

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT



MASTER THESIS

Enhancing the low-level tape layer of CERN Tape Archive software

Scientific Advisor(s):

Sl.dr.ing. Laura Cristina Ruse
As.drd.ing. Laura Vasilescu
Ing. Eric Cano

Student:

Cristina-Gabriela Moraru

BUCHAREST

September 2017

Table of Contents

1. Introduction	4
2. State of Art	5
2.1 Hardware	5
2.2 SCSI	6
2.3 Tape Virtualization Software	7
2.4 CERN Tape Infrastructure	7
3. Recommended Access Order	10
3.1 Sequential Order vs. Recommended Access Order	10
3.2 RAO in CERN Tape Archive Software	12
4. Replacing the usage of st driver with sg	19
4.1 Stateless commands	23
4.2 Stateful commands	25
4.3 Special commands	27
5. Results	28
5.1 Speedup of adding RAO support	28
5.2 Status of replacing st with sg	31
6. Conclusions and Further Work	32

1. Introduction

CERN data center holds the largest data archive in the HEP domain which gathers measurements collected from the experiments, analysis results, simulations etc. The archive has reached 180 PB and the predictions indicate that will increase with approximately 100 PB per year[1]. Given that a considerable part of this data is rarely accessed, the tape storage, despite the bias of being an old-fashioned technology, is the best storage solution in terms of space and cost. CERN owns a large tape infrastructure spread across both Tier-0 and Tier-1 levels of the WLCG (Worldwide LHC Computing Grid). The CERN tape archive software is composed of CASTOR (**C**ERN **A**dvanced **S**TORage manager) and its successor CTA (**C**ERN **T**ape **A**rchive) [2]. CASTOR is the system currently in use, developed to manage the file transfers between hard disks and tape drives but, due to hardware evolution, it has become limitative in terms of efficiency and support for new disk storage features. CTA is a prototype which aims to redesign CASTOR's architecture by implementing a direct disk-to-tape communication. However, a challenge that both projects encounter is mediating the transfer between two different technologies: disk and tape storage. While accessing disk files seems instant from a user's point of view, access time to tape files is significantly higher, raising to several minutes for a series of files. If large volumes are being traded, this issue becomes a serious performance bottleneck and optimizations are required in order to balance the numbers.

This thesis introduces two low-level tape access optimizations: first, adding support for the Recommended Access Order (RAO). The RAO is the order that corresponds to the minimum access time for a series of files. Some tape drives are able to compute the RAO for a series of files, in order to optimize further accesses. This thesis will present our solution for including RAO in the file retrieval operations in CERN tape archive software in order to benefit from the reduced reading time. The second optimization is replacing the usage of the SCSI tape specialized driver *st* with the generic SCSI driver *sg*. The purpose of the latter one is to eliminate the undesired status caching and the unnecessary permission checking mechanism (Linux capabilities) performed by the *st* driver for a more straightforward and robust implementation.

Further, our solutions for the two optimizations will be presented. Chapter 2 will offer a generic view over the tape hardware infrastructure and existing software solutions, Chapter 3 will explain the difference between the Recommended Access Order and the standard Sequential Order and how the CERN tape archive software implementation has been extended to support RAO. Chapter 4 details the replacement of the calls to the *st* driver with calls to the *sg* driver. Chapter 5 presents the speedup and advantages obtained as a result of our optimizations and finally, Chapter 6 offers the perspectives of further development.

2. State of the Art

This chapter will briefly present the background about some key hardware concepts underlying the objective of this project, will describe the SCSI interface and the existing low-level storage technologies.

2.1 Hardware

Magnetic tape storage is a system for storing digital information on a magnetic tape. The tape is held rolled on one or more reels, inside plastic enclosure called generically cartridge (Fig. 1). According to manufacturer and version, cartridges vary in tape length, technology and storage capacity. Current versions, such as Linear Tape Open (LTO) Ultrium, can support up to 15TB of data[3]. Although this may seem the key parameter, a shorter tape length leads to a lower access time for file access. Also, wide data centers hold usually up to thousands of cartridges, therefore the selected technology must provide scalability and high-availability. A dedicated device called tape drive (Fig. 2) is responsible with performing direct tape access such as rewinding, positioning by block or file, reading and writing. Additionally, some tape drives provide more advanced functionalities such as logical block protection, compression or encryption[4]. A modular library system (Fig. 3) is a solution of type Capacity-on-Demand for gathering a large number of cartridges and drives in a rack environment that allows linear growth. For example, Oracle StorageTek SL3000[5] offers the possibility to scale from 200 to almost 6000 cartridge slots and from 1 to 56 tape drives. Finally, the media changer (commonly known as “picker”) is a dedicated device able to change the location of a cartridge from the rack to the drive and vice-versa - processes called mounting (loading) and unmounting (unloading).



