The new CERN tape software - getting ready for total performance

This content has been downloaded from IOPscience. Please scroll down to see the full text.
2015 J. Phys.: Conf. Ser. 664 042007
(http://iopscience.iop.org/1742-6596/664/4/042007)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 137.138.125.164
This content was downloaded on 09/03/2016 at 09:09

Please note that terms and conditions apply.
The new CERN tape software - getting ready for total performance

E Cano¹, S Murray¹, D F Kruse¹, V Kotlyar² and D Côme¹,³

¹CERN (IT/DSS), Geneva, Switzerland
²Institute for High Energy Physics, Protvino, Russia
³Now at ISAE-Supéry, Toulouse, France

E-mail: {Eric.Cano,Steven.Murray,Daniele.Francesco.Kruse}@cern.ch, Victor.Kotlyar@ihep.ru, davidbrcz@gmail.com

Abstract. CASTOR (the CERN Advanced STORage system) is used to store the custodial copy of all of the physics data collected from the CERN experiments, both past and present. CASTOR is a hierarchical storage management system that has a disk-based front-end and a tape-based back-end. The software responsible for controlling the tape back-end has been redesigned and redeveloped over the last year and was put in production at the beginning of 2015. This paper summarises the motives behind the redesign, describes in detail the redevelopment work and concludes with the short and long-term benefits.

1. Introduction

The CERN Advanced STORage manager (CASTOR) [1] is used to archive to tape the physics data of past and present physics experiments. CASTOR currently holds about 100 PB of data. The tape transfer system is expected to face increasing challenges in the medium-term future, with the increasing amount of data recording from LHC experiments. This data flow increase is augmented by the periodic repacking of all data from one tape generation to the next [2] [3].

The CASTOR tape system is composed of a set of database-backed scheduling daemons for queueing file-transfer jobs and a set of disk and tape servers for carrying out those transfers. The jobs are either called migrations (to tape) or recalls (from tape) depending on their direction. This article describes the reimplementation of the software stack running on the tape servers.

1.1. A legacy software stack

The previous tape server software stack initially evolved from the SHIFT [4] system, the predecessor of CASTOR. The daemons and utilities of SHIFT were left mostly intact, with additional daemons from CASTOR running on top of them. Those daemons implemented a hierarchical storage management (HSM) system by adding a namespace and a staging area. The daemons from SHIFT and CASTOR were initially developed in C including the tape server software stack re-written by this C++ project.

This layering of processes created many interprocess interfaces, either through command line parameters or TCP connections. Those interfaces and the long modification history rendered software maintenance and extensions of the tape server software complicated and costly. The software also reached its performance ceiling, and provided only limited information...
for monitoring. With modern tape drives achieving 250 MB/s and speeds up to 1 GB/s on the
roadmaps, operating the drives at full speed promised to become very difficult.

1.2. A simpler replacement with new features
We decided to replace the tape server software with a new one, unencumbered from constrains
initially present at the time of SHIFT to CASTOR transition. This new tape server software is a
drop-in replacement of the previous stack, which was consolidated in a single daemon for easier
operations. It has a pipelined architecture to minimize blocking. Asynchronous instruction
fetch, data read, data write and result reporting all run in separate threads. The new tape
server introduces several features. In depth performance counters enable fast discrimination
between disk servers and tape hardware slowness. Full SCSI tape alerts reporting delivers early
warnings for drive and media failures [5]. A watchdog function unblocks stuck sessions and
cleans up after failures. Special care was taken to integrate the new tape software with the
existing CERN tape logging system in order to improve preventive maintenance procedures.
Those new features permit reducing the workload for tape operators.

The features of the previous software stack were also included, like buffered file marks to
ensure write performance for any size of files [6]. The tape file format and the communication
protocols between the tape server and the other daemons were kept unchanged.

