The CERN Tape Archive

Release 0

Generated by Doxygen 1.8.5

Mon Apr 10 2017 11:26:16

Contents

1 Introduction

Author

German Cancio, Eric Cano, Daniele Kruse and Steven Murray

The main objective of the 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) group of the CERN IT department 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 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.

2 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 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.

The storage class is what we assign 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.

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 criteria and limitations (together called "mount policy") that trigger a tape mount. Here we offer a simplified description of the archive process:

{enumerate} EOS issues an archive command for a specific file, providing its source path, its storage class (see above), and the user requesting the archival 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) Asynchronosly, CTA carries out the archival of the file to tape, in the following steps: {itemize} CTA looks up the storage class provided by EOS and makes sure it has correct routings to one or more tape pools (more than one when multiple copies are required by the storage class) CTA queues the corresponding archive job(s) to the proper tape pool(s) in the meantime each free tape drive queries the central "scheduler" for work to be done, by communicating its name and its logical library for each work request CTA checks whether there is a free tape in the required pool (as specified in b.), that belongs to the desired logical library (as specified in c.) 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 (as specified in 1.) belongs 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 after a successful archival CTA notifies EOS through an asynchronous callback {itemize} {enumerate}

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

Here we offer a simplified description of the retrieve process:

{enumerate} EOS issues a retrieve command for a specific file, providing its ArchiveFileID and desired destination path, and the user requesting the retrieval CTA returns immediately Asynchronosly, CTA carries out the retrieval of the file from tape, in the following steps: {itemize} CTA queues the corresponding retrieve job(s) to the proper tape(s) (depending on where the tape copies are located) in the meantime each free tape drive queries the central "scheduler" for work to be done, by communicating its name and its logical library for each work request CTA checks whether the logical library (as specified in b.) is the same of (one of) the tape(s) (as specified in a.) 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 (as specified in 1.) belongs 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. after a successful retrieval CTA notifies EOS through an asynchronous callback {itemize} {enumerate}

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

3 Authorization Guidelines

One of the requirements of CTA is to limit the crosstalk among different EOS instances. In more detail:

{enumerate} A listStorageClass command should return the list of storage classes belonging to the instance from where the command was executed only

A queueArchive command should be authorized only if: {itemize} 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 {itemize}

A queueRetrieve command should be authorized only if: {itemize} 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 {itemize}

A deleteArchive command should be authorized only if: {itemize} 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 {itemize}

A cancelRetrieve command should be authorized only if: {itemize} 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 {itemize}

An updateFileStorageClass command should be authorized only if: {itemize} 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 {itemize}

An updateFileInfo command should be authorized only if: {itemize} the instance of the file to be updated coincides with the instance from where the command was executed {itemize} {enumerate}

4 Reconciliation Strategy

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).

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.

EOS communicates with CTA by issuing commands on trusted hosts. EOS can archive a file, retrieve it, update its information/storage class, delete it or simply list the available storage classes. See the LimitingInstanceCrosstalk.txt file for more details on how these commands are authorized by CTA.

{verbatim} 1) EOS REQUEST: cta a/archive

```
--encoded <"true" or "false">      // true if all following arguments are base64 encoded,
```

false if all following arguments are in clear (no mixing of encoded and clear arguments)

```
--user <user>                      // string name of the requester of the action (archival),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--group <group>                    // string group of the requester of the action (archival),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--diskid <disk_id>                 // string disk id of the file to be archived,
```

kept by CTA for reconciliation purposes

```
--instance <instance>              // string kept by CTA for authorizing the request
```

and for disaster recovery

```
--srcurl <src_URL>                 // string source URL of the file to archive of
```

the form scheme://host:port/opaque_part, not kept by CTA after successful archival

```
--size <size>                      // uint64_t size in bytes kept by CTA for
```

correct archival and disaster recovery

```
--checksumtype <checksum_type>     // string checksum type (ex. ADLER32) kept by CTA
```

for correct archival and disaster recovery

```
--checksumvalue <checksum_value>   // string checksum value kept by CTA for correct
```

archival and disaster recovery

```
--storageclass <storage_class>     // string that determines how many copies and
```

which tape pools will be used for archival kept by CTA for routing and authorization

```
--diskfilepath <disk_filepath>     // string the disk logical path kept by CTA
```

for disaster recovery and for logging

```
--diskfileowner <disk_fileowner>   // string owner username kept by CTA
```

for disaster recovery and for logging

```
--diskfilegroup <disk_filegroup>   // string owner group kept by CTA
```

for disaster recovery and for logging

```
--recoveryblob <recovery_blob>     // 2KB string kept by CTA for disaster recovery
```

(opaque string controlled by EOS)

```
--diskpool <diskpool_name>         // string used (and possibly kept)
```

by CTA for proper drive allocation

```
--throughput <diskpool_throughput> // uint64_t (in bytes) used (and possibly kept)
```

by CTA for proper drive allocation

2) CTA IMMEDIATE REPLY: CTA_ArchiveFileID or Error

CTA_ArchiveFileID: string which is the unique ID of the CTA file to be kept by EOS while file exists (for future retrievals). In case of retries, a new ID will be given by CTA (as if it was a new file), the old one can be discarded by EOS.

3) CTA CALLBACK WHEN ARCHIVED SUCCESSFULLY: src_URL and copy_number with or without Error

```
      src_URL: this is the same string provided in the EOS archival request
  copy_number: indicates which copy number was archived
         note: if multiple copies are archived there will be one callback per copy
```

{verbatim}

{verbatim} 1) EOS REQUEST: cta r/retrieve

```
--encoded <"true" or "false">      // true if all following arguments are base64 encoded,
```

false if all following arguments are in clear (no mixing of encoded and clear arguments)

```
--user <user>                      // string name of the requester of the action (retrieval),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--group <group>                    // string group of the requester of the action (retrieval),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--id <CTA_ArchiveFileID>           // uint64_t which is the unique ID of the CTA file

--dsturl <dst_URL>                 // string of the form scheme://host:port/opaque_part,
```

not kept by CTA after successful operation

```
--diskfilepath <disk_filepath>     // string the disk logical path kept by CTA
```

for disaster recovery and for logging

```
--diskfileowner <disk_fileowner>   // string owner username kept by CTA for
```

disaster recovery and for logging

```
--diskfilegroup <disk_filegroup>   // string owner group kept by CTA for disaster
```

recovery and for logging

```
--recoveryblob <recovery_blob>     // 2KB string kept by CTA for disaster recovery
```