Fig. 1 Tape cartridges[6]



Fig. 2 Tape drive[7]



Fig. 3 Modular library system[5]

2.2 SCSI

Small Computer System Interface (SCSI) is a set of standards for physical connection and transfer of data between computers and peripheral devices and it is the default interconnection protocol for hard-disks and tape drives.

In Linux Kernel, the SCSI low-level implementation is supported by four drivers: *sg* (SCSI Generic), *st* (tapes), *sd* (disks), *sr* (CDROMs and DVDs). The first two will be further detailed, as they are relevant in the development of this project.

The *st* driver[8] is the tape specialized higher-level driver which can recognize 32 tape drives. For each attached tape drive it creates 8 character devices: a rewind (which automatically rewinds the tape after any read/write access) and a non-rewind (leaves the tape in the position of the last access) version for each of the four modes emphasised in Table 1.

Table 1

Tape drive modes[9]	
Mode	Description
1	blocksize=0 compression=1
2	blocksize=1024 compression=1
3	blocksize=0 compression=0
4	blocksize=1k compression=0

The *sg* driver[10] is the generic SCSI driver underlying the other three and supports any SCSI device. It is recommended to use the specialized drivers when possible but sometimes the generic one permits a more fine-grained control over the transferred data. Also, this driver handles operations to the media changers. The *sg* driver implements the direct tape communication in three phases:

- A request is received from the user and the required resources, if applicable, are reserved (ex: kernel buffers for indirect IO etc). Sometimes, additional data is transferred into the kernel buffers. The request is queued for execution.
- An interrupt is received when the request is completed. If, for example, the request was a READ, then the data can be found in the kernel buffers (indirect IO) or in the user buffers (direct mmap-ed IO)
- The user performs a call to fetch the results

There is, however, a synchronous type of query, in which the requester asks for the result and the thread blocks until it is received.

The communication protocol is defined by the data format *sg_io_hdr* which is sent along within *ioctl* system calls. This structure indicates the command to be executed by the SCSI device along with the correspondent parameters and offers some buffers for potential error reporting.

Additionally, the *sg* driver offers a virtual file system entry */proc/scsi/scsi* where all SCSI devices currently connected to the bus are listed along with their descriptions: device type, model name, vendor, SCSI channel and ID. This is the source of the *lsscsi* userspace

Linux tool which is capable of filtering and displaying the information exposed by the driver in a more human readable format. Also, the directory */proc/scsi* contains an entry for each SCSI driver used by the system with files specific to the SCSI controllers using that driver.

There are two userspace command-line tools which make use of the above listed driver functionalities and allow tape management:

- **mt** - performs a given operation on the tape drive; supported operations are rewind, eject, repositioning by block or file
- **mtx** - controls single or multi-drive SCSI media changers (tape changers, autoloaders, tape libraries, optical media jukeboxes etc.); possible operations are status interrogation, inquiry, loading, unloading etc.

2.3 Tape Virtualization Software

A virtual tape library (VTL) is a technology for data storage virtualization, used for backup and recovery purposes. A VTL simulates the existence of tape libraries or tape drives over another storage component, usually a hard disk. Among the advantages of tape virtualization we can name storage consolidation and faster data restore process by eliminating streaming incompatibilities that occur when combining two different technologies (disk and tape) and by optimizing the transfer speeds. mhVTL is one example of VTL and its most important components are the following:

- kernel module: **mhvtl**
- user-space daemons: **vtltape**, **vtllibrary** - simulate the SCSI device(s)
- utility commands: **mkmtape**, **vtlcmd**

mhVTL offers most of the features that real tape libraries and drives provide. However, from a user's point of view, the difference in timing for performing some operations is significant comparing to original technologies: for example, mounting a real tape lasts between one and three minutes in contrast to mhVTL which performs the virtualized mounting instantly.

2.4 CERN Tape Infrastructure

The large volume of physics data gets parcelled up and sent via a great global network called Worldwide Large Hadron Collider Grid (WLCG), which is currently the world's largest grid. WLCG is a distributed infrastructure organized in tiers as follows:

- **Tier 0:** CERN Data Center from Geneva, Switzerland and the Wigner Research Centre for Physics in Budapest, Hungary
- **Tier 1:** 13 computational centers in the world
- **Tier 2:** universities and scientific institutes
- **Tier 3:** university clusters or local PCs

Most of the processing, event reconstruction and simulations are performed in Tiers 0 and 1, where also storing large amounts of raw and processed data is needed. As visible in Fig. 1, each of this levels owns a tape infrastructure for this purpose. Despite the convenience of a large storing capacity, the latency on access limits the performance both on regular tape usage and on large-scale *repacks* [11] (migrations of the entire stored data when switching to a newer tape generation, data recovery or defragmentation etc.).

