Archiving tools for EOS

Elvin-Alin Sindrilaru
CERN IT, Geneva, Switzerland
E-mail: elvin.alin.sindrilaru@cern.ch

Andreas-Joachim Peters
CERN IT, Geneva, Switzerland
E-mail: andreas.joachim.peters@cern.ch

Dirk Duellmann
CERN IT, Geneva, Switzerland
E-mail: dirk.duellmann@cern.ch

Abstract.
Archiving data to tape is a critical operation for any storage system, especially for the EOS system at CERN which holds production data for all major LHC experiments. Each collaboration has an allocated quota it can use at any given time therefore, a mechanism for archiving "stale" data is needed so that storage space is reclaimed for online analysis operations. The archiving tool that we propose for EOS aims to provide a robust client interface for moving data between EOS and CASTOR (tape backed storage system) while enforcing best practices when it comes to data integrity and verification.

All data transfers are done using a third-party copy mechanism which ensures point-to-point communication between the source and destination, thus providing maximum aggregate throughput. Using ZMQ message-passing paradigm and a process-based approach enabled us to achieve optimal utilisation of the resources and a stateless architecture which can easily be tuned during operation. The modular design and the implementation done in a high-level language like Python, has enabled us to easily extended the code base to address new demands like offering full and incremental backup capabilities.

1. Introduction
The EOS project started in April 2010 at CERN in the IT Data and Storage Services Group aims to provide a cost-effective, low-latency, disk-only storage solution for all the analysis use cases of both LHC and non-LHC experiments. The amount of data stored in EOS has continuously increased over the years, reaching at this point more than 140 PB of raw disk space spread across several instances. The system was built with modularity and flexibility in mind from the beginning. In order to accomodate different scenarios, both administrators and users are provided with the necessary tools to tune the system to their particular requirements.

One such example of a critical functionality is the quota sub-system. The system or group administrators can enforce different quota values for both the number of inodes and the volume of data that is stored under a certain sub-tree. The ability to manage and easily configure precise
policies when it comes to resource allocation is paramount in a system which targets a highly collaborative and dynamic community such as the case for high-energy physics. Given that the amount of hardware resources made available to the EOS system are limited, users are regularly faced with the situation in which they reach their allocated quota. In such a case they have several options:

- Ask for an increase of the quota - operation which most of the times is impossible because of hardware resource limitations or it can be done but at the expense of a different user/group having their quota decreased.
- Delete some of the data under their account thus freeing space for new data. Although this is always an option, most of the times it is not desirable from the user’s point of view as they might want to review or just have the old data available for reference.
- Move the data to a tape backend so that it is available in the future if required. This relies on the availability of a tape storage system which we consider as providing "infinite" storage capability.

In this paper we will present an archiving tool developed for the EOS system that deals with the quota limitation by taking care of the movement of data between the disk-based system and any XRootD\cite{2} capable tape storage system. In our case, the tape storage system of choice is the CERN Advanced STORage manager - CASTOR\cite{3}. The rest of the paper will concentrate on the architecture of the system and the necessary pre-existing functionality that facilitates the archiving operation for a system which is concurrently accessed and used by multiple users.

2. Architecture and implementation

The archiving tool is implemented as a separate daemon which is able to communicate directly using the XRootD client with both the disk (EOS) and the tape (CASTOR) storage systems. Any archiving command must be triggered by the client interacting with the EOS system. From this point of view, the client has no knowledge of how the archiving mechanism is implemented. Decoupling the archiving system from direct contact with the clients enables future incremental improvements and additions without any modifications to the interface.

Figure \ref{fig:arch} describes the main interacting entities and the stages through which an archive command goes: the client uses the EOS command-line interface to submit a request to the MGM\cite{1}, the MGM decides based on the identity of the client if the request can be authorized and if so forwards it to the archive daemon using a custom communication protocol, the archive daemon then registers the request in the pending queue and depending on the configuration policy decides when to start executing the transfer.

The first stage in creating an archive is taking a snapshot of the metadata information of the corresponding directories and files included in the archive. In our case, an archive can be created only at a directory level and contains all the sub-directories and files within that sub-tree. To ensure that the data and metadata of the archive contents remain consistent throughout the archiving process, we have implemented a mechanism to block any updates or writes to the archived sub-tree by using a dedicated ACL flag. By freezing the archive namespace using the immutable flag: \texttt{sys.acl\textasciitilde z;i}, the users still have the possibility to read the data from the archive until they decide to purge it from EOS.

