## Contents

**1 Introduction**

**2 Archive Workflow**
- 2.1 Create file (CREATE) ................................................. 3
- 2.2 Archive file (CLOSEW) .............................................. 4

**3 Retrieve Workflows**
- 3.1 PREPARE ............................................................... 7
  - 3.1.1 retrieve_written workflow (WFE) ................................. 8
  - 3.1.2 retrieve_failed workflow (WFE) ................................ 9
- 3.2 QUERY_PREPARE ....................................................... 9
  - 3.2.1 STAT vs. QUERY_PREPARE ...................................... 9
  - 3.2.2 Request ............................................................. 9
  - 3.2.3 Response .......................................................... 10
- 3.3 ABORT_PREPARE ..................................................... 11
- 3.4 EVICT_PREPARE ..................................................... 12
- 3.5 Implicit retrieve ..................................................... 12

**4 Repack Workflows**
- 4.1 General repack workflow ........................................ 13
  - 4.1.1 Submitting a Repack request .................................. 13
  - 4.1.2 Expansion of a Repack request ................................. 14
  - 4.1.3 Repack Retrieve subrequest execution ......................... 14
  - 4.1.4 Reporting of the Repack Retrieve subrequest ............... 14
  - 4.1.5 Repack Archive subrequest execution ......................... 14
  - 4.1.6 Reporting of the Repack Archive subrequests ............... 14
  - 4.1.7 End of the general Repack workflow ......................... 14
- 4.2 Repack status during the execution of the Repack request ........ 15
- 4.3 Repack "just move“ .................................................. 15
- 4.4 Repack "just add copies“ .......................................... 15
- 4.5 Repack "Move and add copies“ .................................... 15
- 4.6 Repack "tape repair“ ................................................ 16
- 4.7 Other repack functionalities ..................................... 16
  - 4.7.1 Repack a disabled tape ....................................... 16
  - 4.7.2 Repack with a specific mount policy ........................ 16
  - 4.7.3 Repack cancellation ........................................... 17
  - 4.7.4 cta-admin repack ls indicates the destination tapes .... 17

**5 Deletion and Garbage Collection**
- 5.1 User-triggered disk copy removal ............................... 18
- 5.2 Garbage collection of disk copies .............................. 18
5.3 Complete deletion of files ......................................................... 18

6 EOS-CTA Protocol ................................................................. 19
  6.1 Transport Layer .................................................................. 19
  6.2 Protocol Layer ................................................................... 19
  6.3 Protocol Buffer URL Fields ................................................ 20
    6.3.1 Service Access URL ....................................................... 20
    6.3.2 File Archived Report URL ............................................. 20
    6.3.3 Transport Destination URL ............................................ 21
  6.4 Protocol Buffer versions ..................................................... 21
  6.5 EOS support of tape notions ................................................. 22

7 Security, Identification and Authentication ............................... 23
  7.1 Simple Shared Secrets ......................................................... 23
  7.2 Access Rules ..................................................................... 24

A EOS-CTA Reconciliation Strategy ............................................ 26
  A.1 Slow reconciliation ............................................................ 26
  A.2 Reconciling EOS file info and CTA disk file info .................. 26
  A.3 Reconciling EOS deletes which haven’t been propagated to CTA . 27
  A.4 Slow reconciliation interface .............................................. 27
  A.5 Restoring Deleted Files ...................................................... 27

B Constraints ............................................................................ 28
  B.1 Performance requirements .................................................. 28
  B.2 Synchronous calls ............................................................. 28
  B.3 Data Integrity ..................................................................... 28
  B.4 Operational constraints ..................................................... 28

C Questions and Issues .............................................................. 29
  C.1 Success and Failure for Archive Messages ............................ 29
  C.2 Immutable files .................................................................. 29
  C.3 When can files be deleted? .................................................. 29
  C.4 What should be the protocol for fast reconciliation? ............... 30
  C.5 When a file has multiple tape copies, when are notifications are sent to EOS? .............................................. 30
  C.6 Should the CTA catalogue methods prepareForNewFile() and prepareToRetrieveFile() detect repeated requests from EOS instances? .............................................. 30
    C.6.1 If so how should the catalogue communicate such “duplicate” requests to the caller (Scheduler/cta-frontend plugin)? .............................................. 31
    C.6.2 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? .............................................. 31
  C.7 CTA Failure ....................................................................... 31
  C.8 File life cycle ................................................................. 31
  C.9 Storage Classes ............................................................. 31
  C.10 Request Queue ............................................................ 32
  C.11 Catalogue ................................................................. 32
  C.12 Questions administrators need to be able to answer ............. 32
  C.13 User Commands .......................................................... 32
  C.14 Return value [RESOLVED] .................................................. 35
  C.15 Will EOS instance names within the CTA catalogue be “long” or “short”? [RESOLVED] 35
  C.16 Do we want the EOS namespace to store CTA archive IDs or not? [RESOLVED] .............................................. 33
Chapter 1

Introduction

EOSCTA is CERN’s next-generation software solution for the archival and retrieval of data stored on tape. The CERN Tape Archive (CTA) is the tape back-end to the EOS disk system, and together EOS+CTA forms the complete data archival system:

1. The EOS disk system manages the namespace for data stored on tape; provides a storage buffer for files during archival and retrieval operations; and provides the interface between the external client and CTA.