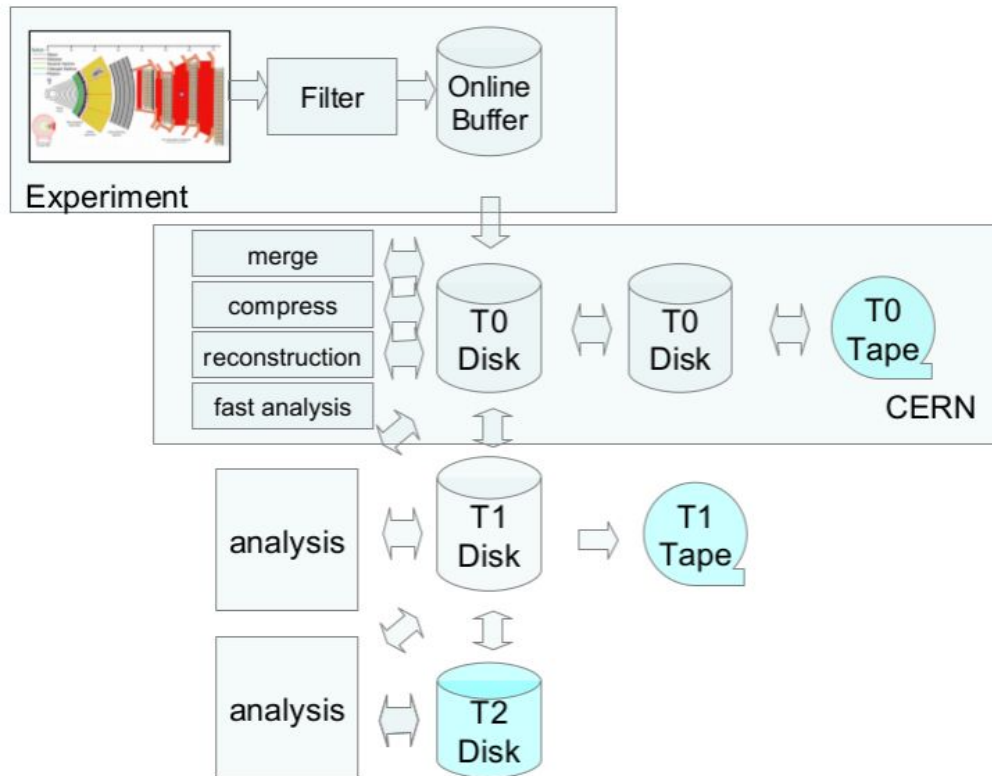


Fig. 1 High-energy physics data flow[12]

The CERN Advanced STORage manager (CASTOR) [13] is a hierarchical storage management system (HSM) used to manage the file transfers from hard disk and tape drive (migrations) and in reverse (recalls). Currently, CASTOR is hosting around 180 PB of physics data and it represents the piece of resistance in the big-volume storage on the tape infrastructure at CERN. CASTOR is composed of a disk cache and a tape backend [14]. A vital component is the stager, which manages the internal disk-based cache used as a staging area for tape drives. Until recently, this component has been an effective solution of serializing multi-stream, block-oriented disk data before accessing a single-stream, file-oriented tape infrastructure. The backend is composed of a multi-threaded tape server with a pipelined architecture in order to minimize blocking [15]. Separate threads are created for asynchronous instruction fetch, data read, data write and result reporting. However, since the capacity:price ratio of the disk-based storage has increased in the latter years, the stager managing tape transfers became unnecessary. Moreover, in the meantime,

disk storage solutions have evolved and now provide the possibility to split a file on multiple disk servers which is highly convenient when designing a distributed architecture. However, the objective is to shift from on demand, per-file accesses to efficient bulk data transfers enabling large volumes to be retrieved in aggregations thus taking advantage of the stream-based nature of tape media [14]. This mechanism would also eliminate the disk-based caching bottleneck.

