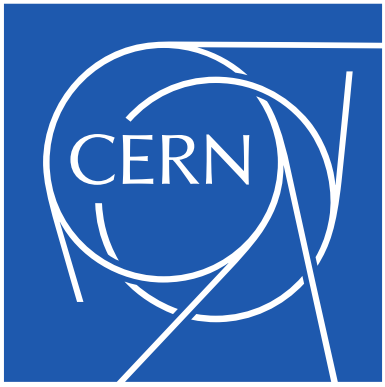

The CERN Tape Archive

Overview and Design

Germán Cancio Eric Cano Michael Davis
Daniele Kruse Steven Murray

April 7, 2020

Contents

Glossary

Archive Route Specifies the set of tapes on which the copies of an archive file will be written, i.e. the relationship between Storage Classes and Tape Pools. There is an archive route for each copy in each storage class. Normally there should be a single archive route per tape pool. 3, 4

CASTOR CERN Advanced STORage Manager. 2

CTA CERN Tape Archive. 2, 3

Data and Storage Services (DSS) Data and Storage Services group in the CERN IT department. 2

EOS A disk-based, low-latency storage service with a highly-scalable hierarchical namespace, using the XRoot protocol for data access possible. (The name **EOS** is not an acronym, it was inspired by the Greek goddess of the dawn, $E\omega\sigma$). 2, 3

HSM Hierarchical Storage Management. 2

Logical Library Specifies which tapes are mountable into which drives. Each tape and each drive belongs to exactly one logical library. The mountability criteria is based on physical location (the tape and the drive must be in the same physical tape library) and on read/write compatibility. 3, 4

Mount Group Creates a link between Users and Mount Policies. 3, 4

Mount Policy Specifies the mount criteria and limitations that trigger a tape mount. 3

SSS Simple Shared Secret. 12

Storage Class Specifies how many tape copies an archive file is expected to have. 3

Tape Pool A logical grouping of tapes. Each tape belongs to exactly one tape pool. Tape pools are used to keep data belonging to different VOs separate, categorise types of data and to separate multiple copies of files so that they are physically stored in different buildings. 3, 4, 8

User An EOS user which triggers the archiving/retrieving of a file to/from tape. 3, 4

Virtual Organisation (VO) Refers to CERN experiments (ALICE, ATLAS, CMS, LHCb) or other groups which require separate data storage, such as the International Linear Collider (ILC). 3

Chapter 1

Introduction

The main objective of the CERN Tape Archive (CTA) project is to develop a prototype tape archive system that transfers files directly between remote disk storage systems and tape drives. The concrete remote storage system of choice is EOS.

The Data and Storage Services (DSS) currently provides a tape archive service. This service is implemented by the Hierarchical Storage Management (HSM) system named the CERN Advanced STORage Manager (CASTOR). This HSM has an internal disk-based storage area that acts as a staging area for tape drives. Until now this staging area has been a vital component of CASTOR. It has provided the necessary buffer between the multi-stream, block-oriented disk drives of end users and the single-stream, file-oriented tape drives of the central tape system. Assuming the absence of a sophisticated disk to tape scheduling system, at any single point in time a disk drive will be required to service multiple data streams whereas a tape drive will only ever have to handle a single stream. This means that a tape stream will be at least one order of magnitude faster than a disk stream. With the advent of disk storage solutions that stripe single files over multiple disk servers, the need for a tape archive system to have an internal disk-based staging area has become redundant. Having a file striped over multiple disk servers means that all of these disk-servers can be used in parallel to transfer that file to a tape drive, hence using multiple disk-drive streams to service a single tape stream.

The CTA project is a prototype for a very good reason. The DSS group needs to investigate and learn what it means to provide a tape archive service that does not have its own internal disk-based staging area. The project also needs to keep its options open in order to give the DSS group the best opportunities to identify the best ways forward for reducing application complexity, easing code maintenance, reducing operation overheads and improving tape efficiency.

The CTA project currently has no constraints that go against collecting a global view of all tape, drive and user request states. This means the CTA project should be able to implement intuitive and effective tape scheduling policies. For example it should be possible to schedule a tape archive mount at the point in time when there is both a free drive and a free tape. The architecture of the CASTOR system does not facilitate such simple solutions due to its history of having separate staging areas per experiment and dividing the mount scheduling problem between these separate staging areas and the central tape system responsible for issuing tape mount requests for all experiments.

Chapter 2

CTA Basic Concepts

CTA is operated by authorized administrators (AdminUsers) who issue CTA commands from authorized machines (AdminHosts), using the CTA command line interface. All administrative metadata (such as tape, tape pools, storage classes, etc.) is tagged with a `creationLog` and a `lastModificationLog` which say who/when/where created them and last modified them. An administrator may create (`add`), delete (`rm`), change (`ch`) or list (`ls`) any of the administrative metadata.

Tape Pools are logical groupings of tapes that are used by operators to separate data belonging to different Virtual Organisations (VOs). They are also used to categorize types of data and to separate multiple copies of files so that they end up in different buildings. Each tape belongs to one and only one tape pool.