2. CTA manages the tape file catalogue; provides queueing and scheduling functions; and controls data transfers to and from the tape hardware.

3. The third element in the system is the client software, which communicates with EOSCTA using the XRootD protocol. The client is responsible for transferring files into and out of EOS, and managing failures and retries. The most commonly-used client is CERN’s File Transfer System (FTS) but in principle any client which can communicate using the XRootD protocol can be used.

This document describes the EOSCTA workflows (archival, retrieval and deletion of files), and the interfaces and protocols used between the different system components.

The rest of this document is organised as follows:

Chapters 2–5 describe the primary workflows: Archival (Ch. 2), Retrieval (Ch. 3) and Deletion/-Garbage Collection (Ch. 5).

Chapters 6–7 describe the communication between the EOS and CTA components of the system. Ch. 6 describes the communication protocol. Ch. 7 describes the security protocols.

Three appendices discuss the need for reconciliation between EOS and CTA (Appendix A), system constraints (Appendix B) and outstanding questions and issues which need to be agreed on or resolved (Appendix C).
Chapter 2

Archive Workflow

Archive File

- File create: check Storage Class and allocate Archive ID on CREATE event. Archive file after close write (CLOSEW) event.
- File update: denied for files with copy on tape. This can be achieved by administrators by adding an immutable flag (!u) to the ACL of the directories configured to go to tape in EOS, or as a rule in EOS.

Figure 2.1 shows the sequence of a client writing a file to EOS. The storage class is checked on CREATE and a synchronous archive request is queued on CLOSEW.

2.1 Create file (CREATE)

The following operations are carried out on a CREATE event:

- 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
- if no error: CTA Frontend generates a unique Archive ID and returns it to EOS in a synchronous response.
- EOS attaches Archive ID to file as an extended attribute (xattr)

In this scheme, if any part of the CREATE workflow fails, nothing is archived and no Archive ID is attached to the file xattrs. This guarantees that EOS is informed that something went wrong, and we stay in a consistent state. EOS cannot execute the CLOSEW workflow without the archive ID.

CTA will guarantee that the CREATE workflow returns quickly. If there is a need to optimise, the Storage Classes can be held in memory by the CTA Frontend to avoid a database lookup. Likewise the process to generate the Archive ID must be fast and guaranteed to complete in bounded time.

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 (see Appendix A).
2.2 Archive file (CLOSEW)

The following operations are carried out on a file close write (CLOSEW) event:

- 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 synchronously returns a 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.
- 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*. 

![Diagram showing the process of file write and archive queuing (synchronous)]
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.

The number of copies on tape is stored in the EOS namespace under FS ID 65535 (see §6.5).

Reporting metadata in “tape replica” (checksum and size) in addition to archive completion allows EOS to detect discrepancies (as happened when requests got mixed up in initial tests).
Chapter 3

Retrieve Workflows

There are three workflows associated with retrieving files:

- PREPARE (§3.1) to stage a file from tape to the EOS disk buffer
- QUERY_PREPARE (§3.2) to query the status of a file (on disk/not on disk, in-flight or not, any errors)
- ABORT_PREPARE (§3.3) to cancel a PREPARE request
- EVICT_PREPARE (§3.4) to remove a retrieved file from EOS disk

The ALICE/JAlien use case may also need to have a more HSM-like behaviour, Implicit Retrieve (discussed in §3.5).

The communication protocol for the above workflows is shown in Fig. 3.1. For most of our use cases, the client is FTS, which sends bulk requests of 200 files at a time. However, the system will also work with *xrdfs* or any client which supports the XRootD protocol. In the case of ALICE/JAlien, each request contains a single file.

Each request consists of a Request ID and a list of files. In the case of PREPARE, the ReqId is generated by XRootD when the request is created. In the case of QUERY_PREPARE and ABORT_PREPARE, this same ReqId should be provided along with the list of files.

The protocol also allows that the MGM can generate the ReqId and override the XRootD-provided default. This requires that the MGM is started with the configuration option `xrootd.fslib -2 libXrdEosMgm.so`.

![Figure 3.1: EOSCTA Retrieve Workflow Protocols](image-url)
EOS does not maintain the mapping between ReqId and the list of files (i.e. it is not possible to look up the list of files associated with a given request, but it is possible to query a file to see if it has the ReqId attached to it). FTS does maintain the mapping of ReqId to files in its own internal database. For other clients, it is the responsibility of the client to maintain this mapping.

The request is transmitted to the EOS MGM using a standard XRootD FileSystem call. This call is processed in the EOS MGM OFS plugin, where it is unpacked to a list of files (which in the case of JAlien is a list of one element). Each file is dispatched to a handler function in the EOS WorkFlow Engine (WFE). Each file is processed in series. If there is an error for any file in the list, it is skipped over and the next one is processed. Once the list has been processed (successfully or not), the ReqId is returned to the client.

Each call to the handler in the WFE generates a single Google protocol buffer which is transmitted to the CTA Front End across the SSI interface. This call returns a synchronous reply with success or error message for each call.

To simplify handling between the MGM OFS/WFE and to reduce the communication overhead between the MGM and CTA Frontend, it would make sense to forward the requests as a batch to the Frontend. i.e., do not split up the requests in the MGM and send a single message containing the list of files to be retrieved. This change would be straightforward to implement on the CTA side. On the EOS side it would require some refactoring of the OFS plugin.