Consequently, CERN Tape Architecture (CTA) [2] is a natural evolution of CASTOR and is currently a prototype of tape software developed to transfer data directly between the remote disk servers and tape drives [16]. In CTA, the role of the stager is replaced by an advanced scheduling system, able to handle multiple data requests from the disk drives with a single tape stream managed by the tape drive. Conceptually, having a file striped over multiple disk servers requires all of them transferring data in parallel, thus using multiple disk-drive streams to service a single tape stream [16]. Other novelties include mechanisms for grouped migration policies and priorities and a new tape format optimized for transferring data in aggregations. One example of problem CASTOR encountered was that statistics show an average of 1.5 user files accessed per tape mount. Considering that the time required to mount a tape is between 1 and 3 minutes, file access implementation is suboptimal. The problem arises from the fact that CASTOR is serving the request immediately, thus mounting the tape almost every time a file is requested. CTA increases the number of files accessed per tape mount by prefetching and queuing while gathering enough requests to warrant the mount. All changes aim to decrease the access time to tape files and to ensure robust operation of the experiments' workflows (raw/processed data transfer, reconstruction, import/export to/from other sites across the WLCG) [17]. However, the impact on performance is highly noticeable when performing large-scale transfers between disk and tape: for example, when repacking data from an older to a newer generation of tape media. Summarily, CTA eliminates the redundancy between the stager and the frontend disk manager (EOS), adds a thin layer on top of the original CASTOR tape server, implements a preemptive scheduler in order to use the tape drive at full speed at any time and diminishes the networking between components [2]. CTA is currently a prototype since it needs to be investigated its ability to provide tape access service without an internal disk-based staging area while reducing application complexity, easing code maintenance and reducing time overheads and improving tape efficiency.

3. Recommended Access Order

This chapter will present the Recommended Access Order in tape context, will explain how is it different from the Sequential Order and, will describe in detail its implementation in CERN tape software from the low-level SCSI format to the interaction of higher level entities in a multithreaded environment.

3.1 Sequential Order vs. Recommended Access Order

This chapter will offer the basic knowledge about the magnetic tape's design and will explain what is the Recommended Access Order and the advantages that it brings in comparison with the standard Sequential Access Order. First of all, we will offer a quick view of the hardware design in order to give a better understanding of the higher level concepts that will be presented further.

A recording volume is spanned on its length of long parallel tracks that split its width into multiple tape bands between the two endpoints beginning-of-medium (BOM) and end-of-medium (EOM) as shown in Fig. 2. In *serpentine recording* [18], the tracks are traversed one track group at a time (here, considering a track group as one physical track). The route of data access is as follows: first track is proceeded from the BOM to EOM and then the head is shifted above the next track and the move continues in the reverse direction and so on.

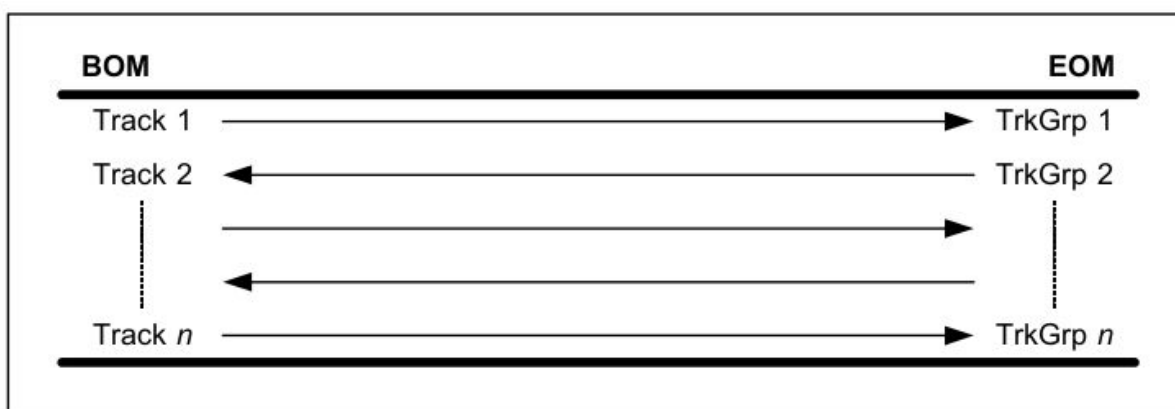


Fig. 2 Serpentine recording[18]

In order to explain the difference between sequential order and recommended access order, we propose following example: supposedly, random tape files are requested and their placement on the tape is emphasised in Fig. 3. Files are requested in the order given by the numbers. In sequential order, the files will be accessed in the order of their request, thus following the red path, performing time-consuming unnecessary shifts between BOM and EOM.

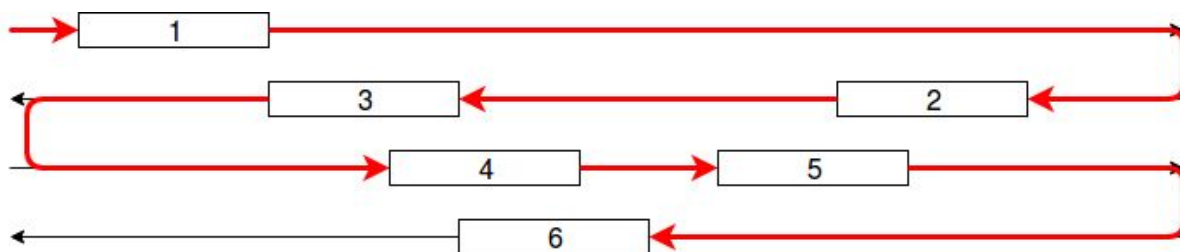


Fig. 3 Sequential Order

However, the situation is a best case scenario which rarely happens in practice: the files are placed consecutively on the tape. If files placed at random positions are recalled, additional slides from BOM to EOM are performed which is extremely costly given that it takes several seconds (30s with the current media) to slide between both ends. In a worst case scenario, 60 minutes could be necessary to access 100 files. The issue arises from the fact that each file is accessed independently which is suboptimal. In CERN tape archive software, there can be hundreds of queued jobs at a time so the relative position between them should be turned into account. The reading head can be repositioned on another track at any time, not just at one of the ends. The repositioning time could be minimized if the proximity of the requested files is taken into consideration.

