

# The CERN Tape Archive: CTA-EOS Interface

Eric Cano

Michael Davis

Steven Murray

December 1, 2017

# Contents

<b>Contents</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
<b>2 CTA Operations</b>	<b>3</b>
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
<b>3 EOS-CTA Protocol</b>	<b>7</b>
3.1 Data serialization	7
3.2 XRootD SSIv2	7
3.3 EOS-CTA Protocol Versioning	7
3.4 EOS support of tape notions	7
<b>4 Constraints</b>	<b>8</b>
4.1 Performance requirements	8
4.2 Synchronous calls	8
4.3 Data Integrity	8
4.4 Security	8
4.5 Operational constraints	8
<b>A Questions and Issues</b>	<b>10</b>
A.1 What is to be shared between the EOS and CTA projects in order to implement the EOS/CTA interface?	10
A.1.1 Rationale for generic C++ headers	10
A.2 What is the mechanism by which the CTA frontend will determine the individual instance names of the EOS instances sending it archive, retrieve and delete requests?	11
A.3 Will EOS instance names within the CTA catalogue be “long” or “short”, in other words “eosdev” or just “dev”?	11
A.4 Do we want the EOS namespace to store CTA archive IDs or not?	11
A.4.1 Proposed Archive ID Solution	12
A.5 Should the CTA catalogue methods prepareForNewFile() and prepareToRetrieveFile() detect repeated requests from EOS instances?	13
A.5.1 If so how should the catalogue communicate such “duplicate” requests to the caller (Scheduler/cta-frontend plugin)?	13

A.5.2	If the CTA catalogue keeps an index of ongoing archive and retrieve requests, what will be the new protocol additions (EOS, cta-frontend and cta-taped) required to guarantee that “never completed” requests are removed from the catalogue? . . . . .	14
A.6	Return value . . . . .	15
A.7	CTA Failure . . . . .	15
A.8	File life cycle . . . . .	15
A.9	Storage Classes . . . . .	15
A.10	Request Queue . . . . .	15
A.11	Catalogue . . . . .	15
<b>B</b>	<b>EOS-CTA Reconciliation Strategy</b>	<b>16</b>
B.1	Reconciling EOS file info and CTA disk file info . . . . .	16
B.2	Reconciling EOS deletes which haven’t been propagated to CTA . . . . .	16
B.3	Slow reconciliation interface . . . . .	16
B.4	Restoring Deleted Files . . . . .	16
<b>C</b>	<b>EOS-CTA Authorization Rules</b>	<b>18</b>

# 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 a 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. Chapter 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.