3.1 PREPARE

Fig. 3.2 shows the full sequence for queuing a PREPARE request and retrieving the file to EOS disk. Inside the MGM OFS plugin, the PREPARE request is unpacked into a list of files. Some basic checks are carried out on each file: the file should exist and the requesting user should have the appropriate permissions to retrieve the file. Then each file is passed one-by-one to the WFE, where the following steps are executed:

1. Check if the file is already on disk; if so take no further action.
2. Check if the sys.retrieve.req_id is set:
   (a) If yes: there is already an in-flight request. Add the ReqId to the list of requests waiting for this file.
   (b) If no: this is the first request for this file. Set sys.retrieve.req_id to the ReqId and set sys.retrieve.error to empty. Send the PREPARE request to the CTA Frontend. Set sys.retrieve.req_time to the time that the PREPARE request was sent.
3. If there is an error: clear the list of requests by setting sys.retrieve.req_id to empty and set the error message in sys.retrieve.error. sys.retrieve.req_time is also cleared.

sys.retrieve.error can also be set by the Garbage Collector. Details to be added.

In addition to the above extended attributes which are set by the WFE, the CTA Frontend also sets an extended attribute when a PREPARE request is queued. The CTA_RetrieveRequestId is set to the address of the retrieve request in the CTA Object Store. This is required for ABORT_PREPARE, because the Object Store has no way to dereference the XRootD request ID into the address of the request object to be deleted. This xattr should be renamed to start with sys. so that it cannot be arbitrarily changed by users. Also the current name is too similar to sys.retrieve.req_id.
Figure 3.2: File read from tape with the synchronous PREPARE workflow

- **#651** Rename the EOS file-attribute named CTA_RetrieveRequestId
- **#660** CTA_RetrieveRequestId is not being wiped after a successful retrieve

sys.retrieve.req_time is set to the last time that a PREPARE request was forwarded to the CTA Frontend. It is intended to be used to detect stale requests (i.e. no successful retrieve but no error either). Automatic detection of stale requests is not implemented but this can be checked by an operator.

- **#663** Automatic detection of stale prepare requests

### 3.1.1 retrieve_written workflow (WFE)

When a file is successfully retrieved from tape, the CLOSEW.retrieve_written workflow is executed. This clears the three attributes sys.retrieve.req_id, sys.retrieve.error and sys.retrieve.req_time.
3.1.2 retrieve_failed workflow (WFE)

CTA may be unable to retrieve the file, due to an error reading the tape or an error writing the file to the disk buffer. In either case, it will retry three times per session.

If the file cannot be retrieved after two retrieve sessions (six attempts in total), the error is stored in `sys.retrieve.error` as above and the list of pending retrieve requests in `sys.retrieve.req_id` is cleared. `sys.retrieve.req_time` is also cleared.

3.2 QUERY_PREPARE

QUERY_PREPARE allows the client to check the status of files which have been requested by a prior PREPARE request.

Implement “xrdfs query prepare” in MGM OFS plugin

3.2.1 STAT vs. QUERY_PREPARE

Previously FTS checked the status of files in a PREPARE request using STAT. The online status of a file is checked using `XrdPosixMap::Flags2Mode`:

```c
if (flags & XrdCl::StatInfo::Offline) *rdv |= XRDSFS_OFFLINE;
```

The `XrdCl::StatInfo::Offline` flag is set if and only if the file has no disk copy (i.e., `d0::t1`). The error status of a file is checked by checking the `sys.retrieve.error` extended attribute.

There were several problems with using STAT:

- A STAT request queries the status of a single file, so it breaks the normal FTS workflow which is to send requests in batches of 200. QUERY_PREPARE allows FTS to query the status of all files in a request at once.
- STAT is not transparent. Some parts of the PREPARE request status are stored in XRootD flags, other parts are stored in EOS extended attributes. If any of these internal implementation details change, FTS has to be updated. In contrast, QUERY_PREPARE is a standard part of the XRootD protocol. It will provide a single, consistent abstract interface to FTS and any future client software.

3.2.2 Request

The XRootD protocol for QUERY_PREPARE takes a ReqId and an optional list of files as its arguments. As the MGM does not maintain the mapping from ReqId to files, the list of files is mandatory in our case.

We can effectively ignore the ReqId as we can query the file using only the filename. However, it is included in the response. If the client makes asynchronous queries, it allows the client to tie which response belongs to which query.
### 3.2.3 Response

The XRootD protocol does not specify the format of the response, other than it is a string. `xrdfs query prepare` simply displays this string on stdout.

As the reply from QUERY_PREPARE needs to be both human-readable and able to be parsed by clients such as FTS, we send the reply in JSON format.

At the time of writing, we are not aware of any other system which implements QUERY_PREPARE. dCache would like to implement it, so we should coordinate with them to ensure that our reply format satisfies their use cases.

The response format is as follows:

```json
{
    "request_id": "43289048340",
    "responses": [
        {
            "path": "/eos/path/to/file",
            "exists": true/false,
            "online": true/false,
            "requested": true/false,
            "has_reqid": true/false,
            "req_time": ",",
            "error_text": ",",
        },
        ...
    ]
}
```

In detail, the response is a JSON object which consists of:

- The same request ID that was passed in the request
- An array of JSON structs, one per file from the original request. The number of array elements will match exactly the number of files in the request. Even if the file does not exist or is inaccessible, there will be a response struct with the `error_text` attribute set.