The Recommended Access Order (RAO) is the order that corresponds to the minimum recall time for a given set of files. Some tape drives have implemented the RAO functionality, namely are able to respond to RAO queries: receive a list of files compute the RAO taking into consideration file sizes and relative positions but also drive internal parameters such as: reading speed, repositioning speed, other time overheads. For the case described above, the RAO is shown in Fig. 4 where it can be observed that the number of unnecessary shifts is drastically reduced.

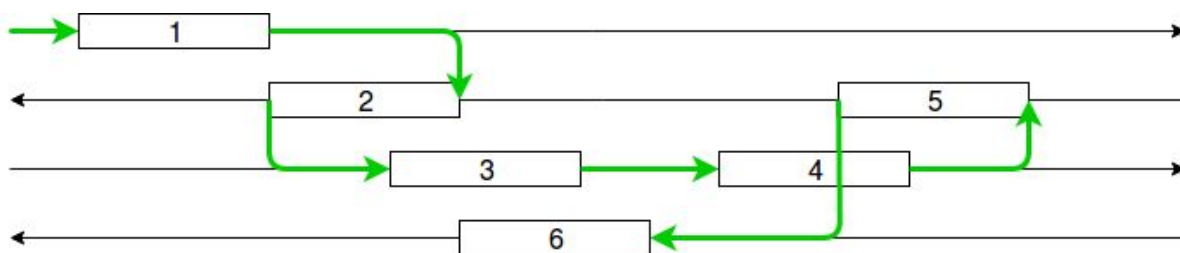


Fig. 4 Recommended Access Order

3.2 RAO in CERN Tape Archive Software

The CERN tape archive software is the backend of the storage system and is responsible for performing direct transfers with tape libraries (recalls - tape to disk - and migrations - disk to tape). This project aims to develop CERN tape archive software to perform a RAO reordering before the actual transfer, in order to minimize the access time. Although there is a series of improvements CTA brings in comparison to its predecessor CASTOR, the implementation of RAO at low-level SCSI layer as well as in the threading model was similar for both projects.

This goal has two components: first, to extend the low-level tape layer in order to support RAO queries and second, to update the execution flow to perform a RAO query before the files are recalled from tape.

3.2.1 Extending the Drive Model

A tape drive is represented in the tape archive software by a common class `DriveGeneric` and a series of specialized derivative classes for each particular tape drive model from the data center. Since RAO is a functionality provided by all physical tape drives available, it has been implemented in the `DriveGeneric` and an exception is thrown by the `mhvtl` associate class because RAO is not supported by this tape virtualization library. All SCSI functions are provided making use of a mixture between the `st` and `sg` drivers.

Tape drives are able to calculate RAO for a series of files, in order to optimize further multiple block recall and retrieval times. There are 3 phases of a RAO query:

1. Query for User Data Segment (UDS) limits. An user data segment references a contiguous sequence of logical objects (ex: logical blocks or files) within a recording partition. The response for this command (Fig. 5) contains both the maximum number of UDS supported by the drive in a single request (16 bits) and the maximum size that an UDS descriptor can have (16 bits).

Bit Byte	7	6	5	4	3	2	1	0
0	(MSB)	MAXIMUM UDS SUPPORTED						
1								
2	(MSB)	MAXIMUM UDS SIZE						
3								(LSB)

Fig. 5 User Data Segment limits page [18]

2. Send a Generate-RAO request to the device server along with a buffer with a serialized list of UDS corresponding to the required files to be reordered; the number of files sent for reordering should not exceed the values obtained by the query from the first step. The UDS layout (Fig. 6) for a file in the list contains 3 key parameters: a

unique identifier and the IDs of the start and end blocks of the file as recorded on the tape. The ID of the start block is part of the file meta-information in the tape software but the ID of the end block must be deducted from the start block ID and file size (also part of the meta-information).