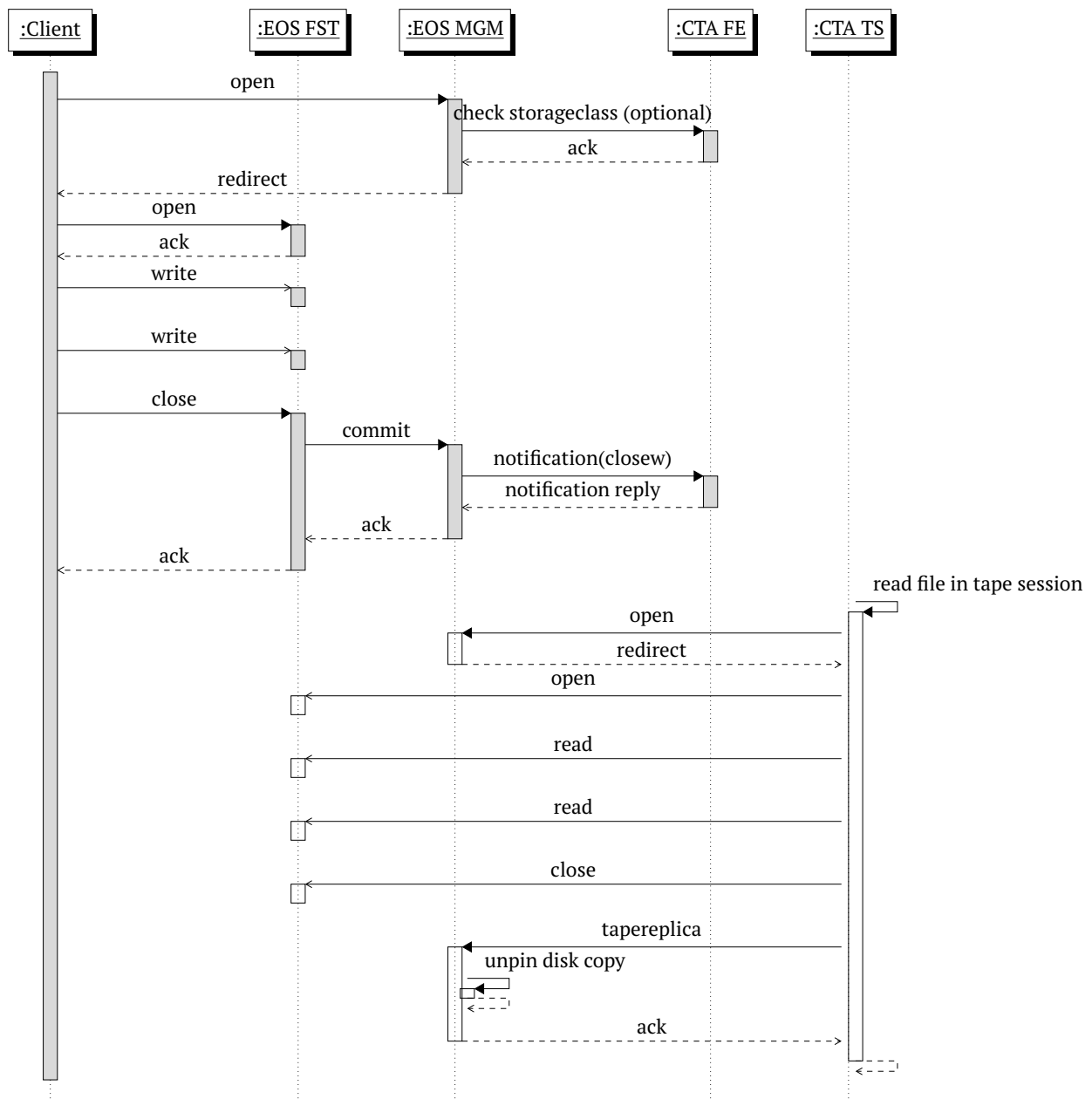


Figure 2.1: File write and archive queuing (synchronous)

- 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".