(opaque string controlled by EOS)

```
--diskpool <diskpool_name>         // string used (and possibly kept) by CTA for
```

proper drive allocation

```
--throughput <diskpool_throughput> // uint64_t (in bytes) used (and possibly kept)
```

by CTA for proper drive allocation

Note: disk info is piggybacked

2) CTA IMMEDIATE REPLY: Empty or Error
3) CTA CALLBACK WHEN RETRIEVED SUCCESSFULLY: dst_URL with or without Error

```
      dst_URL: this is the same string provided in the EOS retrieval request
```

{verbatim}
{verbatim} 1) EOS REQUEST: cta da/deletearchive

```
--encoded <"true" or "false">      // true if all following arguments are base64 encoded,
```

false if all following arguments are in clear (no mixing of encoded and clear arguments)

```
--user <user>                      // string name of the requester of the action (deletion),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--group <group>                    // string group of the requester of the action (deletion),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--id <CTA_ArchiveFileID>           // uint64_t which is the unique ID of the CTA file
```

Note: This command may be issued even before the actual archival process has begun

2) CTA IMMEDIATE REPLY: Empty or Error {verbatim}
{verbatim} 1) EOS REQUEST: cta cr/cancelretrieve

```
--encoded <"true" or "false">      // true if all following arguments are base64 encoded,
```

false if all following arguments are in clear (no mixing of encoded and clear arguments)

```
--user <user>                      // string name of the requester of the action (cancel),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--group <group>                    // string group of the requester of the action (cancel),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--id <CTA_ArchiveFileID>           // uint64_t which is the unique ID of the CTA file
--dsturl <dst_URL>                 // this is the same string provided in the EOS
```

retrieval request

```
--diskfilepath <disk_filepath>     // string the disk logical path kept by CTA for
```

disaster recovery and for logging

```
--diskfileowner <disk_fileowner>   // string owner username kept by CTA for disaster
```

recovery and for logging

```
--diskfilegroup <disk_filegroup>   // string owner group kept by CTA for disaster
```

recovery and for logging

```
--recoveryblob <recovery_blob>     // 2KB string kept by CTA for disaster recovery
```

(opaque string controlled by EOS)

Note: This command will succeed ONLY before the actual retrieval process has begun Note: disk info is piggy-backed

2) CTA IMMEDIATE REPLY: Empty or Error {verbatim}

{verbatim} 1) EOS REQUEST: cta ufsc/updatefilestorageclass

```
--encoded <"true" or "false">      // true if all following arguments are base64 encoded,
```

false if all following arguments are in clear (no mixing of encoded and clear arguments)

```
--user <user>                      // string name of the requester of the action (update),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--group <group>                    // string group of the requester of the action (update),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--id <CTA_ArchiveFileID>           // uint64_t which is the unique ID of the CTA file
--storageclass <storage_class>     // updated storage class which may or may not have
```

a different routing

```
--diskfilepath <disk_filepath>     // string the disk logical path kept by CTA for
```

disaster recovery and for logging

```
--diskfileowner <disk_fileowner>   // string owner username kept by CTA for disaster
```

recovery and for logging

```
--diskfilegroup <disk_filegroup>   // string owner group kept by CTA for disaster
```

recovery and for logging

```
--recoveryblob <recovery_blob>     // 2KB string kept by CTA for disaster recovery
```

(opaque string controlled by EOS)

Note: This command DOES NOT change the number of tape copies! The number will change asynchronously (next repack or "reconciliation"). Note: disk info is piggybacked

2) CTA IMMEDIATE REPLY: Empty or Error {verbatim}

{verbatim} 1) EOS REQUEST: cta ufi/updatefileinfo

```
--encoded <"true" or "false">      // true if all following arguments are base64 encoded,
```

false if all following arguments are in clear (no mixing of encoded and clear arguments)

```
--id <CTA_ArchiveFileID>           // uint64_t which is the unique ID of the CTA file
--diskfilepath <disk_filepath>     // string the disk logical path kept by CTA for
```

disaster recovery and for logging

```
--diskfileowner <disk_fileowner>   // string owner username kept by CTA for disaster
```

recovery and for logging

```
--diskfilegroup <disk_filegroup>   // string owner group kept by CTA for disaster
```

recovery and for logging

```
--recoveryblob <recovery_blob>     // 2KB string kept by CTA for disaster recovery
```

(opaque string controlled by EOS)

Note: This command is not executed on behalf of an EOS user. Instead it is part of a resynchronization process initiated by EOS.

2) CTA IMMEDIATE REPLY: Empty or Error {verbatim}

{verbatim} 1) EOS REQUEST: cta lsc/liststorageclass

```
--encoded <"true" or "false">      // true if all following arguments are base64 encoded,
```

false if all following arguments are in clear (no mixing of encoded and clear arguments)

```
--user <user>                      // string name of the requester of the action (listing),
```

used for SLAs and logging, not kept by CTA after successful operation

```
--group <group>                    // string group of the requester of the action (listing),
```

used for SLAs and logging, not kept by CTA after successful operation

2) CTA IMMEDIATE REPLY: storage class list {verbatim}

5 the prototype up and running

This chapter explains how to install the CTA prototype together with a local EOS instance on a single local development box.

The CTA project requires xroot version 4 or higher. EOS depends on xroot and therefore the EOS version used must also be compatible with xroot version 4 or higher. An example combination of EOS and xroot versions compatible with the CTA project are EOS version 4.0.4 Citrine together with xroot version 4.2.3-1.