Bit Byte	7	6	5	4	3	2	1	0
0	DESCRIPTOR LENGTH (1 Eh)							
1								
2	Reserved							
...								
4								
5	UDS NAME							
...								
14								
15								
16	PARTITION NUMBER							
...	(MSB)	BEGINNING LOGICAL OBJECT IDENTIFIER						(LSB)
23								
24	(MSB)							
...		ENDING LOGICAL OBJECT IDENTIFIER						(LSB)
31								

Fig. 6 User Data Segment descriptor format [18]

The CASTOR tape format (Fig. 7) is composed for 3 header blocks and one separator block (*tm*) prior the actual data and, symmetrically, 3 end-of-file blocks and a separator as trailer. The number of the blocks occupied by the file data is calculated from the file size, knowing the size of the block (235 kB).

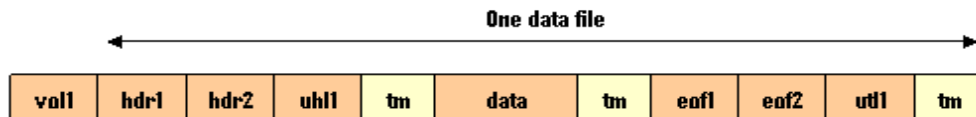


Fig. 7 CASTOR tape file format

Therefore, to calculate the ID of the end block, we deduct the following formula (1). Given that files can have arbitrary file sizes, the number of blocks is ceiled to cover all potential situations.

$$blockEnd = blockStart + 8 + \frac{fileSize + blockSize - 1}{blockSize} \quad (1)$$

3. Send a Receive-RAO command to collect the list of reordered files. The format of the response, shown in Fig. 7, contains list of UDS sent by the previous command, ordered by the estimated recall time.

Bit Byte	7	6	5	4	3	2	1	0
0							RAO PROCESS	
1							STATUS	
2	Reserved							
3	Reserved							
4	(MSB)							
...	RAO DSCRIPTOR LIST LENGTH (n-7)							
7	(LSB)							
RAO user data segement descriptor list								
8	RRAO - user data segment descriptor [first] (see table 82)							
⋮								
n	RRAO - user data segment descriptor [last] (see table 82)							

Fig. 7 Recommended Access Order List [18]

All steps are performed via ioctl system call, through the sg driver.

3.2.2 Extending the threading model

This chapter will discuss the second milestone of this optimization, namely enforcing the execution flow to perform a RAO query before the actual recalls in order to benefit from the waiting time decrease. The threading model is similar in both CASTOR and CTA and will be presented in continuation along with our solution.

The tape server is the entity responsible for managing the tape access and its highly-multithreaded implementation will be further presented. Tasks such as asynchronous instruction fetch, result reporting, data read and write run in separate threads in order to minimize blocking. The main process is the data transfer session which, after performing the initializations, fetches a set of jobs from the client and starts a thread for each specialized function. Afterwards, several threads are created among which we name:

- TapeReadSingleThread
- RecallTaskInjector
- DiskWriteThreadPool
- TapeServerReporter ||
- RecallReportPacker || => execution monitoring and error reporting
- RecallWatchDog ||

The execution flow for the recall of a series of files is presented in Fig. 8 and will be further described. After a fixed size number of recall jobs is prefetched, two specialized storage management threads are created. Given that the tape is a serial medium, a single thread - the *TapeReadSingleThread* - manages operations such as tape mounting and unmounting, positioning, transferring data to / from memory and other tape checks. The approach of having one thread handling the tape mount session eases the detection and killing of stuck sessions and offers consistent timing statistics for performance analysis. The *TapeReadSingleThread* is built up over the RAII (Resource Acquisition is Initialization) programming idiom where resources (in this case, the tape drive) are acquired during the initialization and freed in the destructor. In contrast, as disk storage supports parallel access, is managed by a pool of threads (*DiskWriteThreadPool*). The *RecallTaskInjector* manages higher-level client jobs that are queued and sent further for execution. For each file recall job, two simpler tasks are created: one for a tape read, managed by the *TapeReadSingleThread* and one for a disk write performed by the *DiskWriteThreadPool*. A synchronization method is implemented in order to notify the *RecallTaskInjector* when the tape read tasks are fewer than a certain threshold, therefore to request more jobs.