2.1. Archive file

After the namespace freeze is enforced, the archive system takes a snapshot of the metadata information corresponding to the entries in the sub-tree. All this metadata along with other

\footnote{EOS metadata service}
Figure 1. Architecture

information such as the time the archive was made, name of the user who triggered it etc. are
saved in a special file called .archive.init. The archive file is an ASCII file made up of JSON
objects corresponding to metadata information associated to each entry from the archive.

The first line of the archive file contains the archive header. In order to reduce as much as
possible the size of the archive file, the header contains information about the metadata that is
saved for each of the sub-directories and files in the archive - information which is later on used
in the verification step. An example of an archive header is presented in [Listing 1].

Listing 1. Archive header

```json
{
    "src": "root://eos.cern.ch//eos/src/",
    "dst": "root://castor.cern.ch//castor/dst/",
    "svc_class": "diskonly",
    "dir_meta": ["uid", "gid", "mode", "attr"],
    "file_meta": ["size", "mtime", "ctime", "uid", "gid", "mode", "
xstype", "xs"],
    "uid": "0",
    "gid": "0",
    "timestamp": 1424625754,
    "num_dirs": 2,
    "num_files": 5
}
```

The dir_meta and file_meta entries represent the metadata information that is saved for
each sub-directory/file in the archived sub-tree. The last two entries in the JSON archive header
represent the number of sub-directories and files the archive contains. This information is useful
when providing feedback to the user on the progress of various activities. The rest of the archive
file contains information about the entries making up the archive, also in JSON format. The
archive file is saved at the root of the archived directory and will later on be used by the archive
daemon to initiate the transfer and also perform the necessary integrity checks.

2.2. Communication protocol
The communication between the MGM and the archive daemon is done using ZMQ\cite{5}. In the current implementation both daemons run on the same physical machine therefore all the communication is done using an IPC (inter-process communication) mechanism. By leveraging the flexibility of ZMQ, this can easily be changed to a full TCP stack without major modification of the underlying code. Another feature of ZMQ that we have fully exploited in our implementation is the ease of routing messages using the JSON format. By building around this model, we ensure that the entire system can be easily extended with different types of messages as well as enhancing the current messages with additional information.

The protocol relies on a request-reply model, where the archive daemon either just acknowledges the receipt of a particular request or actually provides information back to the MGM as it is the case for listing the status of ongoing transfers.

2.3. Transfer mechanism
The core of the archive system is represented by the transfer mechanism which is built around the XRootD third-party copy functionality. For each archive request the archive daemon spawns a new process which is responsible with fulfilling all stages of the request. The process-based approach ensures there is no dependency between transfers and any error or crash is contained within the transfer it occurred.

The advantage of the third-party copy mechanism is that the data flows directly from the source to the destination with minimal network traffic. Moreover, the archive process managing the transfer can benefit from enhancements of the underlying copy procedure without any further modifications. Such an example is the introduction of the parallel transfer mode for the XRootD third-party copy functionality that now allows launching several jobs at the same time considerably reducing the total transfer time.

3. Archive life-cycle
An archive follows a well defined series of state machine transitions summarised in Figure 2. The archive command-line interface contains a series of sub-commands that correspond to a specific transition. The first step in archiving a sub-tree is creating the archive file which is done by using the `archive create` command.

Once the archive file is available the user has the option of doing an `archive put` operation in which the data is copied from the disk to the tape system. At any point during the transfer, the user can get a summary of the status of the ongoing operations by doing `archive transfers`. As soon a the "put" operation finishes, one can issue the `archive purge` command which effectively deletes the data from the disk storage making the cleared space available to the user. When the user decides to get back the archived data, he/she can launch the `archive get` command which copies the data from the tape to the disk storage. Obviously, this operations is possible as long as the quota of the user in EOS is not exhausted.