For the EOS rpms create the {/etc/yum.repos.d/eos.repo} file with the following contents. {verbatim} [eos-citrine] name=EOS 4.1 Version baseurl=http://dss-ci-repo.web.cern.ch/dss-ci-repo/eos/citrine/tag/el-7/x8 _64/ gpgcheck=0 enabled=1

[eos-citrine-depend] name=EOS 4.1 Version baseurl=http://dss-ci-repo.web.cern.ch/dss-ci-repo/eos/citri _64/ gpgcheck=0 enabled=1 {verbatim}

For the xroot rpms create the {/etc/yum.repos.d/epel.repo} file with the following contents or run {yum install epel-release} {verbatim} [epel] name=Extra Packages for Enterprise Linux 7 - $basearch baseurl=http-://linuxsoft.cern.ch/epel/7/$basearch enabled=1 gpgcheck=1 gpgkey=file:///etc/pki/rpm-gpg/-RPM-GPG-KEY-EPEL-7 {verbatim}

{verbatim} [epel-debuginfo] name=Extra Packages for Enterprise Linux 7 - $basearch - Debug baseurl=http-://linuxsoft.cern.ch/epel/7/$basearch/debug enabled=1 gpgkey=file:///etc/pki/rpm-gpg/-RPM-GPG-KEY-EPEL-7 gpgcheck=1

{verbatim}

{verbatim} [epel-source] name=Extra Packages for Enterprise Linux 7 - $basearch - Source baseurl=http-://linuxsoft.cern.ch/epel/7/SRPMS enabled=0 gpgkey=file:///etc/pki/rpm-gpg/RPM--GPG-KEY-EPEL-7 gpgcheck=1 {verbatim}

Install the rpms using yum. {verbatim} sudo yum install eos-client eos-server xrootd-client xrootd-debuginfo xrootd-server {verbatim}

Here is an example list of succesfully installed EOS and {xrootd} rpms. {verbatim} rpm -qa | egrep 'eos|xrootd' | sort eos-client-4.1.3-1.el7.x86_64 eos-server-4.1.3-1.el7.x86_64 libmicrohttpd-0.9.38-eos.wves.el7.cern.x86_64 xrootd-4.4.0-1.el7.x86_64 xrootd-client-4.4.0-1.el7.x86_64 xrootd-client-devel-4.4.0-1.el7.x86_64 xrootd-client-libs-4.4.0-1.el7.x86_64 xrootd-debuginfo-4.4.0-1.el7.x86_64 xrootd-devel-4.4.0-1.el7.x86_64 xrootd-libs-4.4.0-1.el7.-x86_64 xrootd-private-devel-4.4.0-1.el7.noarch xrootd-selinux-4.4.0-1.el7.noarch xrootd-server-4.4.0-1.el7.x86_64 xrootd-server-devel-4.4.0-1.el7.x86_64 xrootd-server-libs-4.4.0-1.el7.x86_64 {verbatim}

Create the {/etc/syconfig/eos} file based on the example installed by the .{eos-server} rpm: {verbatim} sudo cp /etc/sysconfig/eos.example /etc/sysconfig/eos {verbatim}

Reduce the {xrootd} daemon roles to the bare minimum of just {mq}, {mgm} and {fst}. This means there will be a total of three {xrootd} daemons running for EOS on the local development box. {verbatim} XRD_ROLES="mq mgm fst" {verbatim}

Set the name of the EOS instance, for example. {verbatim} export EOS_INSTANCE_NAME=eoscta {verbatim}

Replace all of the hostnames with the fully qualified hostname of the local development box. The resulting hostname entries should look something like the following, where {devbox.cern.ch} should be replaced with the fully qualified name of the development box where EOS is being installed. {verbatim} export EOS_INSTANCE_NAME=eoscta export EOS_BROKER_URL=root://devbox.cern.ch:1097//eos/ export EOS_MGM_MASTER1=devbox.cern.ch export EOS_MGM_MASTER2=devbox.cern.ch export EOS_MGM_ALIAS=devbox.cern.ch export EOS_FUSE_MGM_ALIAS=devbox.cern.ch export EOS_FED_MANAGER=devbox.cern.ch:1094 export EOS_TEST_REDIRECTOR=devbox.cern.ch

export EOS_VST_BROKER_URL=root://devbox.cern.ch:1099//eos/

export EOS_VST_TRUSTED_HOST=devbox.cern.ch

{verbatim}

In order to internally authenticate the {mgm} and {fst} nodes using the simple shared secret mechanism, create a simple shared secret {keytab} file. {verbatim} xrdsssadmin -k eoscta -u daemon -g daemon add /etc/eos.keytab {verbatim}

Create a system {/etc/krb5.keytab} file if one does not already exist, for example install the {cern-get-keytab} rpm if the development box is at CERN and runs a CERN supported version of linux. {verbatim} yum install cern-get-keytab {verbatim}

In order for the EOS {mgm} to authenticate users using kerberos, create a a new {eos} service principal in the {kdc}, and get the key installed in the keytab. This will also recreate new version of every other key for this host. The key the eos principal can then be extracted to a new keytab, which will be owned by user daemon so it becomes readable by the {mgm}. {verbatim} [root ∼]# cern-get-keytab –service eostest -f Waiting for password replication (0 seconds past) Waiting for password replication (5 seconds past) Waiting for password replication (10 seconds past) Keytab file saved: /etc/krb5.keytab [root ∼]# ktutil ktutil: rkt /etc/krb5.keytab ktutil: l slot KVNO Principal

1 14 devbox$.CH 2 14 devbox$.CH 3 14 devbox$.CH 4 14 eoscta/devbox.cern.ch.CH 5 14 eoscta/devbox.cern.-ch.CH 6 14 eoscta/devbox.cern.ch.CH ktutil: delent 1 ktutil: delent 1 ktutil: delent 1 ktutil: l slot KVNO Principal

1 14 eoscta/devbox.cern.ch.CH 2 14 eoscta/devbox.cern.ch.CH 3 14 eoscta/devbox.cern.ch.CH ktutil: wkt /etc/krb5.keytab.eos ktutil: q [root ∼]# chown daemon.daemon /etc/krb5.keytab.eos {verbatim}

This operation will re-generate all the keys of the host. It might require the client users to {kdestroy} their corresponding tickets in caches.

Backup the original {/etc/xrd.cf.mgm} file installed by the {eos-server} rpm. {verbatim} sudo cp /etc/xrd.cf.mgm /etc/xrd.cf.mgm_ORGINIAL {verbatim}

Disable the unix based authentication mechanism of xroot. {verbatim} sudo sed -i 's/^sec.protocol unix.