Each array element has the following fields:

- **String path**: The absolute path to the file (relative paths are explicitly forbidden by the XRoot protocol)
- **Boolean exists**: True if the file exists in the EOS namespace
- **Boolean online**: True if there is a copy of the file on disk
- **Boolean requested**: True if there is at least one request ID attached to the file
- **Boolean has_reqid**: True if the supplied request ID is attached to the file
- **String req_time**: UNIX timestamp (seconds since the Epoch) indicating when the request was sent to CTA to be serviced. The value is a 64-bit number, but it is transmitted as a string as not all JSON parsers have a 64-bit type.
- **String error_text**: If an error occurred, a human-readable error message will be returned here

The client logic for each response should be something like:
if(online):
    SUCCESS
elif(not exists):
    BAD REQUEST: Unrecoverable error, file does not exist
elif(requested):
    WAIT: come back later
else:
    ERROR: pass error_text to data management software stack
endif

There are a couple of corner cases which can be detected as follows:

- If online is false AND requested is false AND error_text is empty, something happened in the MGM and the request never made it to the CTA Frontend. This should be treated as a transient error (resend the request).

- If requested is true, but has_reqid is false, this means that our request never made it to the CTA Frontend, but anyway the file is being retrieved as someone else managed to request it. This should also be treated as a transient error (resend the request), to cover the case where the other requester cancels their request.

- If req_time is very old (e.g. > 1 week), probably something went badly wrong and the request has been lost. It is up to the experiment to decide whether an operator should be notified or whether the request should simply be retried.

### 3.3 ABORT_PREPARE

ABORT_PREPARE allows the client to cancel PREPARE requests which are not “interesting” any more. One typical use case (commonly used by ATLAS) is when the PREPARE request for a file is simultane-ously sent to CERN T0 and one or more T1 sites. As soon as one of the requests succeeds, the other ones are cancelled.

In CASTOR, the queue for “bring online” is seven days long. Users can cancel “bring online” at any time. CTA follows this same behaviour.

The ABORT_PREPARE workflow is implemented as follows:

```python
if(req_id in sys.retrieve.req_id):
    // Request is still in-flight
    Remove req_id from sys.retrieve.req_id
if(sys.retrieve.req_id is empty):
    Send ABORT_PREPARE request to the CTA Frontend
    Clear sys.retrieve.error and sys.retrieve.req_time
endif
else:
    // Either the file has been retrieved successfully,
    // an error has occurred,
    // or the client did not provided the correct req_id
    No action
endif
```
3.4 EVICT_PREPARE

The EVICT_PREPARE workflow offers the same functionality as the CASTOR stager_rm command. It can be triggered using `eos stagerrm` or `xrdfs prepare -e`.

Whereas ABORT_PREPARE is typically triggered by an experiment's data management software, EVICT_PREPARE is intended for use by operators. ABORT_PREPARE is for cancelling in-flight requests, while EVICT_PREPARE is for removing files which have been successfully retrieved.

To be added: description of how EVICT_PREPARE works, including which xattrs are updated.

To be added in future: EVICT_PREPARE should take a request ID like ABORT_PREPARE does.

3.5 Implicit retrieve

The ALICE/JAlien use case may require us to implement an implicit prepare workflow. That is, when a user attempts to open an EOS file which only has a tape copy, EOS should implicitly generate a PREPARE request.

There are two cases: (a) hanging PREPARE, (b) implicit open for read

This has not yet been implemented. Currently under discussion with ALICE.
Chapter 4

Repack Workflows

Repacking a tape is very useful to an operator who wants to migrate data from one tape to another, to repair a tape, or to add missing copies of several files.

This chapter covers all the Repack functionalities that have been implemented for now.

4.1 General repack workflow

4.1.1 Submitting a Repack request

In order to submit a Repack request, the user should use the `cta-admin repack` command tool:

```
cta-admin re/repack add/rm/ls/err:
```

This command allows to manage repack requests.

Submit a repack request by using the "add" subcommand:

- Specify the vid (\(--vid\) option) or all the vids to repack by giving a file path to the \(--vidfile\) option.
- If the \(--bufferURL\) option is set, it will overwrite the default one. It should respect the following format: root://eosinstance//path/to/repack/buffer.
  - The default bufferURL is set in the CTA frontend configuration file.
- If the \(--justmove\) option is set, the files located on the tape to repack will be migrated on one or multiple tapes.
- If the \(--justaddcopies\) option is set, new (or missing) copies (as defined by the storage class) of the files located on the tape to repack will be created and migrated.
  - By default, CTA will migrate AND add new (or missing) copies (as defined by the storage class) of the files located on the tape to repack.
  - The \(--mountpolicy\) option allows to give a specific mount policy that will be applied to the repack subrequests (retrieve and archive requests).
  - By default, a hardcoded mount policy is applied (every request priorities and minimum request ages = 1).
- If the \(--disabledtape\) flag is set, the tape to repack will be mounted for retrieval even if it is disabled.

```
add \[--vid/-v <vid>\] \[--vidfile/-f <filename>\] \[--bufferurl/-b <buffer URL>\]
\[--justmove/-m\] \[--justaddcopies/-a\]
\[--mountpolicy/-u <mount_policy_name>\] \[--disabledtape/-d\]
rm \(--vid/-v <vid>\)
ls \|--vid/-v <vid>\)
err \|--vid/-v <vid>\)
```
After the submission of the Repack request, it will be queued in the **RepackQueuePending**. The maintenance process will then pop the repack request from the **RepackQueuePending** and will start the repack request **expansion**.