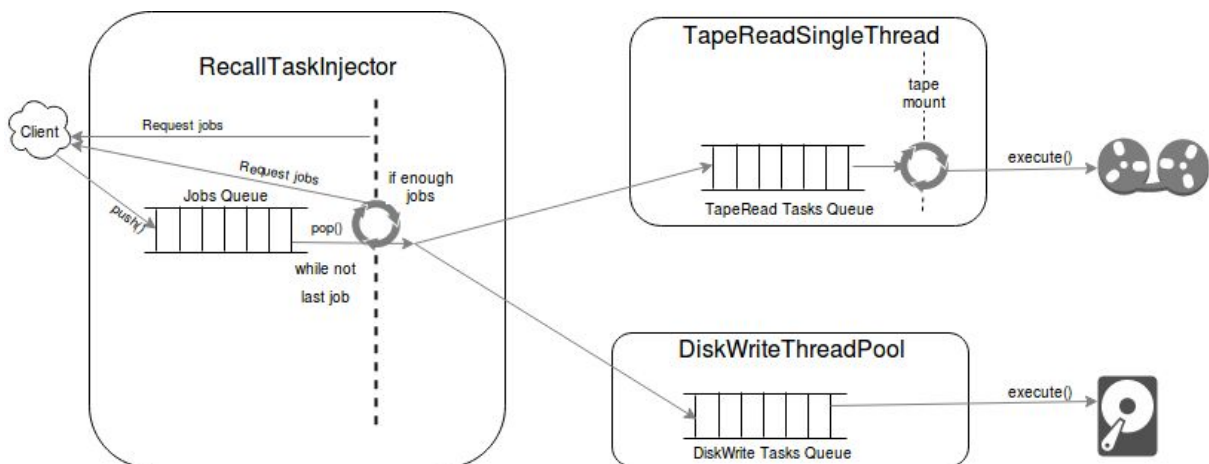


Fig. 8 File recall execution flow

The execution flow, presented in Fig. 8, is fully-pipelined in order to minimize blocking but one drawback of it is that it follows the FIFO principle for the files' recall. Consequently, the tape read tasks are created and executed sequentially, in the same order as they come from the client. As an optimization in this matter, the client already sorts the jobs in an increasing order of the file ID to avoid costly seek time on the tape. However, as proven in the previous chapter, the sequential access order is a suboptimal method of accessing tape files. Further, we present our approaches of extending the threading model to support drive-assisted Recommended Access Order along with the fully functional final solution.

The jobs are received from the client in sequential order and this fact is a constant in this project. Within this project, a new phase of reordering has been inserted between the job fetching and the creation of tape read / disk write tasks. Several approaches have been implemented and will be further discussed.

There is a direct one-to-one correspondence between the tape and disk tasks from the queues of the *TapeReadSingleThread* and *DiskWriteThreadPool* threads. Therefore, the RAO reordering must be performed at the job level in the *RecallTaskInjector* in order to create the tasks following the same principle and avoid altering the correct execution. To be noted that previously only one thread (*TapeReadSingleThread*) was managing the operations for the tape drive and for performing RAO query, the access to the drive must be shared with the *RecallTaskInjector*. The resource sharing happens in an initialization stage from the data transfer session, prior the start of the threads.

In our first approach, the access to the the drive is owned alternatively by the two threads. After the tape mount, the *TapeReadSingleThread* passes the access to the drive to the *RecallTaskInjector* which performs a RAO query for a set of files to recall. When the optimal file order is determined, the tape-read and disk-write tasks are created for execution and the access to the drive is passed back to the original owning thread. When the tasks queue of the *TapeReadSingleThread* is emptied, the *RecallTaskInjector* is notified and given access to the drive to prefetch and reorder a new set of files. The synchronization for the shared resource access is done using the C++11 promise-future feature. A visual representation of this approach is offered by Fig. 9.