6 Command Line Interface

These are the commands that only administrators are allowed to use to operate CTA. For each command there is a short version and a long one, example: {op/operator}. Subcommands ({add}, {rm}, {ls}, {ch}, {reclaim}) do not have short versions.

{itemize} cta ad/admin cta ah/adminhost{ {hostgroups also? not for the prototype}} cta us/user{ {or maybe power users? or egroups? will see later, for the moment just users}} cta tp/tapepool{ {listing the tapepools should include also stats such as total number of tapes and number of free tapes to help operator}} cta ar/archiveroute cta ll/logicallibrary cta ta/tape cta sc/storageclass{ {storage classes should be as static as possible, no change nor deletion if there is at least 1 directory using it}} cta lpa/listpendingarchives cta lpr/listpendingretrieves cta lds/listdrivestates{ {more or less the "drive" part of the current showqueues -x}} {itemize}

The detailed list of the commands with their subcommands and parameters follows.

cta admin} This command is used to manage the administrators of the system. These are special users allowed to perform the admin commands described in this section. {center} {tabular}{ l l } [gray]{0.9}{ Command & Subcommand} & { Parameters} \ {3}{∗}{cta ad/admin add} & -{}-uid/-u <{uid}> \ & -{}-gid/-g <{gid}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{3}{∗}{cta ad/admin ch} & -{}-uid/-u <\textit{uid}> \\ & -{}-gid/-g <\textit{gid}> \\ & -{}-comment/-m <\textit{comment"}> \ {2}{∗}{cta ad/admin rm} & -{}-uid/-u <{uid}> \ & -{}-gid/-g <{gid}> \ cta ad/admin ls & \ {tabular} {center}

cta adminhost} This command is used to manage the administrator hosts. These are specific machines from which administrators may issue the admin commands described in this section. {center} {tabular}{ l l } [gray]{0.-9}{ Command & Subcommand} & { Parameters} \ {2}{∗}{cta ah/adminhost add} & -{}-name/-n <{host_name}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{2}{∗}{cta ah/adminhost ch} & -{}-name/-n <\textit{host_name}> \\ & -{}-comment/-m <\textit{comment"}> \ cta ah/adminhost rm & -{}-name/-n <{host_name}> \ cta ah/adminhost ls & \ {tabular} {center}

cta user} This command is used to manage the "normal" users of CTA. These are typically physicists and experiment data storage managers, who are able to perform only the actions described in the next section ({archive}, {retrieve}, etc.). {center} {tabular}{ l l } [gray]{0.9}{ Command & Subcommand} & { Parameters} \ {3}{∗}{cta us/user add} & -{}-uid/-u <{uid}> \ & -{}-gid/-g <{gid}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{3}{∗}{cta us/user ch} & -{}-uid/-u <\textit{uid}> \\ & -{}-gid/-g <\textit{gid}> \\ & -{}-comment/-m <\textit{comment"}> \ {2}{∗}{cta us/user rm} & -{}-uid/-u <{uid}> \ & -{}-gid/-g <{gid}> \ cta us/user ls & \ {tabular} {center}

cta tapepool} This command is used to manage the tape pools, which are logical sets of tapes. These are useful to manage the life cycle of tapes tolabel $$ supply $$ user pool $$ erase $$ tolabel. {center} {tabular}{ l l } [gray]{0.9}{ Command & Subcommand} & { Parameters} \ {3}{∗}{cta tp/tapepool add} & -{}-name/-n <{tapepool_name}> \ & -{}-partialtapesnumber/-p <{number_of_partial_tapes}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{3}{∗}{cta tp/tapepool ch} & -{}-name/-n <\textit{tapepool_name}> \\ & -{}-partialtapesnumber/-p <\textit{number_of_partial_tapes}> \\ & -{}-comment/-m <\textit{comment"}> \ cta tp/tapepool rm & -{}-name/-n <{tapepool_name}> \ cta tp/tapepool ls & \ {tabular} {center}

cta archiveroute} This command is used to manage the archive routes, which are the policies linking name space entries to tape pools. {center} {tabular}{ l l } [gray]{0.9}{ Command & Subcommand} & { Parameters} \ {4}{∗}{cta ar/archiveroute add} & -{}-storageclass/-s <{storage_class_name}> \ & -{}-copynb/-c <{copy_number}> \ & -{}-tapepool/-t <{tapepool_name}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{4}{∗}{cta ar/archiveroute ch} & -{}-storageclass/-s <\textit{storage_class_name}> \\ & -{}-copynb/-c <\textit{copy_number}> \\ & -{}-tapepool/-t <\textit{tapepool_name}> \\ & -{}-comment/-m <\textit{comment"}> \ {2}{∗}{cta ar/archiveroute rm} & -{}-storageclass/-s <{storage_class_name}> \ & -{}-copynb/-c <{copy_number}> \ cta ar/archiveroute ls & \ {tabular} {center}

cta logicallibrary} This command is used to manage the logical libraries. These are logical groupings of tapes and drives based on physical location and tape drive capabilities. Basically a tape can be accessed by a drive if it is in the same physical library and if the drive is capable of reading or writing the tape, in that case we typically have that that tape and that drive are in the same logical library. {center} {tabular}{ l l } [gray]{0.9}{ Command & Subcommand} & { Parameters} \ {2}{∗}{cta ll/logicallibrary add} & -{}-name/-n

