG = 1; ENDUSER = 2;}
    Audience audience = 1;      //< The intended audience of the error message
    fixed64 code      = 2;      //< Zero means success, non-zero means error
    string message     = 3;      //< An empty if success, else an error message
}

// The following message is used to wrap all messages sent between EOS and its
// peers.
//
// This wrapper message allows new message types to be added to the protocol in
// future.
//
// This wrapper message also allows EOS peers to receive non-EOS messages as long
// as the following two conditions are met:
// 1. The peer uses a wrapper message with exactly the same (simple) structure.
// 2. No two message types use the same numeric tag value.
//
// The structure of this message is based on the "Union Types" section of the
// following Google protocol buffers web page:
//
//   https://developers.google.com/protocol-buffers/docs/techniques
//
// A protocol buffer parser cannot determine a message type based solely on its
// contents. The type field of this wrapper message provides the required metadata.

message Wrapper {
    enum Type {NONE = 0; ERROR = 1; NOTIFICATION = 2; XATTR = 3; TAPEREPLICA = 4;}
    Type type = 1;
    Error error = 2;
    Notification notification = 3;
    Xattr xattr = 4;
    Tapereplica tapereplica = 5;
}

```

3.2 CTA front end extension

As SSIv2 is not available yet, we could use an interim interface over the current CLI. We could modify the current CTA command-line tool to receive a protocol buffer message on its standard in and send that message to the CTA front end as the contents of a virtual file in the CTA front-end's virtual namespace.

The CTA command-line tool currently:

1. Base64 encodes each element of the argv[] array passed to its main() function.
2. Concatenates all of the encoded elements together into a single string with an ampersand '&' between each encoded element.
3. Calls the XrdCl::FILE::Open() method with the resulting string as the name of the file to be opened.

For example executing `cta admin ls` at command prompt would cause the cta program to call:

```
XrdCl::FILE::Open("Y3Rh&YWRtaW4=&bHM=")
```

where `Y3Rh` is the base64 encoding of `cta`, `YWRtaW4` is `admin` and `bHM=` is `ls`. This therefore means the file opened is always named `Y3Rh` which is the base64 encoding of `cta`. The modified CTA command-line tool could use a different file name when sending the protocol buffer message so that the CTA front end could be incrementally modified to continue to handle all of the current `cta` commands by processing the “Y3Rh” file and all of the new protocol buffer messages by handling the opening of the new file, for example `protobuf`.

The EOS workflow engine currently talks to CTA by executing bash scripts that call the `cta` command-line tool. These bash scripts selectively pass the appropriate arguments to the `cta` command-line tool for each required action. When a protocol buffer message is used instead of executing a bash script, the protocol buffer will contain everything known by EOS about the file being acted upon. The CTA front end will therefore receive everything it needs to know about the file and more. It will be responsibility of the CTA front end to only select what information it needs from the protocol buffer message.

3.3 Move to Xrootd’s SSIv2

The **Scalable Service Interface** functionality of XRootD is expected to provide a threading model matching our requirements (i.e. without the per-file serialization of the calls).

3.4 EOS support of tape notions

In order to achieve those function, EOS will keep track of tape-related mode (should have a copy on tape), as well as keeping track of the tape replica status.

Chapter 4

Constraints

4.1 Performance requirements

The CTA metadata performance requirements have been assessed in previous work. This ensures smooth repacking where small files can be encountered. The performance requirements are described in the general document about CTA in section 8.5 (Object Store/Performance considerations).

The data taking performance might be less stringent in terms of metadata, but involves a bigger stack with EOS in front should be assessed in order to have full system performance targets. This includes the bandwidth to be achieved and the latency experienced by the user (a synchronous close on write will increase latency, we should make sure the result is adequate).

4.2 Operational constraints

Before a repack campaign, user should be encouraged to purge any unnecessary data from EOS. After this operation, a reconciliation between the user catalogue and EOS (And then EOS and CTA) should be done to ensure no unexpected data will get deleted during the repack operation.

4.3 Synchronous and Asynchronous Calls

The EOS workflow engine will be modified to support synchronous actions as well as the existing asynchronous ones. A synchronous action will enable an EOS client to call EOS which in turn will call CTA, which will return a “return value” reply that is synchronously relayed back to the EOS client.

A concrete example would be when an EOS client opens a file for creation that is supposed to be eventually archived to tape. EOS will synchronously call CTA in order to determine whether or not the storage class of the file is known and has a destination tape pool. If these two conditions are not met then the EOS client will get an immediate synchronous reply saying the file cannot be created because it cannot be archived to tape.

4.4 Data Integrity

4.4.1 After-the-fact check on archive from EOS

EOS will schedule a second workflow job when an archive is triggered. This will check the archive status and re-trigger it if needed at a later time.

4.5 Security

CTA will filter EOS requests per instance and forbid cross talk between instances. Each instance should be authenticated with a unique simple shared secret (SSS) key.

As CTA will have access to every EOS instance, it is desirable to apply the least privilege principle to CTA and to limit the actions allowed to CTA's SSS key to the strict minimum. This will be achieved by limiting CTA's credentials to the protocol interface.

Appendix A

Questions and Issues

This appendix contains questions and open issues which are still under discussion.

A.1 API

Should the EOS-CTA interface be defined as a shared library (e.g. SSIV2 client plugin) or as a framework (boilerplate code)?

So we just need serialise—send—receive—parse ?

A.2 Communication Layer

RPC channel between EOS and CTA: current strategy is to use SSIV2. In case of unforeseen problems, other options are still open: opaque query or open-write-read-close.

Is adding SSIV2 support as simple as loading the SSI plugin?

A.3 Return value

Notification return structure ("return value") should be defined. It will contain:

- Success
- Action to be taken on success (for example set the “CTA archive ID” extended attribute of the EOS file being queued for archival)
- Failure code
- Failure message for logging in EOS (still under discussion)
- Failure message for the end user executing a synchronous workflow (for example “Cannot open file for writing because there is no route to tape”)

A.4 `xattr`

Reporting of retrieve status could use the `xattr` message (to be confirmed, see [3.1](#)). Reporting of failed archival could also use `xattr`. Reporting of the last error encountered in CTA could also use the `xattr` message (to be confirmed, see [3.1](#)).

A.5 CTA Failure

What is the mechanism for restarting a failed archive request (in the case that EOS accepts the request and CTA fails subsequently)?

If CTA is unavailable or unable to perform an archive operation, should EOS refuse the archive request and report failure to the User?

What is the retry policy?

A.6 File life cycle

Full life cycle of files in EOS with copies on tape should be determined (they inherit their tape properties from the directory, but what happens when the file gets moved or the directory properties changed?).

A.7 Handling of immutability

It is forbidden to update files archived on tape. Devise an update policy for backup-type behaviour (2.2).

A.8 Slow reconciliation interface

Action on storage class change for a file? (postponed to repack?)

Possible admin daemon that handles slow reconciliations and repacks?

Full chain reconciliation should be devised.

A.9 Restoring Deleted Files

A method to re-create a deleted file in EOS from CTA data/metadata should be devised.

We might want to pass the information that a file deletion has been confirmed after reconciliation with the user's catalogue. Also delete could be passed to CTA when the file is moved to the recycle bin in EOS, or when it is definitely deleted from EOS.

A.10 Storage Classes

The list of valid storage classes needs to be synchronized between EOS and CTA. EOS should not allow a power user to label a directory with an invalid storage class. CTA should not delete or invalidate a storage class that is being used by EOS.

A.11 Request Queue

Chaining of archive and retrieve requests to retrieve requests.

Excution of retrieve requests as disk to disk copy if possible.

Catalogue will also keep track of requests for each files (archive and retrieve) so that queueing can be made idempotent.

A.12 Catalogue

Catalogue files could hold the necessary info to recreate the archive request if needed.

Appendix B

EOS-CTA Reconciliation Strategy

B.1 Reconciling EOS file info and CTA disk file info

This should be the most common scenario causing discrepancies between the EOS namespace and the disk file info within the CTA catalogue. The proposal is to attack this in two ways: first (already done) we piggyback disk file info on most commands acting on CTA Archive files ("archive", "retrieve", "cancelretrieve", etc.), second (to be agreed with Andreas) EOS could have a trigger on file renames or other file information changes (owner, group, path, etc.) that calls our `updatefileinfo` command with the updated fields. In addition (also to be agreed with Andreas) there should also be a separate low priority process (a sort of EOS-side reconciliation process) going through the entire EOS namespace periodically calling `updatefileinfo` on each of the known files, we would also store the date when this update function was called (see below to know why).

B.2 Reconciling EOS deletes which haven't been propagated to CTA

Say that the above EOS-side low-priority reconciliation process takes on average 3 months and it is run continuously. We could use the last reconciliation date to determine the list of possible candidates of files which EOS does not know about anymore, by just taking the ones which haven't been updated say in the last 6 months. Since we have the EOS instance name and EOS file id for each file (and Andreas confirmed that IDs are unique and never reused within a single instance), we can then automatically check (through our own CTA-side reconciliation process) whether indeed these files exist or not. For the ones that still exist we notify EOS admins for a possible bug in their reconciliation process and we ask them to issue the `updatefileinfo` command, for the ones which don't exist anymore we double check with their owners before deleting them from CTA.

Note: It's important to note that we do not reconcile storage class information. Any storage class change is triggered by the EOS user and it is synchronous: once we successfully record the change our command returns.

Appendix C

EOS-CTA Authorization Rules

One of the requirements of CTA is to prevent crosstalk between EOS instances belonging to different VOs, e.g. the ATLAS EOS instance should not be able to access (or even know about) files belonging to CMS.

Shared Secrets?

Should we have a different shared secret for each VO, to explicitly prohibit access to files not belonging to that EOS instance?

Redundant Rules?

The highlighted rules below are probably not required.

1. A listStorageClass command should return the list of storage classes belonging to the instance from where the command was executed only
2. A queueArchive command should be authorized only if:
 - the instance provided in the command line coincides with the instance from where the command was executed
 - the storage class provided in the command line belongs to the instance from where the command was executed
 - the EOS username and/or group (of the original archive requester) provided in the command line belongs to the instance from where the command was executed
3. A queueRetrieve command should be authorized only if:
 - the instance of the requested file coincides with the instance from where the command was executed
 - the EOS username and/or group (of the original retrieve requester) provided in the command line belongs to the instance from where the command was executed
4. A deleteArchive command should be authorized only if:
 - the instance of the file to be deleted coincides with the instance from where the command was executed
 - the EOS username and/or group (of the original delete requester) provided in the command line belongs to the instance from where the command was executed
5. A cancelRetrieve command should be authorized only if:
 - the instance of the file to be canceled coincides with the instance from where the command was executed

- the EOS username and/or group (of the original cancel requester) provided in the command line belongs to the instance from where the command was executed
6. An `updateFileStorageClass` command should be authorized only if:
- the instance of the file to be updated coincides with the instance from where the command was executed
 - the storage class provided in the command line belongs to the instance from where the command was executed
 - the EOS username and/or group (of the original update requester) provided in the command line belongs to the instance from where the command was executed
7. An `updateFileInfo` command should be authorized only if:
- the instance of the file to be updated coincides with the instance from where the command was executed