Logical Libraries are the concept that is used to link tapes and drives together. We use logical libraries to specify which tapes are mountable into which drives, and normally this mountability criteria is based on location, that is the tape has to be in the same physical library as the drive, and on read/write compatibility. Each tape and each drive has one and only one logical library.

A Storage Class is assigned to each archive file to specify how many tape copies the file is expected to have.

Archive Routes link storage classes to tape pools. An archive route specifies onto which set of tapes the copies will be written. There is an archive route for each copy in each storage class, and normally there should be a single archive route per tape pool.

So to summarize, an archive file has a storage class that determines how many copies on tape that file should have. A storage class has an archive route per tape copy to specify into which tape pool each copy goes. Each tape tool is made of a disjoint set of tapes. And tapes can be mounted on drives that are in their same logical library.

2.1 Archiving a file with CTA

CTA has a CLI for archiving and retrieving files to/from tape, that is meant to be used by an external disk-based storage system with an archiving workflow engine such as EOS. A non-administrative User in CTA is an EOS user which triggers the need for archiving or retrieving a file to/from tape. A User normally belongs to a specific CTA Mount Group which specifies the Mount Policy.

Here we offer a simplified description of the archive process:

1. EOS issues an archive command for a specific file, providing its source path, its Storage Class and the User requesting the archival.
2. CTA returns immediately an `ArchiveFileID` which is used by CTA to uniquely identify files archived on tape. This ID will be kept by EOS for any operations on this file (such as retrieval).

3. Asynchronosly, CTA carries out the archival of the file to tape, in the following steps:
   (a) CTA looks up the Storage Class provided by EOS and makes sure it has correct Archive Routes to one or more Tape Pools (more than one when multiple copies are required by the Storage Class).
   (b) CTA queues the corresponding archive job(s) to the proper Tape Pool(s).
   (c) in the meantime each free tape drive queries the central "scheduler" for work to be done, by communicating its name and its Logical Library.
   (d) for each work request, CTA checks whether there is a free tape in the required Tape Pool (as specified in 3b), that belongs to the desired Logical Library (specified in 3c).
   (e) if that is the case, CTA checks whether the work queued for that Tape Pool is worth a mount, i.e. if it meets the archive criteria specified in the Mount Group to which the User (specified in 1) belongs.
   (f) if that is the case, the tape is mounted in the drive and the file gets written from the source path (specified in 1) to the tape.
   (g) after a successful archival, CTA notifies EOS through an asynchronous callback.

An archival process can be canceled at any moment (even after correct archival, but in this case it's a `delete`) through the `delete archive` command.

2.2 Retrieving a file with CTA

Here we offer a simplified description of the retrieve process:

1. EOS issues a retrieve command for a specific file, providing its `ArchiveFileID`, desired destination path and the User requesting the retrieval.
2. CTA returns immediately.
3. Asynchronosly, CTA carries out the retrieval of the file from tape, in the following steps:
   (a) CTA queues the corresponding retrieve job(s) to the proper tape(s) (depending on where the tape copies are located).
   (b) in the meantime each free tape drive queries the central "scheduler" for work to be done, by communicating its name and its Logical Library.
   (c) for each work request CTA checks whether the Logical Library (specified in 3b) is the same of (one of) the tape(s) (specified in 3a).
   (d) if that is the case, CTA checks whether the work queued for that tape is worth the mount, i.e. if it meets the retrieve criteria specified in the Mount Group to which the User (specified in 1) belongs
   (e) if that is the case, the tape is mounted in the drive and the file gets read from tape to the destination (specified in 1).
   (f) after a successful retrieval CTA notifies EOS through an asynchronous callback.

A retrieval process can be canceled at any moment prior to correct retrieval through the "cancel retrieve" command.

Chapter 3

Tape Sessions and Sub-processes

3.1 Introduction

The program `cta-taped` is a daemon managing the tape drive and transferring data from tape to drive. The daemon has two levels of processes:

The Daemon Process: a single threaded sub-process manager which does not have any external connectivity. The Daemon Process is very simple and has a long lifetime (in the order of months).

Sub-processes: implement external connectivity. Sub-processes can be multi-threaded. They have a short lifetime with regular restarts, to limit the consequences of memory leaks or other potential bugs in third-party tape libraries.

There are several types of Sub-process. The main Sub-process is the drive sub-process (see below).

Other possible sub-processes:

- Labelling process?
- Drive cleaning process?
- Verification process? Perhaps not necessary as a read-only Drive process could do the job?

Are these part of the Drive sub-process or are they separate processes launched by the Drive sub-process?

3.2 Drive sub-process

One Drive sub-process is launched per drive in the tape server. The Drive sub-process executes one mount and then exits.

The daemon then restarts a new Drive sub-process instance, passing in the previous instances' exit status. Based on this status from the previous run, the session could become either:

A Cleanup Session: any potentially still-mounted tape is removed from the drive, or

A Scheduling Session: Scheduling can lead to Archive, Retrieve or Labelling sessions.