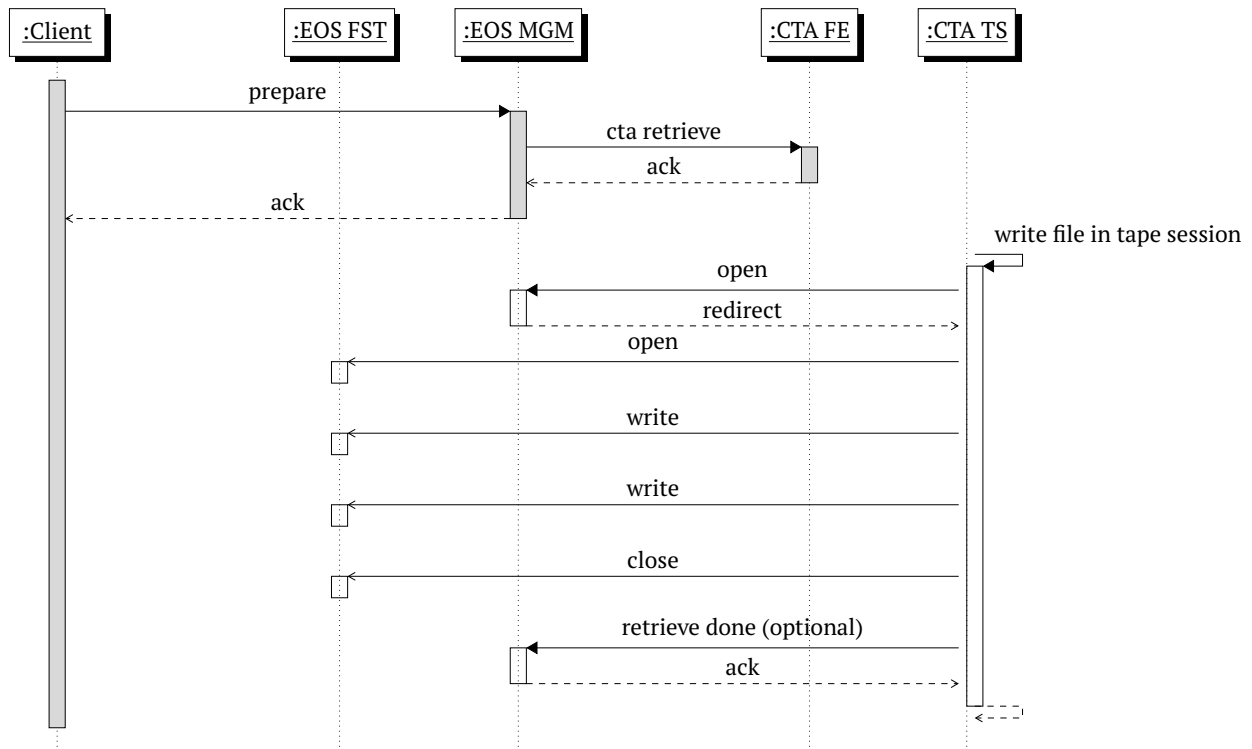


Figure 2.2: File read from tape with explicit prepare (synchronous)

## 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 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 by the administrator to the events that should be propagated to CTA. This will trigger the sending of a `notification` message. The sending must be synchronous, with error propagation to the user. (Asynchronous events could be allowed for events with no requirement for error propagation to the user.)

The workflows will not set any file properties on the EOS side, as those will be set by CTA via “tape replica” and “xattr” calls.

The protocol buffer definition will provide the CTA Frontend with everything EOS knows about the file. It will be responsibility of the CTA Frontend to select only the information it needs from the protocol buffer message.

### 3.2 XRootD SSIv2

The communication channel between EOS and CTA will use the XRootD [Scalable Service Interface version 2](#) (SSIv2) protocol.

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

The XRootD server should allow maximum parallelism of requests served to the CTA Frontend. This is used inside the frontend to internally pack requests together to improve database and object store bandwidth, in order to meet the performance requirements described in [4.1](#).

### 3.3 EOS-CTA Protocol Versioning

The protocol buffer specification will be defined in a separate project shared by the EOS and CTA projects. This project will also include a set of generic headers to couple the protocol buffer definitions with the XRootD SSIv2 transport layer.

In the case that the protocol buffer definitions are updated, a new version of CTA should be installed first, followed by upgrading the EOS instances one-at-a-time. CTA should maintain compatibility with old protocol versions until the upgrades have been rolled out to all experiments.

### 3.4 EOS support of tape notions

EOS will keep track of tape-related mode (should have a copy on tape) and the tape replica status.

# Chapter 4

## Constraints

### 4.1 Performance requirements

The CTA metadata performance requirements have been assessed in previous work<sup>1</sup>. Efficient metadata handling is particularly important for repacking, where small files can be encountered.

The data-taking performance might be less stringent in terms of metadata, but involves a bigger stack with EOS in front. This 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 Synchronous calls

The EOS workflow engine will support synchronous actions. EOS clients will call EOS which in turn will call CTA. CTA will send back a “return value” reply which will be synchronously relayed back to the EOS client.

**Example:** An EOS client opens a file for creation that should be eventually archived to tape. EOS synchronously calls CTA to determine whether or not the Storage Class of the file is known and that it 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.

### 4.3 Data Integrity

**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.

### 4.4 Security

CTA will filter EOS requests per instance and forbid crosstalk 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.

### 4.5 Operational constraints

Before a repack campaign, users should be encouraged to purge any unnecessary data from EOS. After this operation, a reconciliation between the user catalogue and EOS (and then between EOS and CTA)

---

<sup>1</sup>see cta.pdf: Object Store/Performance considerations.

should be done to ensure no unexpected data will get deleted during the repack operation.

# Appendix A

## Questions and Issues

### A.1 What is to be shared between the EOS and CTA projects in order to implement the EOS/CTA interface?

We will set up a new project shared by EOS and CTA which will contain:

1. Generic C++ headers which bind a (generic) Google protocol buffer definition to the XRootD SSIv2 transport layer.
2. The Google protocol buffer definition which instantiates the specific EOS-CTA interface.

The C++ headers are not specific to EOS nor to CTA; in principle they can be used for any project which has a client and server communicating using protocol buffers over XRootD SSIv2. The protocol buffer is the specific instantiation of the EOS-CTA interface.