The new tape server daemon forks a separate process for each request to mount a tape. The
forked process contacts a scheduling daemon in order to get instructions for the mount
session. Having one process per session allows for the detection and killing of stuck sessions.
This problem can arise when disk servers stop responding. The detection of stuck sessions is
implemented using a heartbeat protocol.

2. Architecture
The tape mount session implies several steps for each file, which are executed asynchronously
by the pipelined architecture to minimise blocking. For an individual file, we retrieve the work
to be done from the scheduling daemons, create a per-file control structure, read data from tape
or disk to memory, then from memory to disk or tape, and finally report completion or failure
back to the scheduling daemon. Each of those steps happen in a separate thread. In the case
of migrations an additional thread drives memory management. The control and data flow for
a migration session is presented schematically in figure 1.

For each file transfer, a pair of read and write tasks are created and queued for asynchronous
execution. The tasks themselves are memory structures holding file information and in the case
of write tasks they also contain FIFOs for queueing memory blocks.

Separating a transfer session into several simple steps allowed for simpler, quasi single
threaded programming at each step, and facilitated the development of the various steps
independently by several developers.

2.1. FIFO based communication
Inter thread communication is almost entirely based on FIFOs. Writer tasks contain a FIFO
for memory blocks containing data for both recall and migration sessions. In addition, the write
tasks of migration sessions also have a FIFO for queuing free memory blocks. These per task
FIFOs decouple read and write tasks, and makes their implementations straightforward. Error
propagation from reader to writer tasks is achieved by piggy backing status information on the
memory blocks. The remainder of the inter-thread communication is implemented through a
global flag that indicates the current session should shut down.
2.2. Memory management

The tape server memory is used to buffer data between reading from disk (or tape) and writing to tape (or disk). The more memory available, the better the decoupling can be achieved between reading and writing and the more disk files we can transfer in parallel. The memory is divided into fixed sized blocks. The blocks are first filled by the reader task, and are then queued for the writer task. Once data is written to its destination by the writer task, the memory blocks are returned to the central memory manager for recycling. The memory blocks are pre-allocated at session start and recycled until session end.

Figure 1. Threading and data flow model for migration to tape
2.3. Threading models for migrations and recalls
The migration and recall sessions are almost, but not quite symmetric and the threading model reflects this. In both cases, one thread fetches instructions, another reports results and several others transfer data. Disk transfers are carried out by a pool of threads, whereas tape transfers are carried by a single thread. The memory management of migration sessions requires an additional thread.

2.4. Performance monitoring
The FIFOs interfaces to each thread allow easy performance monitoring. Statistics of time waited on each FIFO is gathered and logged. The time to access tape and disk are also gathered and logged [7]. With those statistics in hand, it is very straightforward for tape operators to identify performance bottlenecks — like an overloaded disk server or a poorly performing tape.

2.5. Getting more work and reporting results
The first operation in a data transfer is the fetching of work to be done. In order to keep the pipeline full, we prefetch fixed sized batches of work to be done, and turn them into tasks for tape and disk access. When the read tasks FIFO crosses a certain threshold, a request for more work is posted into the input FIFO of the task injector, who wakes up and fetches more work. At the end of a session, the task injector pushes an end of work token in all FIFOs. This token is used to implement a graceful shutdown of the session.

On the opposite end of the pipeline, the reporting of the successful or failed file transfers is also sent in batches. For migrations, a batch of successful transfers is only sent after the data has been successfully flushed to tape. Files are flushed to tape in batches in order to achieve the maximum throughput of the drive [6]. In the case of recall, they are also batched by an arbitrary amount (identical to the migration flushing usually) to minimize network traffic and allow the stager to batch database updates.

2.6. Reading and writing to disk
The data can be read from disk using various protocols. The disk access classes provide an uniform interface for the disk tasks to use, as described in 3.5. The disk tasks are run by a pool of threads, draining a shared FIFO. This allows disk transfers to run in parallel, as long as memory and disk threads are available. The number of threads is configurable by the operator, and the default with which we run in production is 10 threads. This ensures making use of a few gigabytes of memory when accessing file of around 300 MB, the current average size in CASTOR. When migrating, the disk tasks send data and errors to the input FIFO of the corresponding tape tasks. When recalling, the disk tasks send the final file transfer results to the result packer.