The tape session types and state changes are shown in Figure 3.1.

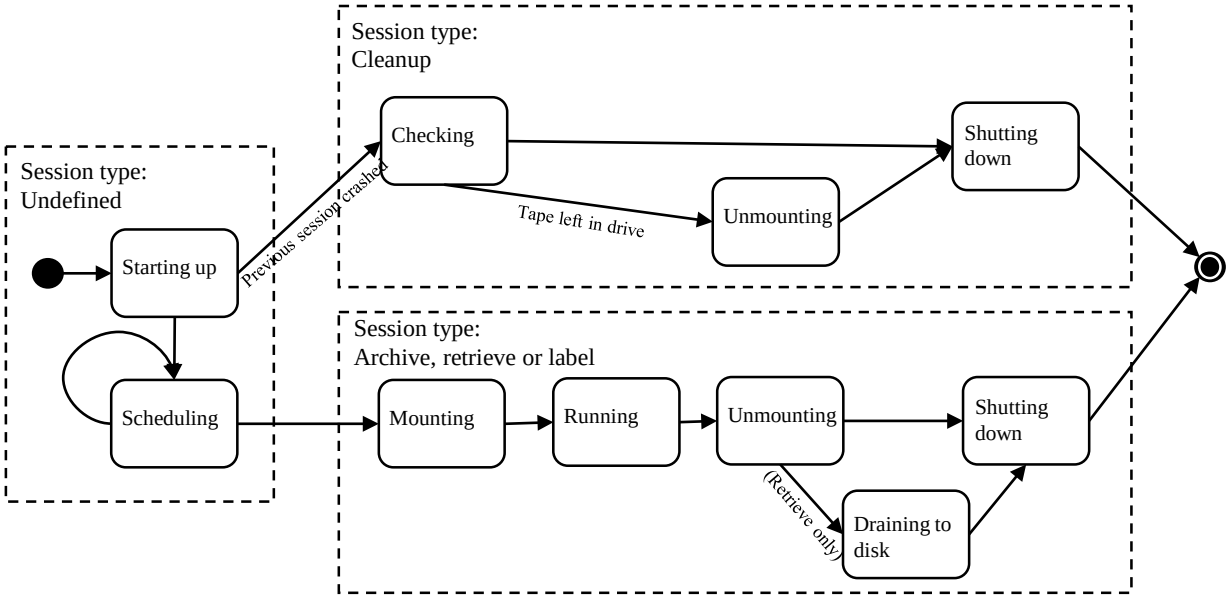

Figure 3.1: Tape sessions state diagram

Chapter 4

Object Store

4.1 Introduction

The queuing system of CTA is implemented over an Object Store. This is preferred over databases that do not provide a good modeling of multiple independent queues and objects. Databases also struggle to shrink tables that once contained lots of entries, which is the fate of a queue. Classical databases are also a single point of failure and contention, and regularly require downtime for software maintenance.

The targeted implementation is Ceph, which scales horizontally and provides parallel access to objects. A Ceph cluster also provides excellent resilience against component failures.

The CTA Scheduler object relies on a SchedulerDatabase object to store the queuing related information.

The techniques employed in the Object Store have several aspects:

1. The in-memory representation of individual objects and the functions used to serialize and de-serialize data between memory and Object Store.
2. The connection of the objects together to constitute a multi-object structure. As the Object Store only provides per-object transactions, safe multi-object operations require usage of a few conventions.
3. Finally, a garbage collector allows resetting objects left behind by crashed processes, by requeuing requests and deleting uncommitted objects.

4.2 Classes and memory side representation

The processes of CTA (namely user front end and tape drive) rely on a shared Scheduler object to queue, dequeue and report about data transfer requests. The Scheduler itself relies on an Object Store-based SchedulerDatabase for queuing, and a file Catalogue to keep persistent information about stored files.

The OStoreDB implementation of the SchedulerDatabase interface relies on a collection of classes in the cta::objectstore namespace. Those classes are responsible for providing the high level functionality specific to each object type, on top of the common methods provided by all objects (lock, fetch, commit, etc.). The common part is inherited from the template ObjectOps. The parameter to this template is the Google protocol buffer type used to serialize the content of the object to persistent storage. The commonalities of all the template instances are inherited from a base class ObjectOpsBase. This base class in used for special operations that can apply to any object type, namely garbage collection. The memory side class hierarchy is shown in figure 4.1.