#### A.1.1 Rationale for generic C++ headers

SSIv2 is a very broad framework with many options. A specific implementation requires a number of design decisions. The SSI generic headers provide:

1. A binding between a generic protocol buffer and the SSIv2 transport layer for the request and response messages
2. Handlers for metadata-only responses/data responses/stream responses
3. A mechanism for making requests/responses synchronous
4. A mechanism for sending alerts from the server to the client
5. Mechanisms for handling the following types of exceptions:
  - Protocol buffer serialization/deserialization errors
  - XRootD transport errors
  - Server errors (CTA Frontend in our case)
  - Timeouts: sending request/processing request

The headers implement these various design decisions, ensuring that the client and server are consistent. It is actually a small amount of code (< 500 lines on the client side).

The headers also allow EOS and the CTA Admin tool to reuse exactly the same interface code on the client side, which reduces maintenance effort as bugs only need to be found and fixed once.

The generic headers can be reused by other projects which are nothing to do with CTA.

The alternative to using generic code would be to document all the design decisions and leave it to the EOS team to implement them.

## A.2 What is the mechanism by which the CTA frontend will determine the individual instance names of the EOS instances sending it archive, retrieve and delete requests?

As CTA will have access to every EOS instance, we want to prevent crosstalk: one VO should not be able to access or interfere with the files from another VO. The principle of least privilege should apply.

SSS keys are used for both identification and authentication. There should be a unique SSS key for each VO. Luca tells us the EOS team are using the following SSS keys:

Number	Len	Date/Time	Created	Expires	Keyname	User & Group
2	32	02/10/12	17:34:12	-----	eosalice	daemon daemon
2	32	12/10/10	15:22:01	-----	eoscms	daemon daemon
1	32	06/25/12	15:42:16	-----	eoslhcb	daemon daemon
1	32	02/14/13	16:20:06	-----	eospublic	daemon daemon
3	32	10/06/14	12:04:59	-----	eosuser	daemon daemon

There are two problems we are aware of:

1. There is an XRootD bug that if the keys are created within the same second, XRoot treats them as different versions of the same key (see issue [#592](#)). The workaround is to have at least a 2-second delay between generating each key, so this is not really an issue in production.
2. XRoot allows us to detect the user name of the key but not the keyname, and currently all key user names are set to the same value (daemon). We need to check with Luca/Andreas if the key user name can be changed in production or find another way to discriminate between the different keys if not.

## A.3 Will EOS instance names within the CTA catalogue be “long” or “short”, in other words “eosdev” or just “dev”?

Following the point above, we should have a separate instance name for each VO (“eosatlas”, “eoscms”, etc.) and a unique key for each instance name.

Personally I prefer to follow the long names that the EOS team are already using, but it’s not a religious issue.

## A.4 Do we want the EOS namespace to store CTA archive IDs or not?

**If no:** we are allowing that the EOS file ID uniquely identifies the file. We must maintain a one-to-one mapping from EOS ID to CTA archive ID on our side. This also implies that the file is immutable.

**If yes:** we must generate the CTA archive ID and return it to EOS. There must be a guarantee that EOS has attached the archive ID to the file (probably as an xattr but that’s up to the EOS team), i.e. **the EOS end-user must never see an EOS file with a tape replica but without an archive ID**. EOS must provide the CTA archive ID as the key to all requests.

Design notes from Steve:

One of the reasons I wanted an archive ID in the EOS namespace was that I wanted to have one primary key for the CTA file catalogue and I wanted it to be the CTA archive ID. Therefore I expected that retrieve and delete requests issued by EOS would use that key.

This “primary key” requirement is blown apart by the requirement of the CTA catalogue to identify duplicate archive requests. The CTA archive ID represents an “archive request” and not an individual EOS file. Today, 5 requests from EOS to archive the same EOS file will result in 5 unique CTA archive IDs. Making the CTA catalogue detect 4 of these requests as duplicate means adding a “second” primary key composed of the EOS instance name and the EOS file ID. It also adds the necessity to make sure that archive requests complete in the event of failure, so that retries from EOS will eventually be accepted and not forever refused as duplicate requests. It goes without saying that dropping the CTA archive ID from EOS also means using the EOS instance name and EOS file ID as primary key for retrieve and delete requests from EOS.

The requirement for a “second” primary key may be inevitable for reasons other than (idempotent) archive, retrieve and delete requests from EOS. CTA tape operators will want to drill down into the CTA catalogue for individual end user files when data has been lost or something has “gone wrong”. The question here is, should it be a “primary key” as in no duplicate values or should it just be an index for efficient lookup?