2.7. Reading and writing to tape
Tape is a serial medium onto which CASTOR writes one complete file after another. Access to tape is therefore managed by a single thread. The tape thread is responsible for mounting the tape, checking the tape is in fact the correct one, positioning the tape, validating the position of the tape, transferring data to and from tape and memory, and unmounting the tape at the end of the session. The timing statistics of the tape thread are used as the primary performance indicator for tape operators.

The data structures on tape have been left unchanged so tapes can be read and written interchangeably by both the current and previous software. The file format is based on ANSI user label [8] [9].
2.8. Tape session watchdog, logging and SCSI tape alerts
The tape thread can get stuck for various reasons, the most common being a stuck disk transfer. Not all disk-transfer protocols and their client libraries are guaranteed to time out and may block forever. The tape session process reports its advancement to the master process. When no tape block movement has been seen after a long period — currently 30 minutes — the tape session process is killed so the tape drive is free to accept another job.

In order to log statistics for a session irrespective of its fate, the session process regularly transmits transfer statistics. Then, the master process logs those statistics on behalf of the tape session process.

In addition to performance counters, the end of session log also includes error counts for the session, and SCSI tape alerts [5]. The tape alerts are hardware statuses reported asynchronously by the tape drive. They usually allow for finer grained diagnostics — whether a tape or drive is having a problem, as well as giving hints for preventive maintenance such as near end of life warnings for tapes.

End of session statistics including error counts are stored in an historical tape log database, where all tape mounts are tracked by the operations team.

2.9. Drive status and cleaner process
When a data transfer session completes, it reports the status of the drive — ready for a new mount or blocked by a stuck tape. When the session needs to be killed, a special cleaner process is forked to check the drive status and eject a tape if necessary. The status of the drive is then reported to the scheduling daemons.

3. Classes layout
The support for the features needed to implement the tape server were developed using a layered set of C++ classes. Each layer solves a particular problem and provides a simple interface to the layer above. Each layer is limited to the functionality needed for our project. For example, we did not attempt to create general purpose SCSI libraries.

The API of each layer was defined early in the project so that multiple developers could work on the layers in parallel.

3.1. SCSI access layer
A set of classes were developed to reduce the repetitive implementation of boilerplate code associated with SCSI development. These classes also provide a generic approach to endianness and status handling.

The SCSI data structures, which are defined at the bit level, were implemented using C bit fields. This allows clear coding when using them and automatic display in debuggers.

Particular care was taken to strictly adhere to the naming conventions of the SCSI specification in order to ease cross referencing [10] [11].

3.2. Tape drive layer
The tape drive support is implemented using a mixture of calls to the Linux st (SCSI tape) driver and direct SCSI access for functions not available in the st driver. The tape drive support is architectured as a partially implemented base class for functions available in all drives. Derived classes implement the functions that differ across hardware types, for example compression statistics [12] [13].
3.3. Tape format layer
This layer is responsible for the full life cycle of a tape mount. This includes validating the expected tape was mounted, positioning the tape, validating the tape is in the correct position, and enforcing adherence to the file format.

3.4. Tape library layer
There are two daemons to support tape library control. One for SCSI base tape libraries (tested in-house with IBM TS3500 and Spectra Logic libraries) [10] [11] [14] [15], and one for proprietary Oracle based libraries (tested with SL8500 libraries) [16]. The daemon supporting SCSI based libraries was kept as-is from the previous software stack. The daemon for Oracle based tape libraries was created to allow for the optional deployment of support for such libraries.

Both tape library daemon use a unified client library.