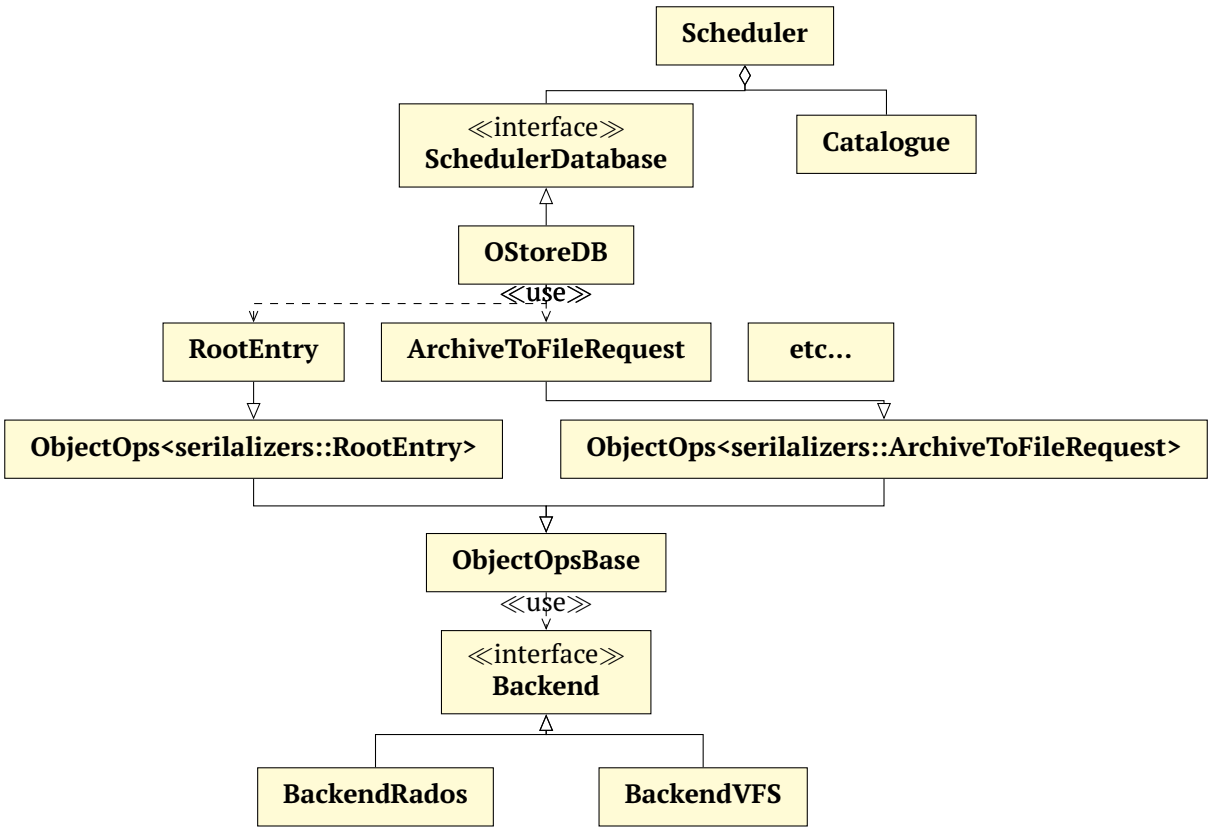

Figure 4.1: Object store's classes

4.3 Data model and Object Store side representation

To achieve even performance with various amounts of requests queued, the implementation will store the requests into queues, one per Tape Pool for archival and one per tape for retrieves. The targeted queuing and dequeuing complexity is $\mathcal{O}(1)$, but higher order complexity is necessary for retrieve queues, where requests are stored in tape location order and not arrival order.

The Object Store contains one queue per tape pool for archival, one queue per tape for retrieval. The status of the drives is also stored, with which tape they are working on. A singleton object is used as a lock, as the mount scheduling is executed one drive at a time. The combination of how much is queued for each tape and tape pool, plus what is currently being done by other drives is used to determine the next mount for the drive being scheduled.

Finally each actor on the Object Store is represented as a Agent object, which keeps references to objects created and worked on by the actor, preventing object leak. The data model of the Object Store is shown in figure 4.2.

4.3.1 RootEntry

The RootEntry is an object with a conventional name in the Object Store. It is the entry point to the object tree, and is the only object which does not require a lookup. It contains references to each queue, the drive register, the agent register and the scheduling lock. It only needs to be modified when a new queue (archive or retrieve) is created or removed.

4.3.2 Queues and request objects

Requests represent a full file request. An archive request is hence composed of one or several transfers — one for each copy, and all of them should be executed. A retrieve request is also composed of one or several transfers, but only one of them needs to be executed in order for the file to be retrieved.