To summarize:

- We would like to have a unique key to identify files
- In our current design, the CTA Archive ID uniquely identifies archive requests, not files

#### A.4.1 Proposed Archive ID Solution

We could change the current design to allocate the CTA Archive ID to files rather than archive requests, by allocating the ID on file write open instead of file write close. This results in the following workflow:

**On OPENW:** EOS calls CTA with the file metadata

- CTA Frontend validates Storage Class
- CTA Frontend determines destination Tape Pool
- CTA Frontend sends metadata to the Catalogue
- CTA Frontend generates a unique archive ID and returns it to EOS
- EOS attaches archive ID as an xattr

**On CLOSEW:** EOS calls CTA with the file metadata

- The xattrs now include the validated Storage Class and the CTA archive ID
- CTA Frontend queues the archive request and returns status code to EOS. File state is *archive in progress*.
- On successful write of the first tape copy, the tape server notifies EOS. File state is *one copy on tape*. This equates to *m-bit set* in CASTOR.
- On successful write of each tape copy, the tape server notifies EOS. The number of copies on tape can be stored as an xattr.
- On successful write of the last tape copy, the tape server notifies EOS. The number of copies on tape is updated. File state is *archived*.

**On PREPARE:** EOS calls CTA with the file metadata. The file is retrieved. In case of failure, the file will not be retrieved and the reason will be logged.

**On DELETE:** EOS calls CTA with the file metadata. All copies of the file are marked for deletion. In case of failure, the file will not be deleted and the reason will be logged.

In this scheme, if any part of the **OPENW** workflow fails, nothing is archived and no archive ID is attached to the file xattrs, so we are in a guaranteed consistent state and EOS is informed that something went wrong. EOS will not be able to execute the **CLOSEW** workflow without the archive ID.

In the **CLOSEW** workflow, we cannot end up in a state where the file was successfully archived but EOS does not have the archive ID. The only possible inconsistency between EOS and CTA is when we successfully archived at least one copy of the file but did not successfully notify EOS. In this case, the operator should be notified and the EOS user can retry.

If a file is appended to or otherwise modified, this is a new file as far as at CTA is concerned, and a new archive ID will be generated<sup>1</sup>

If EOS loses the archive ID for any reason, no further operations on the file are possible. Inconsistencies between the EOS namespace and CTA namespace will be picked up during reconciliation.

The **OPENW** workflow should be guaranteed to return quickly. There will be a small number of valid Storage Classes, so this can be held in memory. Likewise the process to generate or obtain archive IDs should be fast and run in bounded time.

For CASTOR, there is an additional constraint that the disk copy cannot be deleted until all tape copies have been successfully written. The above scheme keeps track of the number of tape copies written and it will be up to the EOS developers to ensure that this constraint is observed.

## A.5 Should the CTA catalogue methods `prepareForNewFile()` and `prepareToRetrieveFile()` detect repeated requests from EOS instances?

EOS does not keep track of requests which have been issued. We have said that CTA should implement idempotent retrieve queuing.

What are the consequences if we do not implement idempotent retrieve queuing?

What about archives and deletes?

### A.5.1 If so how should the catalogue communicate such “duplicate” requests to the caller (Scheduler/cta-frontend plugin)?

The CTA Frontend calls the Scheduler which calls the Catalogue. There are several possible schemes for handling duplicate jobs:

1. If duplicates are rare, perhaps they don't need to be explicitly handled
2. When a retrieve job is submitted, the Scheduler could check in the Catalogue for duplicates
3. When a retrieve job completes, the Tape Server could notify the Scheduler, which could then check for and drop any duplicate jobs in its queue.

Reporting of retrieve status could set an `xattr`. Then the user would be able to monitor status which could reduce duplicate requests.

Failed archivals or other CTA errors could also be logged as an `xattr`.

<sup>1</sup> Archive files are immutable in any case. This applies only to the backup use case.

**A.5.2 If the CTA catalogue keeps an index of ongoing archive and retrieve requests, what will be the new protocol additions (EOS, cta-frontend and cta-taped) required to guarantee that “never completed” requests are removed from the catalogue?**

Such a protocol addition could be something as simple as a timeout.



## A.6 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.7 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.8 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.9 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.10 Request Queue

Chaining of archive and retrieve requests to retrieve requests.

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

## A.11 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.

### B.3 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.

### B.4 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.

# 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