3.5. Disk access layer
The disk access layer is implemented as an abstract class defining the interface needed for the tape server, and a set of concrete classes implementing this interface for each of the supported protocols. Instances of the concrete classes are instantiated by a factory. This factory chooses the appropriate concrete class based on the file URL and daemon configuration. Implementing support for a new disk transfer protocol simply means implementing a new concrete class and updating the URL parsing in the factory.

The currently supported disk transfer protocols are RFIO and XRootD [17]. Support for RSA signatures was required in order to implement the disk layer. This signature was initially implemented using OpenSSL [18], but during memory leak validation (see 4.2), many problems arose. The implementation was then successfully switched to the Crypto++ library [19].

3.6. Threading
Necessary threading primitives were implemented to wrap the pthread library and POSIX semaphores with a more convenient C++ interface. The design of the threading layer was inspired by the threading primitives of Qt [20].

The inter-thread FIFOs were implemented as templated classes relying on STL containers, and semaphore and mutex classes provided by the threading layer. The overall design of the FIFOs was focused on code maintainability and ease of use. This design is simple enough to grant a local re-implementation. The resulting performances were more than adequate given the low performance requirement of 2 kHz data block flow rate.

3.7. Exceptions
The tape server software makes extensive use of C++ exceptions. The CASTOR exception classes were extended for the development of the tape server. Automatically generated stack traces were added to ease debugging. The backtrace call from the cxxabi.h header of gcc [21] was used.

3.8. Process communication protocols
Several new protocols were devised for the tape server: a protocol to communicate from the session process to the master process for statistics propagation, and the protocol to the new daemon for controlling Oracle tape libraries. In both cases we used ZeroMQ [22] as the transport and Google protocol buffers [23] for the payload serialization. For the rest of the communication, legacy C libraries for communicating with existing CASTOR components were reimplemented in C++. This reimplementation standardised previously heterogeneous solutions for error handling and propagation. The more recent parts of CASTOR were simply reused.
4. Development time line and methodologies
The development of the new tape server started on 31\textsuperscript{st} of May 2013 and concluded with the final deployment to production in mid January 2015. Work started with the independent development of the SCSI, drive and tape file layers, and the prototyping of the memory management. The initial development was done outside of the CASTOR tree in order to test new techniques and methodologies. Once validated, code was then integrated back into the CASTOR sources. This included moving the CASTOR build system from an in-house version of IMake to CMake [24].

4.1. Unit testing
Unit testing was used extensively during development. The Google test [25] and Google mock [26] frameworks were used. This provided fast error detection, and the ability to easily run test cases in a debugger, as opposed to requiring a debugger to be attached to a fully running system. The unit tests are executed by the continuous integration system of CASTOR in order to prevent future regressions. The system used by CASTOR is TeamCity [27].

4.2. Memory leaks and race conditions detection
Unit tests were automatically run through Valgrind [28]. This helped to find race conditions using helgind and memory leaks detection using memcheck.

4.3. Validation and deployment
Beta testing with physical validation tapes started at the end of 2014. This included a two week run during the end of year closure of CERN. 58 PB were successfully transferred between the final deployment in mid-January 2015 and the end of March 2015 (see figure 2).

5. Future improvements
With this new tape server structure, we now have a tape server that exercises the full performance of the underlying tape drive. The new software structure can easy accommodate new disk client libraries. The modular design will also facilitate future enhancements and maintenance.

We intend to add support for SCSI Stream Commands-4 logical block protection [29], which will improve the data writing reliability by adding extra checksum protection from the tape server memory all the way to the tape medium.

Support for Ceph [30] is being added to the disk based staging area of CASTOR in order to improve the performance of file transfers between disk, tape and users. The introduction of Ceph will also provide an off-the-shelf solution for data replication and self repair. This will replace the current use of RAID. This will stripe individual files over several disk servers, ensuring an even disk performance without the need for explicit disk scheduling.