### 4.1.2 Expansion of a Repack request

The expansion of the Repack request is the "transformation" of the Repack request into multiple retrieve requests. In order to do that, the expansion algorithm will ask the CTA Catalogue to give him all the files that are located in the source tape (method `catalogue.getArchiveFilesForRepackItor(sourceVID, fSeq)`).

For each files in the source tape, the scheduler will create a Retrieve subrequest and queue each subrequest into the **RetrieveQueueToTransfer**.

### 4.1.3 Repack Retrieve subrequest execution

According to the mount policies given to the Repack request during its submission, the Retrieve subrequests will be popped from the **RetrieveQueueToTransfer** and will be executed. The successful Retrieve requests will be queued in the **RetrieveQueueToReportToRepackForSuccess**.

The failed Retrieve requests will be queued in the **RetrieveQueueToReportToRepackForFailure** (after 5 attempts of Retrieving).

### 4.1.4 Reporting of the Repack Retrieve subrequest

The maintenance process will pop the Retrieve subrequest queued in the **RetrieveQueueToReportToRepackForSuccess** and in the **RetrieveQueueToReportToRepackForFailure**. In the case of success or failure, the Repack Request will have its statistics updated (retrieved files, retrieved bytes, failed to retrieve files, failed to retrieve bytes).

The successful Retrieve subrequests will be transformed into **Archive subrequests** and queued into the **ArchiveQueueToTransferForRepack**.

### 4.1.5 Repack Archive subrequest execution

The repack Archive subrequests will be popped from the **ArchiveQueueToTransferForUser** and will be executed. The successful ones will be queued into the **ArchiveQueueToReportToRepackForSuccess**, the failed ones will be queued in the **ArchiveQueueToReportToRepackForFailure**.

### 4.1.6 Reporting of the Repack Archive subrequests

The maintenance process will pop the Archive subrequests queued in the **ArchiveQueueToReportToRepackForSuccess** and in the **ArchiveQueueToReportToRepackForFailure**. It will then update the Repack Request statistics (Archived files, Archive bytes, failed to archive files, failed to archive bytes).

### 4.1.7 End of the general Repack workflow

When all the Retrieve and Archive subrequests are reported as successful or failed, the Repack Request will have its status updated as **Complete** or **Failed**.
4.2 Repack status during the execution of the Repack request

A Repack request can have all these status during the Repack workflow:

- Pending : the Repack request is in the RepackQueuePending waiting to be popped by the maintenance process of a tapeserver.
- ToExpand : the Repack request is in the RepackQueueToExpand waiting to be popped by the maintenance process of a tapeserver.
- Running : The first Retrieve or Archive subrequest has been reported as successful or failed
- Complete : All the Retrieve and Archive subrequest are completed have been reported as successful to the Repack Request.
- Failed : All the Retrieve and Archive subrequest are completed but at least one Retrieve or Archive subrequest has failed.

4.3 Repack "just move"

The Repack "just move" workflow allows the user to move the files located in a source tape into another one (destination tape).

The files located in the source tape will be superseded by the ones located in the destination tape. After a successful repack "just move", the source tape can be reclaimed.

In order to launch a Repack "just move" workflow, add the "-m" option to the repack add command.

```
cta-admin repack add --vid V01001 -m
```

4.4 Repack "just add copies"

The Repack "just add copies" workflow allow the user to create missing copies of the files that are on the source tape. In order to do that, the operator will have to update the storage class of the files present on the source tape in order to increase the number of copies the files should have.

The expansion algorithm of the Repack request will create the Retrieve subrequest and indicate them that multiple files should be archived. According to this, one succesful Retrieve subrequest will be transformed into multiple Archive subrequests.

In order to launch a Repack "just add copies", add the "-a" option to the repack add command.

```
cta-admin repack add --vid V01001 -a
```

4.5 Repack "Move and add copies"

This feature is the combinaison of the two previous one. It will allow the user to move data from the source tape to another destination one and to create the missing copies of the files of the source tape.

In order to launch this workflow, no flags have to be added to the command :
4.6 Repack "tape repair"

This workflow allows the operator to reinject files into CTA via Repack.

Imagine that a tape is broken and that 10 out of 100 files could not be retrieved from the tape with a normal Retrieve request. The operator could try to recover the files with specific tools and copy these files directly into the Repack buffer. The name of each copied file should contain 9 characters and be named according to their fSeq in the source tape. Example: for the file located at the fSeq 10 on the source tape, the name of the file copied in the buffer has to be 000000010.

The operator will then launch the Repack request.

During the expansion algorithm loop, the tapeserver will detect that 10 files are already in the buffer. It will then create 10 Retrieve requests with the status `ToReportToRepackForSuccess` and queue them in the `RetrieveQueueToReportToRepackForSuccess`. These 10 Retrieve requests will then be transformed into Archive requests and the Repack process will continue.

4.7 Other repack functionalities

The functionalities presented here are other Repack-related functionalities that are implemented.