<{logical_library_name}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{2}{*}{cta ll/logicallibrary ch} & -{}-name/-n <\textit{logical_library_name}> \\ & -{}-comment/-m <\textit{comment"}> \ cta ll/logicallibrary rm & -{}-name/-n <{logical_library_name}> \ cta ll/logicallibrary ls & \ {tabular} {center}

cta tape} This command is used to manage the tapes. These are the physical containers of data. {center} {tabular}{ l l } [gray]{0.9}{ Command & Subcommand} & { Parameters} \ {5}{*}{cta ta/tape add} & -{}-vid/-v <{vid}> \ & -{}-logicallibrary/-l <{logical_library_name}> \ & -{}-tapepool/-t <{tapepool_name}> \ & -{}-capacity/-c <{capacity_in_bytes}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{5}{*}{cta ta/tape ch} & -{}-vid/-v <\textit{vid}> \\ & -{}-logicallibrary/-l <\textit{logical_library_name}> \\ & -{}-tapepool/-t <\textit{tapepool_name}> \\ & -{}-capacity/-c <\textit{capacity_in_bytes}> \\ & -{}-comment/-m <\textit{comment"}> \ cta ta/tape rm & -{}-vid/-v <{vid}> \ cta ta/tape reclaim & -{}-vid/-v <{vid}> \ cta ta/tape ls & \ {tabular} {center}

cta storageclass} This command is used to manage the storage classes. These can be associated with CTA directories and they determine the number of tape copies the files in the directory should have. {center} {tabular}{ l l } [gray]{0.9}{ Command & Subcommand} & { Parameters} \ {3}{*}{cta sc/storageclass add} & -{}-name/-n <{storage_class_name}> \ & -{}-copynb/-c <{number_of_tape_copies}> \ & -{}-comment/-m <{comment"}> \\ \hline \multirow{3}{*}{cta sc/storageclass ch} & -{}-name/-n <\textit{storage_class_name}> \\ & -{}-copynb/-c <\textit{number_of_tape_copies}> \\ & -{}-comment/-m <\textit{comment"}> \ cta sc/storageclass rm & -{}-name/-n <{storage_class_name}> \ cta sc/storageclass ls & \ {tabular} {center}

cta list} This set of commands is used to list the pending archives and retrieves as well as the state of each tape drive (its status –up or down–, its contents, etc.). {center} {tabular}{ l l } [gray]{0.9}{ Command} & { Parameters} \ cta lpa/listpendingarchives & -{}-tapepool/-t <{tapepool_name}> \ cta lpr/listpendingretrieves & -{}-vid/-v <{vid}> \ cta lds/listdrivestates & \ {tabular} {center}

Questions that administrators need to be able to answer easily using commands above (not necessarily for the prototype, but to keep in mind):

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

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).

{itemize} cta lsc/liststorageclass{ {this command might seem a duplicate of the corresponding admin command but it actually shows a subset of fields (name and number of copies)}} cta ssc/setstorageclass <dirpath> <storage_class_name> cta csc/clearstorageclass <dirpath> cta mkdir <dirpath> cta chown <uid> <gid> <dirpath>{ {we may want to add {chmod} later on}} cta rmdir <dirpath> cta ls <dirpath> cta a/archive <src1> [<src2> [<src3> [...]]] <dst> cta r/retrieve <src1> [<src2> [<src3> [...]]] <dst> cta da/deletearchive <dst>{ {this works both on ongoing and finished archives, that is why it's called delete"}} \item cta cr/cancelretrieve <dst>{\normalfont \footnote{this clearly works only on ongoing retrieves, obviously does not delete destination files, that's why it's calledcancel"}} {itemize}

{ObjectStore.tex}

{TapeSessions.tex}

7 ‘ctafrontend‘ docker image repository

- This is the repository for CERN Tape Archive frontend docker image.

Aim

The aim of this project is to provide a CTA frontend through Docker® containers.

Components

- Dockerfile - The file describing how to build the Docker image for building CASTOR, in turn.
- etc/yum.repos.d - directory containing yum repos for installing necessary packages.
- etc/xrootd - directory containing static configuration files for xrootd
- run.sh - The main script to setup runtime environment.

Setup

In order to be able to use the container, you should have Docker installed on your machine. You can get more information on how to setup Docker here.

The base image used is CERN CentOS 7 (gitlab-registry.cern.ch/linuxsupport/cc7-base).

Build image

In order to build the image, after making sure that the Docker daemon is running, run from the repository directory:

"'bash docker build –force-rm -t ctafrontend-cc7 cc7/ "'

After the image has finished building successfully, run the following command:

"'bash

Run CTA fronted with prepared object store and catalogue DB.

All logs will be passed back to the docker host through /dev/log socket.

Shared path must be passed inside to the container.

A host name have to be used for the container to run xrootd.

#

objectstore CTA object store directory path name.

catdb CTA catalogue DB setup.

#

docker run -h ctasystestf.cern.ch -it -e objectstore="/shared/jobStoreVFS1FlpYW" -e catdb="sqlite:/shared/sqlite-Db/db" -v /dev/log:/dev/log -v /opt/cta/docker:/shared ctafrontend-cc7 "'

8 'ctafrontend' docker image repository

- This is the repository for CERN Tape Archive frontend docker image.

Aim

The aim of this project is to provide a CTA frontend through Docker® containers.

Components

- Dockerfile - The file describing how to build the Docker image for building CASTOR, in turn.
- etc/yum.repos.d - directory containing yum repos for installing necessary packages.
- etc/xrootd - directory containing static configuration files for xrootd
- run.sh - The main script to setup runtime environment.

Setup

In order to be able to use the container, you should have Docker installed on your machine. You can get more information on how to setup Docker `here`.

The base image used is CERN CentOS 7 (gitlab-registry.cern.ch/linuxsupport/cc7-base).

Build image

In order to build the image, after making sure that the Docker daemon is running, run from the repository directory:

"'bash docker build –force-rm -t ctafrontend-cc7 cc7/ "'

After the image has finished building successfully, run the following command:

"'bash

Run CTA fronted with prepared object store and catalogue DB.

All logs will be passed back to the docker host through /dev/log socket.

Shared path must be passed inside to the container.

A host name have to be used for the container to run xrootd.

#

objectstore CTA object store directory path name.

catdb CTA catalogue DB setup.

#

docker run -h ctasystestf.cern.ch -it -e objectstore="/shared/jobStoreVFS1FlpYW" -e catdb="sqlite:/shared/sqlite-Db/db" -v /dev/log:/dev/log -v /opt/cta/docker:/shared ctafrontend-cc7 "'

9 z_24990_README

Folder content:

- `ci_helpers`: helper scripts used by gilab_ci
- `docker`: Docker files and content needed to build the needed docker images
- `orchestration`: orchestration specific files: this contains all the pods definitions and *startup scripts* launched in the pod containers during instanciation.