During LHC run 1, the experiments recorded their raw data to the disk pools and tapes of CASTOR, with analysis running on EOS [31]. For run 2, a new trend is coming where the data goes straight to EOS, and one gets transferred to CASTOR for custodial copy recording to tape. We intend to leverage the flexibility of the tape server software to add support for direct EOS access with the addition of a new request queuing system. This will eliminate the need for the disk staging area of CASTOR in this scenario.

Figure 2. Data volume transferred
References
[1] CASTOR homepage http://cern.ch/castor
[2] Kruse D F 2013 The repack challenge Jour. of Phys.: Conf. Ser. 513 042028
[3] Cancio Melia G et al. 2015 Experiences and challenges running CERN’s high-capacity tape archive Unpublished paper presented at CHEP 2015, Okinawa
[4] Baud J-P et al. 1991 SHIFT, the Scalable Heterogeneous Integrated Facility Proc. of the Int. Conf. on CHEP ’91, Univ. Acad. Press, Tokyo pp 571-582
[5] Tape Alert v. 3.0 specification ftp://www.t10.org/t10/document.02/02-142r0.pdf
[6] Murray S et al. 2012 Tape write-efficiency improvements in CASTOR Jour. of Phys.: Conf. Ser. 396 042042
[7] Nikolaidis F et al. 2014 Transaction aware tape-infrastructure monitoring Jour. of Phys.: Conf. Ser. 513 032070
[8] ANSI user label format for tape in IBM documentation http://www-01.ibm.com/support/knowledgecenter/SSLTBW_1.12.0/com.ibm.zos.r12.idam300/labdef.htm
[9] CERN Internal documentation for tape file format http://it-dep-fio-ds.web.cern.ch/it-dep-fios ds/Documentation/tapedrive/labels.html
[10] SCSI T10 Standards http://www.t10.org/pubs.htm
[11] Hackipedia’s archive of SCSI drafts http://hackipedia.org/Hardware/SCSI/
[12] IBM enterprise tape drives http://ibm.com/systems/storage/tape/drives/
[13] Oracle StorageTek T10000D Tape Drive http://www.oracle.com/us/products/servers-storage/storage/tape/ storage/t10000d-tape-drive/overview/index.html
[14] TS3500 Tape Library SCSI Reference http://publibb.dhe.ibm.com/epubs/pdf/a3295612.pdf
[15] Spectra Tape Libraries SCSI Developers Guide https://support.spectrologic.com/documentation/user guides/tape-scsi-developer.pdf
[16] Automated Cartridge System Library Software Client System Component Developer Toolkit Downloads http://www.oracle.com/technetwork/developer-tools/acs-cl-sci-toolkit-1703520.html
[17] XRootD: a generic suite for fast, low latency and scalable data access http://xrootd.org
[18] OpenSSL: The Open Source toolkit for SSL/TLS https://www.openssl.org
[19] Crypto++: Library: a free C++ class library of cryptographic schemes http://www.cryptopp.com
[20] Qt: Cross platform application and UI development framework http://www.qt.io/
[21] GNU C Library: Backtraces http://www.gnu.org/software/libc/manual/html_node/Backtraces.html
[22] Zero MQ transport layer http://zeromq.org
[23] Protocol buffers: Google’s mechanism for serializing structured data https://developers.google.com/protocol buffers/
[24] CMake, the cross-platform, open-source build system http://www.cmake.org/
[25] googletest: Google C++ Testing Framework https://code.google.com/p/googletest/
[26] googlomock: Google C++ Mocking Framework https://code.google.com/p/googlomock/
[27] TeamCity: Continuous integration for everybody https://www.jetbrains.com/teamcity/
[28] Valgrind a GPL’d system for debugging and profiling Linux programs http://valgrind.org/
[29] Butt K 2007 Tape end-to-end data protection proposal http://www.t10.org/fltp/t10/document.07/07 37r0.pdf
[30] Ceph distributed object store and file system http://ceph.com
[31] EOS homepage http://cern.ch/eos