4.7.1 Repack a disabled tape

Currently, it is impossible for an operator to Retrieve files from a tape that is disabled. Launching a Repack request on a disabled tape without a specific option will fail the Repack request because it is not possible to Retrieve files from a disabled tape.

In order to override this behavior, the operator can set a flag `-disabledtape` or `-d`. CTA will then know that the disabled tape could be mounted in order to Repack it.

4.7.2 Repack with a specific mount policy

The operator can launch a Repack request on a tape with a specific mount policy. In order to do that, he will have to give the mount policy name to the Repack request using the option `-mountpolicy` or `-u`.

If the operator does not specify a mount policy name, the default mount policy for Repack will be applied to all the subrequests. The default Repack mount policy is:
archivepriority = 1
minarchiverequestage = 1
retrievpriority = 1
minretrieverequestage = 1

4.7.3 RePack cancellation

The operator can cancel a running RePack request by using this command:

```
cta-admin repack rm --vid V01001
```

The CTA frontend will then remove all the RePack subrequests and the RePack request itself from the objectstore.

4.7.4 cta-admin repack ls indicates the destination tapes

By using the flag --json, an operator can see in which destination tape the archived files from a RePack request have been written to.

```
bash: cta-admin --json re ls | jq
```

```json
[
 { 
   "vid": "V01001",
   "repackBufferUrl": "root://ctaeos//eos/ctaeos/repack",
   "userProvidedFiles": "0",
   "totalFilesToRetrieve": "1153",
   "totalBytesToRetrieve": "17710080",
   "totalFilesToArchive": "1153",
   "totalBytesToArchive": "17710080",
   "retrievedFiles": "1153",
   "archivedFiles": "1153",
   "failedToRetrieveFiles": "0",
   "failedToRetrieveBytes": "0",
   "failedToArchiveFiles": "0",
   "failedToArchiveBytes": "0",
   "lastExpandedFseq": "1153",
   "status": "Complete",
   "destinationInfos": [
   { 
     "vid": "V01003",
     "files": "1153",
     "bytes": "17710080"
   }
   ]
 }
]```
Chapter 5

Deletion and Garbage Collection

Delete File

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

5.1 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.

5.2 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).

5.3 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).
Chapter 6

EOS-CTA Protocol

6.1 Transport Layer

The communication channel between EOS and CTA is the XRootD Scalable Service Interface v2 (SSIv2) protocol.

The XRootD SSI protocol allows maximum parallelisation of requests served to the CTA Frontend, i.e. without per-file serialization. The CTA Frontend internally packs requests together to improve database and object store bandwidth, to meet the performance requirements described in §B.1.

6.2 Protocol Layer

All Requests and Responses sent between EOS to CTA are defined and serialized using Google Protocol Buffers v3.

- A workflow event of type `proto:` is assigned by the administrator to directories where events should be propagated to CTA.

- When a workflow event occurs, a `notification` message will be sent to the CTA Frontend. Notifications are synchronous: EOS will receive an acknowledgement or an error message.

- The workflows do not set any file properties on the EOS side. These are set by CTA via the `xattr` map in the Response message or by one of the URL callbacks specified below.

- The Protocol Buffer definition provides the CTA Frontend with everything EOS knows about the file. The CTA Frontend will select only the information it needs from the Protocol Buffer message.

The Protocol Buffers are defined in the `xrootd-ssi-protoBuf-interface` project, which is shared between EOS and CTA. This project comprises:

- A set of generic C++ headers to bind a set of Protocol Buffer definitions to the XRootD SSIv2 transport layer\(^1\).

- The Protocol Buffer definitions which instantiate the EOS–CTA interface.

\(^1\)The headers are not specific to EOS or CTA; in principle they can be used for any project which has a client and server communicating using protocol buffers over XRootD SSIv2.
6.3 Protocol Buffer URL Fields

The event notification Protocol Buffer includes three URL fields which are constructed by EOS:

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

message Workflow {
  ...
  cta.common.Service instance = 5; //< instance information
  ...
}

message Transport {
  string dst_url = 1; //< transport destination URL
  string report_url = 2; //< URL to report successful archiving
}
```

The access URL and report URL are sent with archive events.
The transport destination URL is sent with retrieve events.

### Retrieval Notification Message

Possibly we need another callback URL to provide notification of a successful retrieval/error message in case of a failed retrieval (under discussion).

6.3.1 Service Access URL

The Service Access URL Request.notification.wf.instance.url is used by the CTA Tape Server daemon `cta-taped` to read the disk file from the EOS MGM during an archive event.

Example URL:

```
root://ctapps-eossrv01.cern.ch//eos/ctapps/preprod/ed8f3136/0/test099999?eos.lfn=291198a
```

which is composed as follows:

```
root://<hostname>.cern.ch/<path>/<filename>?eos.lfn=<fid>
```

- `<hostname>` is the hostname of the EOS MGM
- `<path>/<filename>` is the absolute EOS path to the file
- `<fid>` is the file ID, given in hex format as `fxid:<fid-hex>`

6.3.2 File Archived Report URL

The File Archived Report URL Request.notification.transport.report_url is used by the Tape Server to asynchronously report to the EOS MGM that a file has been safely archived to tape.
The URL is composed as follows:


- `mgm.pcmd` tells the MGM to execute a workflow event (event)
- `mgm.fid` gives the file ID of the file that was archived, in hexadecimal digits
- `mgm.logid` gives the Log ID (cta)
- `mgm.event` sets which workflow event to execute (archived)
- `mgm.workflow` sets which workflow to execute (default)
- `mgm.path` gives the path to the EOS disk file
- `mgm.ruid` sets the user ID of the user invoking the workflow
- `mgm.rgid` sets the group ID of the user invoking the workflow

The MGM will respond by executing the `sys.workflow.archived.default` workflow against the specified file as user/group `<ruid>`:`<rgid>`.