- `buildtree_runner`: setup scripts for running the ci tests from the build tree instead of RPMs
- `buildtree_runner/vmBootstrap`: As set of scripts describing the preparation of a machine for fresh install to running CI scripts **MUST NOT BE RUN BLINDLY**

10 Run a CTA test instance from build tree in an independent virtual machine.

For the test instance details, see the orchestration directory.

The scripts in this directory allow to run the continuous integration system tests from a locally built CTA, without RPMs.

The target is to run in a virtual machine, possibly in a disconnected laptop.

Setting up a fresh virtual machine

User environment

The vmBootstrap directory contains all the necessary script to go from minimal CC7 instalation to running kubernetes with CTA checked out and compiled.

The directory vmBootstrap should be copied at the root of the new machine and the script bootstrapSystem.sh should be run from /wmBootstrap:

"' cd /vmBootstrap ./bootstrapSystem.sh "'

This will create a new user (currently hardcoded to "eric") and prompt for the password. The use will be a sudoer (no password).

CTA build tree

The user should then login as the user, kinit with a valid CERN.CH token, and then run the next step: bootstrapCTA.sh: "' kinit `user@CERN.CH` cd /vmBootstrap ./bootstrapCTA.sh "'

This will check out CTA from git, in install the necessary build RPMs and compile.

Kubernetes setup

The user should then run the script to setup kubernetes: "' cd /vmBootstrap ./bootstrapKubernetes.sh "'

A reboot is currently required at that point.

Running the system tests.

Docker image

The system tests run with a single image with all the non CTA rpms pre-loaded. The image should be generated once for all:

"' cd ∼/CTA/continuousintegration/buildtree_runner ./prepareImage.sh "'

This image also contains embedded scripts from the CTA tree. As the build is 2 stage, the embedding of scripts can be run separately, saving the time to install the RPMs.

Preparing the environment (MHVTL, kubernetes volumes...)

MHVTL should then be setup by running:

"' cd ∼/CTA/continuousintegration/buildtree_runner ./recreate_buildtree_running_environment.sh "'

Preparing the test instance

The test instance can then be created by running (typically):

"‘ cd ∼/CTA/continuousintegration/orchestration ./create_instance.sh -n ctatest -b /home/eric -B CTA-build -D -O "‘

and the tests will run like in the continuous integration environment.

11 Launching a CTA test instance

A CTA test instance is a kubernetes namespace. Basically it is a cluster of *pods* on its own DNS sub-domain, this means that inside a CTA namespace `ping ctafrontend` will ping *ctafrontend.<namespace>.cluster.local* i.e. the CTA frontend in the current instance, same for `ctaeos` and other services defined in the namespace.

Before going further, if you are completely new to `kubernetes`, you can have a look at this `CS3 workshop presentation`. The web based presentation is available `here`.

setting up the CTA `kubernetes` infrastructure

All the needed tools are self contained in the `CTA` repository. It allows to version the system tests and all the required tools with the code being tested. Therefore setting up the system test infrastructure only means to checkout `CTA` repository on a kubernetes cluster: a `ctadev` system for example.

Everything in one go *aka* the **Big Shortcut**

This is basically the command that is run by the `gitlab CI` in the `CI pipeline` executed at every commit during the `test` stage in the `archieveretrieve` build. `Here` is an example of successfully executed `archieveretrieve` build. Only one command is run in this build: "‘ $ cd continuousintegration/orchestration/; \ ./run_systemtest.sh -n ${NAMESPACE} -p ${CI_PIPELINE_ID} -s tests/systest.sh "‘

`CI_PIPELINE_ID` is not needed to run this command interactively: you can just launch: "‘ [root CTA]# cd continuousintegration/orchestration/ [root orchestration]# ./run_systemtest.sh -n mynamespace -s tests/systest.sh "‘

But be careful: this command instantiate a CTA test instance, runs the tests and **immediately deletes it**. If you want to keep it after the test script run is over, just add the `-k` flag to the command.

The following sections just explain what happens during the system test step and gives a few tricks and useful kubernetes commands.

List existing test instances

This just means listing the current kubernetes namespaces: "‘ [root ∼]# kubectl get namespace NAME STATUS AGE default Active 18d kube-system Active 3d "‘

Here we just have the 2 kubernetes *system* namespaces, and therefore no test instance.

Create a `kubernetes` test instance

For example, to create `ctatest` CTA test instance, simply launch `./create_instance.sh` from `CTA/continuousintegration/orchestration` directory with your choice of arguments. By default it will use a file based objectstore and an sqlite database, but you can use an Oracle database and/or Ceph based objectstore if you specify it in the command line. "‘ [root CTA]# ./create_instance.sh Usage: ./create_instance.sh -n <namespace> [-o <objectstore_configmap>] [-d <database_configmap>] "‘

Objectstore configmap files and database configmap files are respectively available on `cta/dev/ci` hosts in `/opt/kubernetes/CTA/[database|objectstore]/config`, those directories are managed by Puppet and the accounts configured on your machine are yours.

YOU ARE RESPONSIBLE FOR ENSURING THAT ONLY 1 INSTANCE USES 1 EXCLUSIVE REMOTE RESOURCE. RUNNING 2 INSTANCES WITH THE SAME REMOTE RESOURCE WILL CREATE CONFLICTS IN THE WORKFLOWS AND IT WILL BE YOUR FAULT

After all those WARNINGS, let's create a CTA test instance that uses **your** Oracle database and **your** Ceph objectstore.