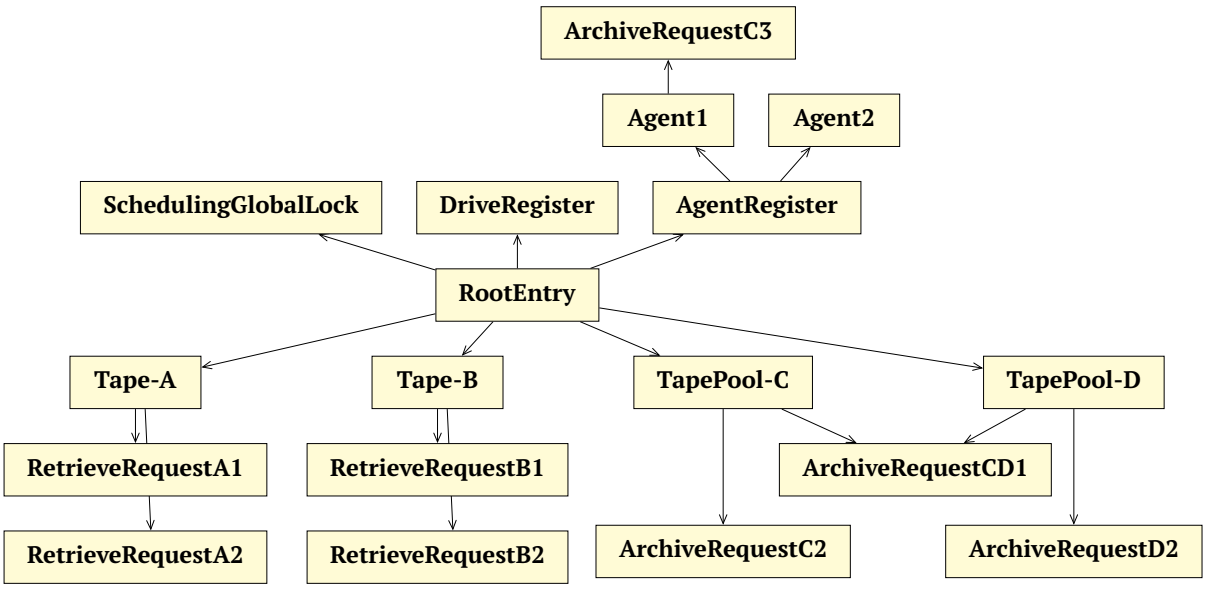

Figure 4.2: Object store's instance diagram

The archive request

The archive requests is a set of one or several transfer jobs (one per copy on tape) for a given tape file. All should be executed. The archive requests has a life cycle deriving from the ones of the individual jobs. Typically, when marking the last job as finished, the request becomes complete as well. In order to simplify this updating, all the related jobs are physically stored in a single object, the archive request. The archive queues hence point not only to the archive request object, but also to the job number within the request. Impact on the multi object operations is described in section 4.4.2. For multi-copies objects, this means that a given archive request object will be queued on several tape pools simultaneously, while in practice each job will be attached to a different queue.

The retrieve request

The retrieve requests is a set of one or several transfer jobs (one per copy on tape) for a given tape file. Only one of them needs to be executed. The requests will hence be queued to only one tape at a time. At queuing time, we decide which tape is the most promising (with the most work already queued) and add the request to this one, minimizing the number of mount and increasing the chances of reaching the mount policy thresholds. As only one job is active at any point in time, the retrieve request has a single owner like the rest of object, the archive request being the only exception.

4.3.3 Archive and retrieve queues

One archive queue is created per tape pool, one retrieve queue per tape with existing requests. They contain references to archive jobs, pre-ordered by age time. This allows $\mathcal{O}(1)$ de-queuing in all cases, $\mathcal{O}(1)$ insertion for archivals, and $\mathcal{O}(\log(n))$ insertion for retrievals (they have to be sorted in tape position order). Re-queing (insertion) failed requests for retries will require $\mathcal{O}(1)$ or $\mathcal{O}(\log(n))$ depending on the policy and the direction. The initially intended policy is to re-queue archive requests at the end of the queue to guarantee global system performance.

4.3.4 Drive register, scheduling global lock and agent register

The drive register will allow operators and other drives (when scheduling) to get a picture of the whole system. Each drive schedules itself when idle, and needs to know how much is currently queued, with which age and which priority and what other drives are working on to reach a decision matching mount criteria. This includes which tape is being worked on by the drives, and the states of the drives.

This information is one way, from drive to reader, except for the operator changing the state of drive (DOWN to UP, and vice-versa, when applicable). The state of the drive is time tagged to detect stale drive information for non-running servers.

The scheduling global lock is a object used for locking the system globally while a drive is deciding its next mount. This is discussed further in section 4.5.

The agent register is a list of all the agents operating on the Object Store. The list points to individual agent objects, one per actual process running in the system. This is further discussed in 4.4.1.

4.4 Multi object operations and multi-agent safety

The Object Store provided per-object locking. The ObjectOps base template will validate that proper locking has been taken on a given object before accessing it. The usual sequences are { initialise (in memory), modify in memory, insert new object in the store}, { lock, fetch, modify in memory, commit } and { lock, fetch, remove }.

When a multi-object structure is involved, the process accessing the store should manage to create the object and reference it a way that is semantically atomic for the other processes. This multi-object access is implemented in the OStoreDB object.

To achieve semantic atomicity on multi-object operations, two conventions are used.

The first convention is that references to object can be stale. This allows several references to exist at any point in time, pointing to the same object, with only one being effective (or zero before object creation). References can also point to non-existing objects. The function handling the reference should manage those cases.