### 6.3.3 Transport Destination URL

The Transport Destination URL `Request.notification.transport.dst_url` is used by the Tape Server to write a tape file to the EOS MGM during a `retrieve` event.

The URL is composed as follows:

```
root://<hostname>.cern.ch/<path>/<filename>?eos.lfn=<fid>&eos.ruid=<ruid>&eos.rgid=<rgid>&eos.injection=1&eos.workflow=none
```

- `eos.lfn` specifies the file ID, given in hex format as `fxid:<fid-hex>`
- `eos.ruid` sets the user ID of the user invoking the workflow
- `eos.rgid` sets the group ID of the user invoking the workflow
- `eos.injection=1` informs EOS that we are sending it data to write to the file
- `eos.workflow` sets which workflow event to execute (none)

**Note:** When EOS writes a file to disk, the `closew.default` workflow will be triggered, to archive the file on tape. This workflow should obviously not be triggered when the file being written is being retrieved from tape. The URL overrides the default workflow by explicitly setting it to `none`.

### 6.4 Protocol Buffer versions

If the Protocol Buffer definitions are updated:

- First install the new version into CTA.
- Next, upgrade EOS instances (one-at-a-time).
CTA must maintain compatibility with old protocol versions until the upgrades have been rolled out to all experiments.

6.5 EOS support of tape notions

EOS keeps track of tape-related information using extended attributes:

- The `CTA_StorageClass` attribute indicates that one or more copies should be stored on tape under the specified Storage Class. The CTA database will specify the number of tape copies that should be created for that Storage Class.

- EOS reserves filesystem ID 65535 for tape operations. This should be specified in all tape directories using the extended attribute `CTA_TapeFsId="65535"`. EOS treats FS ID 65535 as a phantom filesystem in the EOS namespace, using it to record the number of copies stored on tape.
Chapter 7

Security, Identification and Authentication

As CTA has access to every EOS instance, it must prevent crosstalk between EOS instances belonging to different VOs. This is achieved by applying the least privilege principle, i.e. the ATLAS EOS instance should not be able to access (or even know about) files belonging to CMS.

The EOS–CTA Protocol Buffer identifies the instance sending the Request. The CTA Frontend must also be able to authenticate that the instance name is legitimate.

7.1 Simple Shared Secrets

XRootD uses Simple Shared Secrets (SSS) to authenticate Protocol Buffer requests. In order to authenticate the instance name, there must be a unique SSS key for each VO.

The EOS team say that the following SSS keys are in use:

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

XRootD 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). However, when the key is communicated from client to server, only the key value is sent across the wire. The value for the \texttt{u:} (user) field is read from the keytab on the server side.

Therefore, the solution to the problem of using the SSS key to validate the instance name in the Protocol Buffer is simply to edit the \texttt{u:} field for all keys in the CTA Frontend keytab so that they match the expected instance names. This also needs to be implemented in the Frontend:

```cpp
#206 Determine EOS instance name from both the CTA frontend SSS keytab and the notification.instance().name() protocol buffer field
```

The \texttt{u:} field from the keytab is passed as a parameter to \texttt{Service::ProcessRequest()}. The \texttt{g:} (group) field does not appear to be used. \texttt{n:} (keyname) is a unique identifier within the keytab and appears to be used for key management only.
SSS key conflict resolution should be key-name based and not creation-date based.

There is an XRootD bug affecting the generation of unique keys: if the keys are created within the same second, XRoot treats them as different versions of the same key. The workaround is to have at least a two-second delay between generating each key, so this is not really an issue in production.

### 7.2 Access Rules

#### 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
Appendix A

EOS-CTA Reconciliation Strategy

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

A.1 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.

A.2 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).
A.3 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.

A.4 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.5 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 B

Constraints

B.1 Performance requirements

The CTA metadata performance requirements have been assessed in previous work\textsuperscript{1}. 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).

B.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.

B.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.

B.4 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) should be done to ensure no unexpected data will get deleted during the repack operation.

\textsuperscript{1}see cta.pdf: Object Store/Performance considerations.
Appendix C

Questions and Issues

C.1 Success and Failure for Archive Messages

Results of a discussion between Jozsef, Giuseppe and Eric about the success and failure behaviour for archive requests:

The current behaviour in CASTOR is that the file remains to be migrated until all copies have been successfully migrated. Failed migration jobs get deleted, so the file cannot be stager_rmed or garbage collected before an operator intervenes.

We think a similar behaviour should be implemented in EOSCTA:

- The file will be opened to garbage collection when all copies are successfully archived to tape. CTA will not report success before that (this is already the case).
- When a failure occurs (after exhausting retries), CTA will report the error to EOS, with a new error-reporting URL (to be implemented). The job will then be placed in a failed job queue to be handled by operators.
- EOS will keep track of and expose to the user only the latest error (we have potentially one per tape copy, and if operator decides to retry the job entirely, the error could be reported again).
- EOS will clear the errors when receiving a success.