"' [root CTA]# cd continuousintegration/orchestration/ [root orchestration]# git pull Already up-to-date. [root orchestration]# ./create_instance.sh -n ctatest \ -o /opt/kubernetes/CTA/objectstore/config/objectstore-ceph-cta-julien.yaml \ -d /opt/kubernetes/CTA/database/config/database-oracle-cta_devdb1.yaml -O -D Creating instance for latest image built for 40369689 (highest PIPELINEID) Creating instance using docker image with tag: 93924git40369689 DB content will be wiped objectstore content will be wiped Creating ctatest instance namespace "ctatest" created configmap "init" created creating configmaps in instance configmap "objectstore-config" created configmap "database-config" created Requesting an unused MHVTL librarypersistentvolumeclaim "claimlibrary" created .OK configmap "library-config" created Got library: sg35 Creating services in instance service "ctacli" created service "ctaeos" created service "ctafrontend" created service "kdc" created Creating pods in instance pod "init" created Waiting for init..OK Launching pods pod "ctacli" created pod "tpsrv" created pod "ctaeos" created pod "ctafrontend" created pod "kdc" created Waiting for other pods.....OK Waiting for KDC to be configured..........................OK Configuring KDC clients (frontend, cli...) OK klist for ctacli: Ticket cache: FILE:/tmp/krb5cc_0 Default principal: `admin1@TEST.CTA`

Valid starting Expires Service principal 03/07/17 23:21:49 03/08/17 23:21:49 krbtgt/TEST.CTA.CTA Configuring cta SSS for ctafrontend access from ctaeos.....................OK Waiting for EOS to be configured........OK Instance ctatest successfully created: NAME READY STATUS RESTARTS AGE ctacli 1/1 Running 0 1m ctaeos 1/1 Running 0 1m ctafrontend 1/1 Running 0 1m init 0/1 Completed 0 2m kdc 1/1 Running 0 1m tpsrv 2/2 Running 0 1m "'

This script starts by creating the `ctatest` namespace. It runs on the latest CTA docker image available in the gitlab registry. If there is no image available for the current commit it will fail. Then it creates the services in this namespace so that when the pods implementing those services are creates the network and DNS names are defined.

For convenience, we can export `NAMESPACE`, set to `ctatest` in this case, so that we can simply execute `kubectl` commands in our current instance with `kubectl --namespace=${NAMESPACE} ...`.

The last part is the pods creation in the namespace, it is performed in 2 steps:

1. run the `init` pod, which created db, objectstore and label tapes
2. launch the other pods that rely on the work of the `init` pod when its status is `Completed` which means that the init script exited correctly

Now the CTA instance is ready and the test can be launched.

Here are the various pods in our `ctatest` namespace: "' [root orchestration]# kubectl get namespace NAME STATUS AGE ctatest Active 4m default Active 88d kube-system Active 88d [root orchestration]# kubectl get pods -a –namespace ctatest NAME READY STATUS RESTARTS AGE ctacli 1/1 Running 0 3m ctaeos 1/1 Running 0 3m ctafrontend 1/1 Running 0 3m init 0/1 Completed 0 4m kdc 1/1 Running 0 3m tpsrv 2/2 Running 0 3m "'

Everything looks fine, you can even check the logs of `eos` mgm: "' [root CTA]# kubectl –namespace $NAMESPACE logs ctaeos ...

ctaeos mgm ready

"'

Or the `Completed` init pod: "' [root CTA]# kubectl –namespace $NAMESPACE logs init ... Using this configuration for library: Configuring mhvtl library DRIVESLOT is not defined, using driveslot 0 export LIBRARYTYPE=mhvtl export LIBRARYNAME=VLSTK60 export LIBRARYDEVICE=sg35 export DRIVENAMES=(VDSTK61 VDSTK62 VDSTK63) export DRIVEDEVICES=(nst15 nst16 nst17) export TAPES=(V06001 V06002 V06003 V06004 V06005 V06006 V06007) export driveslot=0 Configuring objectstore: Configuring ceph objectstore Wiping objectstore Rados objectstore rados://cta-julien:cta-julien is not empty: deleting content New object store path: rados://cta-julien:cta-julien Rados objectstore rados://cta-julien:cta-julien content: driveRegister-cta-objectstore-initialize-init-295-20170307-23:20:49-1 cta-objectstore-initialize-init-295-20170307-23:20:49 agentRegister-cta-objectstore-initialize-init-295-20170307-23:20:49-0 schedulerGlobalLock-cta-objectstore-initialize-init-295-20170307-23:20:49-

2 root Configuring database: Configuring oracle database Wiping database Aborting: unlockSchema failed: executeNonQuery failed for SQL statement... ORA-00942: table or view does not exist Aborting: schemaIsLocked failed: executeQuery failed for SQL statement ... ORA-00942: table or view does not exist Aborting: schema-IsLocked failed: executeQuery failed for SQL statement ... ORA-00942: table or view does not exist Database contains no tables. Assuming the schema has already been dropped. Labelling tapes using the first drive in VL-STK60: VDSTK61 on /dev/nst15: V06001 in slot 1 Loading media from Storage Element 1 into drive 0...done 1+0 records in 1+0 records out 80 bytes (80 B) copied, 0.00448803 s, 17.8 kB/s Unloading drive 0 into Storage Element 1...done OK ... "'

Running a simple test

Go in `CTA/continuousintegration/orchestration/tests` directory: "' [root CTA]# cd continuous-integration/orchestration/tests "'

From there, launch `./systest.sh`. It configures CTA instance for the tests and `xrdcp` a file to EO-S instance, checks that it is on tape and recalls it: "' [root tests]# ./systest.sh -n ctatest Reading library configuration from tpsrvOK Using this configuration for library: LIBRARYTYPE=mhvtl LIBRARYNAME=VLST-K60 LIBRARYDEVICE=sg35 DRIVENAMES=(VDSTK61 VDSTK62 VDSTK63) DRIVEDEVICES=(nst15 nst16 nst17) TAPES=(V06001 V06002 V06003 V06004 V06005 V06006 V06007) driveslot=0 Preparing CTA for tests Drive VDSTK61 set UP. EOS server version is used: eos-server-4.1.11-20170118171644git24cd94a.-el7.x86_64 xrdcp /etc/group root://localhost//eos/ctaeos/cta/4ff9d278-4c92-427a-b72b-a28cefddffcd [420B/420-B][100%][==][420B/s] Waiting for file to be archived to tape: Seconds passed = 0 ...