The second convention is that objects point to their actual reference, allowing to resolve if a reference being used if active or stale.

During object creation or processing (like when a job is selected by the tape server for being executed), the object is referenced by the agent structure representing the current process.

4.4.1 Agent failure management and garbage collection

The conventions previously describe ensure that objects are always uniquely referenced inside the object tree, either by a queue or by an agent. Several instances of a dedicated process, the garbage collector, monitor those agent entries. The agent entry contains a heartbeat counter, which allows the garbage collector to determine that the process is not active anymore, and triggers the resetting of the owned objects. Garbage collector processes themselves are also represented as agents, own other agents (they cannot watch themselves) so that the crash of a garbage collector is also covered (the watched agents will be picked up by another garbage collector instance, on another system, or at another time as the garbage collector will be restarted automatically).

The resetting of the objects is type dependent. Each in memory object type implements a garbage collect method, which is called by the garbage collector when collecting a dead process. The Object Store representation of objects has a common header indicating the type, schema version number and owner (which is a shared notion). This allow the garbage collector to determine the type dynamically and to call the appropriate garbage collection function. Likewise, the owner in the header allows determining whether the object is actually owner by the agent being garbage collected (in which case the object should be reset), or not (in which case the reference was actually stale).

4.4.2 Special case of archive and retrieve requests ownership

As mentioned in section 4.3.2, the archive request is a special case, and has several owners, one per tape copy job. This means that determining ownership will require actually parsing object content

itself instead of just the header. Besides this detail, the re-queuing of the job is identical to the other cases.

4.4.3 Object versioning an schema evolution

The object version, not currently used is intended for live schema evolution. In order to achieve migration from version A to B of the schema, we need to implement a transtional version of the objects which can read and write version A and B. After global deployment of this version, a central trigger (configuration file, etc.) changes the write version of the instances from A to B, and all objects previously written with schema A will be written back with schema B on the next update. This method allows a zero downtime schema transition, with the drawback that an active traversal of the structure is necessary to ensure complete transition. The schema is not yet implemented.

4.5 Performance considerations

Performance numbers have been extracted from the CASTOR runs of 2015. The per tape pool rate has been measured over 10 minutes intervals. The maximum seen was $78\ \mathrm{Hz}$. The initial performance target will hence be $100\ \mathrm{Hz}$ per queue and a total $1\ \mathrm{kHz}$ system wide. The maximum size for a queue will be 10^7, and the system will instruct the user to back-off before crossing this boundary. This limit represents more than a day's worth at the maximum rate. The number of queues existing at a single point in time is estimated to be around 10^3 (as several hundreds can be typically seen in CASTOR).

Using an Object Store allows independent access to each object, so little contention is expected, besides when accessing queues. As there are one queue per tape pool, cross talk between users of different tape pools should be minimal. The main challenge will hence be to ensure efficient queuing in a given queue when many files get added/dequeued in parallel. As the round trip time to the Object Store will not be compressible, we will have to add many elements to the queue in one go. On the tape server side, this could be implemented with bulk access to the queue, followed by many threads updating the jobs in parallel, and then updating all the entries in one go in the queue. This would allow accessing an arbitrary amount of jobs over a fixed number of round trip times.

On the front end side, the fact that each xrootd connection lives in a separate thread can be leveraged, by naturally creating the jobs in each thread, and then relying on shared data structures to accumulate elements to queue in one go. This will allow to increase throughput at the expense of an increased (but bound) latency to the end user.

Chapter 5

CTA Authorization

5.1 Simple Shared Secrets (SSSs)

SSSs are used to authenticate communications using the XRoot protocol, which is the case in the following situations:

1. Internal communication between the EOS `mgm` and `fst` daemons.
2. Communication between the Tape Server and the EOS `mgm` daemon. (On the other hand, communication between the Tape Server and the EOS `fst` daemon does not use SSS; this is handled by internal redirection within the XRoot library layer.)
3. Communication between the EOS `mgm` daemon and the CTA Front End daemon.

5.2 Kerberos

Kerberos authentication is used in the following situations:

1. Communication between the CTA Admin tool and the CTA Front End daemon. In this case, Kerberos is the only available authentication mechanism.
2. Communication between EOS users (Atlas, CMS, etc.) and the EOS `mgm` daemon. In this case, Kerberos is one of several options. Authentication can be performed by any mechanism which is supported by both XRoot and EOS, for example SSS or standard UNIX authentication.

Appendix A

Tape Format

A.1 Overview

CTA uses the same AUL file format as CASTOR[1]. This format is based on ANSI INCITS 27-1987 and is described in detail on the Tape Labels, ANSI and IBM web page (last updated in 2008).

The AUL format has the following descriptors:

- Volume Label (VOL1)
- Header Blocks: Headers (HDR1, HDR2) and User Header Labels (UHL1)
- Trailer Blocks: User Trailer Labels (UTL1)[2]

Each of these descriptor labels is contained in an 80-byte tape block of ASCII text. Empty bytes are stored as spaces (0x20). The label descriptor must begin with the 4-byte identifier. Labels are terminated by a file mark: Tape Mark (TM) or End of File (EOF).