There is an important distinction between the data saved in EOS and the tape system: while in EOS a file can have several replicas or can be stored using a striped layout, on the tape system there will be only one replica of the initial file saved. A similar principle holds when the data is copied back from tape to disk. Therefore, when an initial three replica file is copied from tape back in EOS, it will have the same number of replicas i.e. the initial layout is conserved. All of the archive sub-commands that include a transfer operation have a `recover` flag that can be used by the users in case of transient errors i.e. a file of the archive is not available due to network failures, the tape system is not reachable etc.
Table 1. Archive sub-commands

| Archive sub-command   | Description                                                                 |
|-----------------------|-----------------------------------------------------------------------------|
| create <eos_path>     | create archive file for <eos_path> directory                                 |
| put [-recover] <eos_path> | copy the contents of the EOS directory to the archive destination         |
| get [-recover] <eos_path> | copy the contents of the archive back to EOS                                |
| purge [-recover] <eos_path> | delete all disk resident data keeping the archive file and archived data on tape |
| list [<eos_path>]     | show status of the archived directories in the current sub-tree             |
| transfers [all|put|get|purge|job_uuid] | display ongoing transfers - by default, all transfers are displayed        |
| delete <eos_path>     | delete all archived data including the archive file - admin command         |

Deleting an archive is a privileged operation that can only be performed by the administrator of the system. To achieve this, one can use the `archive delete` command but care should be taken since there are no checks put in place to make sure that the data has been previously recalled from tape and is now available on disk. The delete command effectively deletes all the data associated to the archive from the tape system. A comprehensive list of all the supported archive sub-commands is present in Table 1.

The verification step of an archive operation is different depending on whether it is a disk to tape or a tape to disk transfer. For the former one, only the checksum value and the size of the file are checked for consistency while for the latter one, all metadata information of the newly transferred file has to match exactly with the original one - this means the type of file layout, the creation and modification time, the checksum type and value etc. All of these checks are performed by the archive daemon and any inconsistency leads to the transfer being aborted with a clear indication of the reason why it failed. Each archive transfer has a corresponding log file.
that the user can inspect to determine the reason behind any potential failure.

4. Integration with EOS
Integration of the archive functionality in the existing EOS system was done with minimal modifications to the latter. The authorization mechanism for the archive operations has been implemented as an extension to the already existing system of ACLs. Therefore, a user can archive a directory only if his/her identity is present in the `sys.acl` extended attribute value of the target directory.

Another desirable feature of the system was to have a quick and intuitive mechanism of searching and listing the archived directories in the system. For this purpose, the virtual inode value of the root directory of every archive is saved in a special area of the namespace. To access this information, the user can issue the `archive list` command to get a list of all the archived directories from the system along with their status. Once an archive is deleted, this information is also updated so that it reflects the real-time status of the system.

5. Backup extension
Taking into consideration the requirements of the CERNBox service, the code base of the archive tool was extended with backup capabilities. Conceptually, the backup functionality is to a very large extent similar to the archive `get` operation, when data is copied from tape to disk. Creating a backup or restoring a backup are perfectly symmetric operations. Therefore, from this point of view the implementation of the backup was a bit simpler.

On top of the normal backup operation the CERNBox service also needed an incremental backup functionality. This meant that the backup would only transfer the directories and files that have been created or modified during a certain time window. This is where the extensibility of the JSON archive header came into play, making this additional contraint easy to implement and preserving backward and forward compatibility with other header versions. Since the backup operation tries to create an exact mirror of the directory structure and metadata information of the directory being backed up, the destination must also be an EOS instance with the same set of features as the source. To avoid any compatibility issues a backup can also be done within the same EOS instance.

6. Summary
The EOS archive tool provides the users with the necessary functionality to efficiently manage their transfers from EOS to a tape backend. On top of the simple and intuitive command-line interface, the user is completely decoupled from the intricacies of the tape system and delegates all the verification and integrity checks to the tool. The implementation of the system in a high-level programming language like Python has enabled rapid prototyping and an extensible design which has already been successfully augmented with backup capabilities.

References
[1] Exabyte Scale Storage at CERN, Andreas J Peters and Lukasz Janyst 2011 J. Phys.: Conf. Ser. 331 052015 doi:10.1088/1742-6596/331/5/052015
[2] XRootD, http://xrootd.org
[3] CASTOR, http://castor.web.cern.ch
[4] CERNBox, http://cernbox.cern.ch
[5] ZMQ, http://zeromq.org/