OK: all tests passed "'

If something goes wrong, please check the logs from the various containers running on the pods that were defined during dev meetings:

1. `init`
   - Initializes the system, e.g. labels tapes using cat, creates/wipes the catalog database, creates/wipes the CTA object store and makes sure all drives are empty and tapes are back in their home slots.
2. `kdc`
   - Runs the KDC server for authenticating EOS end-users and CTA tape operators.
3. `eoscli`
   - Hosts the command-line tools to be used by an end-user of EOS, namely xrdcp, xrdfs and eos.
4. `ctaeos`
   - One EOS mgm.
   - One EOS fst.
   - One EOS mq.
   - The cta command-line tool to be run by the EOS workflow engine to communicate with the CTA front end.
   - The EOS Simple Shared Secret (SSS) to be used by the EOS mgm and EOS fst to authenticate each other.
   - The CTA SSS to be used by the cta command-line tool to authenticate itself and therefore the EOS instance with the CTA front end.
   - The tape server SSS to be used by the EOS mgm to authenticate file transfer requests from the tape servers.
5. `ctacli`
   - The cta command-line tool to be used by tape operators.
6. `ctafrontend`

  - One CTA front-end.
  - The CTA SSS of the EOS instance that will be used by the CTA front end to authenticate the cta command-line run by the workflow engine of the EOS instance.

7. `tpsrvXXX` *No two pods in the same namespace can have the same name, hence each tpsrv pod will be numbered differently*
  - One `cta-taped` daemon running in `taped` container of `tpsrvxxx` pod.
  - One `rmcd` daemon running in `rmcd` container of `tpsrvxxx` pod.
  - The tape server SSS to be used by cta-taped to authenticate its file transfer requests with the EOS mgm (all tape servers will use the same SSS).

post-mortem analysis

logs

An interesting feature is to collect the logs from the various processes running in containers on the various pods. Kubernetes allows to collect `stdout` and `stderr` from any container and add time stamps to ease post-mortem analysis.

Those are the logs of the `init` pod first container: "' [root CTA]# kubectl logs init –timestamps –namespace $NAMESPACE 2016-11-08T20:00:28.154608000Z New object store path: file:///shared/test/objectstore 2016-11-08T20:00:31.490246000Z Loading media from Storage Element 1 into drive 0...done 2016-11-08T20:00:32.712066000Z 1+0 records in 2016-11-08T20:00:32.712247000Z 1+0 records out 2016-11-08T20:00:32.712373000Z 80 bytes (80 B) copied, 0.221267 s, 0.4 kB/s 2016-11-08T20:00:32.733046000Z Unloading drive 0 into Storage Element 1...done "'

Dump the objectstore content

Connect to a pod where the objectstore is configured, like `ctafrontend`: "' [root CTA]# kubectl –namespace $NAMESPACE exec ctafrontend -ti bash "'

From there, you need to configure the objecstore environment variables, sourcing `/tmp/objectstore-rc.sh` install the missing `protocolbuffer` tools like `protoc` binary, and then dump all the objects you want: "' [root /]# yum install -y protobuf-compiler ... Installed: protobuf-compiler.x86_64 0:2.5.0-7.el7

Complete!

[root /]# . /tmp/objectstore-rc.sh

[root /]# rados ls -p $OBJECTSTOREPOOL –id $OBJECTSTOREID –namespace $OBJECTSTORENAMESPACE OStoreDBFactory-tpsrv-340-20161216-14:17:51 OStoreDBFactory-ctafrontend-188-20161216-14:15:35 schedulerGlobalLock-makeMinimalVFS-init-624-20161216-14:14:17-2 driveRegister-makeMinimalVFS-init-624-20161216-14:14:17-1 makeMinimalVFS-init-624-20161216-14:14:17 agentRegister-makeMinimalVFS-init-624-20161216-14:14:17-0 root retriveQueue-OStoreDBFactory-ctafrontend-188-20161216-14:15:35-3

[root /]# rados get -p $OBJECTSTOREPOOL –id $OBJECTSTOREID \ –namespace $OBJECTSTORENAMESPACE retriveQueue-OStoreDBFactory-ctafrontend-188-20161216-14:15:35-3 \

- | protoc –decode_raw 1: 10 2: 0 3: "root" 4: "root" 5 { 10100: "V02001" 10131 { 9301: 1 9302: 1 } 10132 { 9301: 1 9302: 1 } 10133 { 9301: 1 9302: 1 } 10140: 406 10150: 0 } "'

Delete a `kubernetes` test instance

Deletion manages the removal of the infrastructure that had been created and populated during the creation procedure: it deletes the database content and drop the schema, remove all the objects in the objectstore independently of the type of resources (ceph objectstore, file based objectstore, oracle database, sqlite database...).

In our example: "' [root CTA]# ./delete_instance.sh -n $NAMESPACE Deleting ctatest instance Unlock cta catalog DB Delete cta catalog DB for instance Deleted 1 admin users Deleted 1 admin hosts Deleted 1 archive routes

Deleted 1 requester mount-rules Deleted 0 requester-group mount-rules Deleted 1 archive files and their associated tape copies Deleted 7 tapes Deleted 1 storage classes Deleted 1 tape pools Deleted 1 logical libraries Deleted 1 mount policies Status cta catalog DB for instance UNLOCKED Drop cta catalog DB schema for instance namespace "ctatest" deleted ....OK Deleting PV sg34 with status Released persistentvolume "sg34" deleted persistentvolume "sg34" created OK Status of library pool after test: NAME CAPACITY ACCESSMODES STATUS CLAIM REASON AGE sg30 1Mi RWO Available 19h sg31 1Mi RWO Available 19h sg32 1Mi RWO Available 19h sg33 1Mi RWO Available 18h sg34 1Mi RWO Available 0s sg35 1Mi RWO Available 19h sg36 1Mi RWO Available 19h sg37 1Mi RWO Available 19h sg38 1Mi RWO Available 19h sg39 1Mi RWO Available 18h "'

When this deletion script is finished, the `ctatest` instance is gone: feel free to start another one...

"' [root CTA]# kubectl get namespace NAME STATUS AGE default Active 6d kube-system Active 6d "'

Long running tests

If some tests run for long, the kerberos token in the cli pod should be renewed with:

"' kubectl –namespace=${instance} exec ctacli – kinit -kt /root/admin1.keytab `admin1@TEST.CTA` "'

Gitlab CI integration

Configure the Runners for `cta` project and add some specific tags for tape library related jobs. I chose `mhvtl` and `kubernetes` for ctadev runners. This allows to use those specific runners for CTA tape library specific tests, while others can use shared runners.

A small issue: by default, `gitlab-runner` service runs as `gitlab-runner` user, which makes it impossible to run some tests as `root` inside the pods has not the privileges to run all the commands needed.