Table A.1: AUL label format

VOL1	HDR1	HDR2	UHL1	TM	DATA	TM	EOF1	EOF2	UTL1	TM
	one data file									

Volumes that have just been initialised contain no data records, just a single 'header label group':

Table A.2: AUL prelabeled tape with one HDR1

VOL1	HDR1(PRELABEL)	TM

A.2 Volume Label (VOLn)

The very first label record on a labelled volume is VOL1. If this label is incorrect, you will not advance at all.

The format is shown in Table A.3. Example for beginning of the tape:

```
00000000  56 4f 4c 31 56 35 32 30  30 31 20 20 20 20 20 20  |VOL1V52001      |
00000010  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                |
00000020  20 20 20 20 20 43 41 53  54 4f 52 20 20 20 20 20  |     CASTOR     |
00000030  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                |
00000040  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 33  |               3|
```

[1] CASTOR used several file formats over time, but by 2013, only the AUL format was in use.

[2] The UHLs and UTLs are defined in ANSI X 3.27. The general description of the ANSI fields was documented in IBM's z/OS documentation.

Table A.3: The structure of the volume label

VOL1			
Bytes	Length	Offset	Content
0-3	4	0x00	Volume label indicator: the characters **VOL1**
4-9	6	0x04	Volume serial number (VSN) (e.g., "AB1234")
10	1	0x0A	Accessibility (left as empty space)
11-23	13	0x0B	Reserved (spaces)
24-36	13	0x18	Implementation identifier (left as empty spaces)
37-50	14	0x25	Owner identifier (the string "CASTOR" or STAGESUPERUSER name, padded with spaces)
51-78	28	0x33	Reserved (spaces)
79	1	0x4F	Label standard level (1, 3 and 4 are listed as valid in IBM's documentation. CASTOR uses ASCII '3')

A.3 Header Label (HDRn)

HDR1 and HDR2 are normally found together at the beginning of a dataset.

The format for HDR1 is shown in Table A.4 and the format for HDR2 is shown in Table A.5. Example for the empty tape with PRELABEL and one HDR1:

```
00000000  56 4f 4c 31 56 35 32 30  30 31 20 20 20 20 20 20  |VOL1V52001      |
00000010  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                 |
00000020  20 20 20 20 20 72 6f 6f  74 20 20 20 20 20 20 20  |     root        |
00000030  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                 |
00000040  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 33  |               3|
00000050  48 44 52 31 50 52 45 4c  41 42 45 4c 20 20 20 20  |HDR1PRELABEL    |
00000060  20 20 20 20 20 56 35 32  30 30 31 30 30 30 31 30  |     V5200100010|
00000070  30 30 31 30 30 30 31 30  30 30 31 33 32 33 34 30  |0010001000132340|
00000080  31 33 32 33 34 20 30 30  30 30 30 30 43 41 53 54  |13234 000000CAST|
00000090  4f 52 20 32 2e 31 2e 31  33 20 20 20 20 20 20 20  |OR 2.1.13       |
```

Example of HDR1 for the second file on the tape:

```
00000000  48 44 52 31 31 32 41 31  36 30 43 33 38 20 20 20  |HDR112A160C38   |
00000010  20 20 20 20 20 56 35 32  30 30 31 30 30 30 31 30  |     V5200100010|
00000020  30 30 32 30 30 30 31 30  30 30 31 32 30 34 31 30  |0020001000120410|
00000030  31 32 30 34 31 20 30 30  30 30 30 30 43 41 53 54  |12041 000000CAST|
00000040  4f 52 20 32 2e 31 2e 31  32 20 20 20 20 20 20 20  |OR 2.1.12       |
```

Example of HDR2 for the first file on the tape:

```
00000000  48 44 52 32 46 30 30 30  30 30 30 30 30 30 30 20  |HDR2F0000000000 |
00000010  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                 |
00000010  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                 |
00000030  20 20 30 30 20 20 20 20  20 20 20 20 20 20 20 20  |  00             |
00000040  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                 |
```

A.4 User Header Label (UHLn)

The format for UHL1 is shown in Table A.6. Example for the second file on the tape:

```
00000000  55 48 4c 31 30 30 30 30  30 30 30 30 30 32 30 30  |UHL1000000000200|
00000010  30 30 32 36 32 31 34 34  30 30 30 30 32 36 32 31  |0026214400002621|
00000020  34 34 43 45 52 4e 20 20  20 20 4c 58 43 32 44 45  |44CERN    LXC2DE|
00000030  56 35 44 32 53 54 4b 20  20 20 20 20 54 31 30 30  |V5D2STK     T100|
00000040  30 30 42 20 58 59 5a 5a  59 5f 42 31 20 20 20 20  |00B XYZZY_B1    |
```