C.2 Immutable files

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).

Immutability of a file is guaranteed by adding $u!$ to the EOS ACL.

Currently we do not enforce this on the CTA Frontend side, we just assume EOS is taking care of it.

If we decide it’s useful for CTA to check immutability of archived files, we could send the ACL across with the xattrs. This is not sent at the moment, because all system and user attributes are filtered out.

C.3 When can files be deleted?

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.

C.4 What should be the protocol for fast reconciliation?

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 which CTA reports asynchronously to EOS (see §6.2).

C.5 When a file has multiple tape copies, when are notifications are sent to EOS?

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 §6.2).

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.

In CASTOR, the following notifications are sent during archiving a file with $n$ tape copies:

- On successful write of the first tape copy, the m-bit is set. This indicates to the experiment that they can safely delete their copy of the data.
- On successful write of the $n^{th}$ tape copy, the CAN_BE_MIGR status is set in the database. This indicates that the file can be deleted from CASTOR's staging area.

For CTA, at what point(s) should we notify EOS that a file has been archived?

- After the first copy is archived?
- After each copy is archived?
- After the $n^{th}$ copy is archived?

C.6 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?
C.6.1 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.

C.6.2 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.

C.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?

C.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?).

C.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.
C.10 Request Queue

Chaining of archive and retrieve requests to retrieve requests. Execution of retrieve requests as disk to disk copy if possible.

C.11 Catalogue

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

C.12 Questions administrators need to be able to answer

The cta-admin command should be include functions to allow administrators to answer the following questions:

- Why is data not going to tape?
- Why is data not coming out of tapes?
- Which user is responsible for system overload?

C.13 User Commands

What user commands are required? This needs to be reviewed. From the previous documentation:

For most commands there is a short version and a long one. Due to the limited number of USER commands it is not convenient (nor intuitive) to use subcommands here (anyway it could be applied only to storage classes).

- cta lsc/liststorageclass
- cta ssc/setstorageclass <dirpath> <storage_class_name>
- cta csc/clearstorageclass <dirpath>
- cta mkdir <dirpath>
- cta chown <uid> <gid> <dirpath>
- cta rmdir <dirpath>
- cta ls <dirpath>
- cta a/archive <src1> [ [<src2> [ [<src3> [...] ] ] ] <dst>
- cta r/retrieve <src1> [ [<src2> [ [<src3> [...] ] ] ] <dst>
- cta da/deletearchive <dst>
- cta cr/cancelretrieve <dst>

1 this command might seem a duplicate of the corresponding admin command but it actually shows a subset of fields (name and number of copies)
2 we may want to add chmod later on
3 this works both on ongoing and finished archives, that is why it’s called “delete”
4 this clearly works only on ongoing retrieves, obviously does not delete destination files, that’s why it’s called “cancel”
C.14 Return value [RESOLVED]

Notification return structure for synchronous workflows contains the following:

- **Success code** (RSP_SUCCESS)
- A list of extended attributes to set (e.g., set the “CTA archive ID” xattr of the EOS file being queued for archival)
- **Failure code** (RSP_ERR_PROTOBUF, RSP_ERR_CTA or RSP_ERR_USER)
- Failure message which can be logged by EOS or communicated to the end user (e.g., “Cannot open file for writing because there is no route to tape”)

```cpp
message Response {
  enum ResponseType {
    RSP_INVALID = 0;  // Response type was not set
    RSP_SUCCESS = 1;  // Request is valid and was accepted for processing
    RSP_ERR_PROTOBUF = 2;  // Framework error caused by Google Protocol Buffers layer
    RSP_ERR_CTA = 3;  // Server error reported by CTA Frontend
    RSP_ERR_USER = 4;  // User request is invalid
  }
  ResponseType type = 1;  // Encode the type of this response
  map<string, string> xattr = 2;  // xattribute map
  string message_txt = 3;  // Optional response message text
}
```

C.15 Will EOS instance names within the CTA catalogue be “long” or “short”? [RESOLVED]

We all agreed to use “long” EOS instance names within CTA and specifically the CTA catalogue. An example of a long EOS instance name is “eosdev” with its corresponding short instance name being “dev”.

— Minutes from today’s tape developments meeting, Wed 22 Nov 2017

This implies that there will be a separate instance name for each VO (“eosatlas”, “eoscms”, etc.) and a unique SSS key for each instance name.

C.16 Do we want the EOS namespace to store CTA archive IDs or not? [RESOLVED]

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.
Solution

Archive IDs will be allocated by CTA when a file is created. The Archive ID will be stored in the EOS namespace as an extended attribute of the file. EOS must use the archive ID to archive, retrieve or delete files.

Archive IDs are not file IDs, i.e. the archive ID identifies the version of the file that was archived. In the case of Physics data, the files should be immutable so in practice there is one Archive ID per file.

In the backup use case, if we allowed mutable files, we would need a mechanism to track archived file versions. On the EOS side, changes to files are versioned, so each time a file is updated, the Archive ID should also be updated. Old versions of the file would maintain a link to their archive copy via the versioned extended attributes. But in this case we probably also need a way to mark archive copies of redundant versions of files for deletion.

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?