Table A.4: The structure of the HDR1, EOF1 labels

HDR1, EOF1			
Bytes	Length	Offset	Content
0-3	4	0x00	Header label: the characters “HDR1 or EOF1”
4-20	17	0x04	File identifier: hexadecimal CASTOR NS file ID. `nsgetpath -x` can be used to find the CASTOR full path name. Aligned to left. In case of prelabeled tape ‘PRELABEL’ is used instead of file ID.
21-26	6	0x15	The volume serial number of the tape.
27-30	4	0x1B	File section number: a number (0001 to 9999) that indicates the order of the volume within the multivolume aggregate. This number is always 0001 for a single volume data set.
31-34	4	0x1F	File sequence number: a number that indicates the relative position of the data set within a multiple data set group (aggregate). CASTOR uses modulus for fseq by 10000
35-38	4	0x23	Generation number: ‘0001’ in CASTOR.
39-40	2	0x27	Version number of generation: ‘00’ in CASTOR.
41-46	6	0x29	Creation date: Date when allocation begins for creating the data set. The date format is cyyddd, where: c = century (blank=19; 0=20; 1=21; etc.) yy = year (00-99) ddd = day (001-366)
47-52	6	0x2F	Expiration date: year and day of the year when the data set may be scratched or overwritten. The data is shown in the format cyyddd. It is always advisable to set the expiration date when a volume is being initialised (‘prelabelled’) to be a date before the current date, so that writing to the tape is immediately possible.
53	1	0x35	Accessibility: a code indicating the security status of the data set and ‘space’ means no data set access protection.
54-60	6	0x36	Block count: This field in the trailer label shows the number of data blocks in the data set on the current volume. This field in the header label is always ‘000000’.
60-72	13	0x3C	System code of creating system: a unique code that identifies the system. CASTOR with CASTOR BASEVERSION number string.
73-79	7	0x49	Reserved

A.5 Data Records

After a ‘header label group’, data records follow of any length and in any number. Eventually, an EOF will appear and then a ‘trailer label group’ is expected.

The data block size is configurable but in practice a block size of 256 KiB has been used everywhere.

A.6 End of File (EOFn)

EOF1 and EOF2 are normally found together at the end of a dataset.

Note that an End of Volume (EOVn) label will appear instead of EOFn if this is the final label group on the volume, but the dataset continues on another volume. EOV1 and EOV2 are only expected together and at the end of a volume.

A.7 User Trailer Label (UTLn)

The format for UTL1 is the same as UHL1 (Table A.6).

Table A.5: The structure of the HDR2, EOF2 labels

HDR2, EOF2			
Bytes	Length	Offset	Content
0-3	4	0x00	Header label: the characters “HDR2 or EOF2”
4	1	0x04	Record format. An alphabetic character that indicates the format of the records in the associated data set. For the AUL it could be only: F - fixed length (U - was used for HDR2 for prelabeled tapes)
5-9	5	0x05	Block length in bytes (maximum). For the block size greater than 100000 the value is 00000.
10-14	5	0x0A	Record length in bytes (maximum). For the record size greater than 100000 the value is 00000.
15	1	0x0F	Tape density. Depends on the tape density values are following: ‘2’ for D800, ‘3’ for D1600, ‘4’ for D6250
16-33	18	0x10	Reserved
34	2	0x22	Tape recording technique. The only technique available for 9-track tape is odd parity with no translation. For a magnetic tape subsystem with Improved Data Recording Capability, the values are: ‘P ’- Record data in compacted format, ‘ ’ - Record data in standard uncompacted format. For CASTOR is is ‘P’ if the drive configured to use compression (i.e. xxxGC)
35-49	14	0x24	Reserved
50-51	2	0x32	Buffer offset ‘00’ for AL and AUL tapes
52-79	28	0x34	Reserved

Table A.6: The structure of the UHL1, UTL1 labels

UHL1, UTL1			
Bytes	Length	Offset	Content
0-3	4	0x00	User header label: the characters “UHL1 or UTL1”.
4-13	10	0x04	Actual file sequence number (‘0’ padded from left).
14-23	10	0x0E	Actual block size (‘0’ padded from left).
24-33	10	0x18	Actual record length (‘0’ padded from left).
34-41	8	0x22	Site : a part of the domain name uppercase.
42-51	10	0x2A	Tape mover host name uppercase without domain name.
52-59	8	0x34	Drive manufacturer.
60-67	8	0x3C	Drive model (first 8 bytes from the field PRODUCT IDENTIFICATION in the SCSI INQUIRY replay).
68-79	12	0x44	Drive serial number.

A.8 Checksums

When a file is written to tape, an Adler32 checksum is computed on the file. The main advantages of Adler32 are that it is faster to compute than CRC32 or MD5, and it is distributive when computing the checksum for a multi-block file. This checksum is not stored on the tape; it is stored as metadata in the Catalogue.

Note: The tape drives also compute a CRC32 checksum on each block, which is checked in firmware. This checksum is not seen by the software.