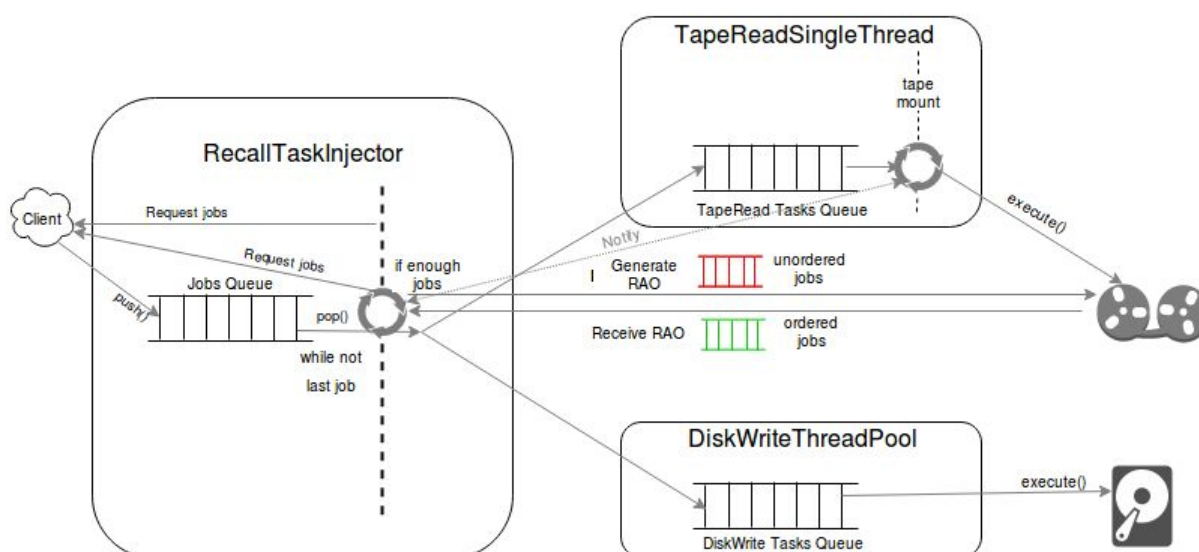


Fig. 9 Approach 1: Alternative access to the drive

This approach would ensure that all jobs will be grouped into batches and sorted according to RAO but the downfall is that it serializes the execution. The highly pipelined architecture that minimizes blocking is changed to a design in which entities depend on each other introducing delays. Moreover, the RAO is brings relevant speed improvements when the batch to be red is above a certain threshold.

The second approach aims to eliminate the problematic alternative access to drive and to perform the RAO query one time, prior all the recalls, for a set of files as large as possible. This solution, presented in Fig. 10, takes advantage of the big waiting time (1-3 minutes) required for a tape mount and enforces the *RecallTaskInjector* to continuously gather recall jobs until gets the notification from the *TapeReadSingleThread* (via the promise-future mechanism) that the tape is mounted and is able to perform the RAO query. After the RAO query is performed, RAO is disabled and the normal execution flow is restored.

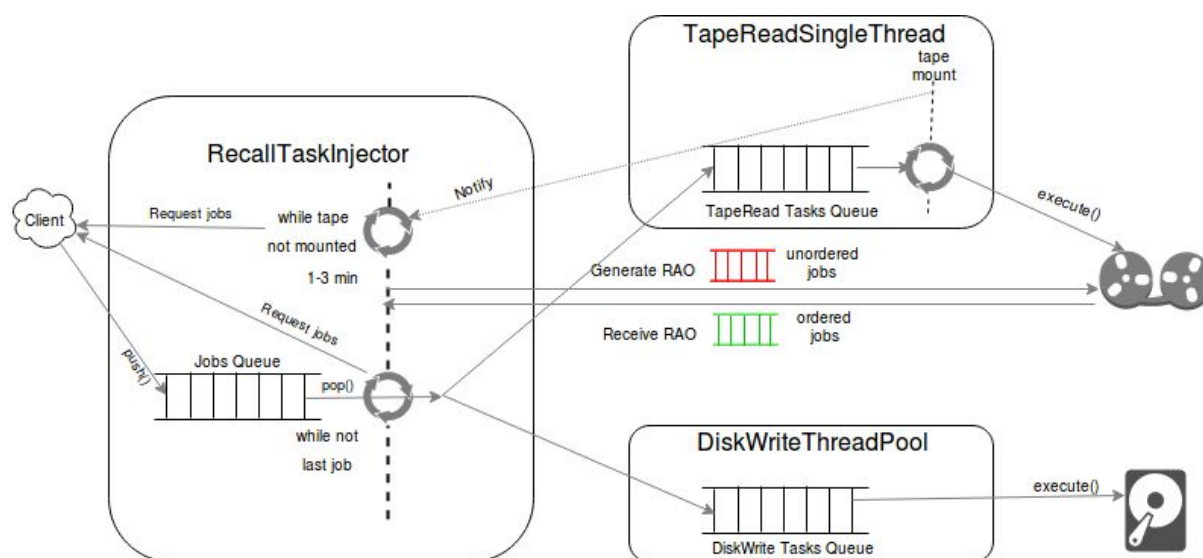


Fig. 10 Approach 2: Reorder files fetched during tape mount

The maximum number of files that can be RAO sorted in one query is a parameter which depends on the tape drive and is discovered via the UDS limits command performed a priori. The maximum user data segments supported (each one corresponding to a file) is a hardware specification which varies among manufacturers but is usually around few thousands. The drive is queried for the UDS in the RAO initialization part of the data transfer session, before the specialized threads are started, thus before the tape is mounted by the *TapeReadSingleThread*. Although the UDS limits are known by the tape drive, performing an UDS query, namely an `ioctl` call to the `sg` driver, when there is no tape in the drive causes the drive to return an error "No medium found". Without knowing the maximum UDS supported a priori, the *RecallTaskInjector* does not have a threshold of files to accumulate while waiting for access to the tape drive. If too few files are available for reordering, the optimization potential of the RAO functionality is not fully exploited. If too many files are queued, the drive will not be able to reorder all of them in one query. In CTA, when the client sends jobs to the *RecallTaskInjector*, it deletes them from its internal queue. Therefore, in case of a read session failure, all unexecuted jobs would have to be copied back into the client's queue for retry which is highly inefficient. Moreover, the continuous polling for new jobs performed during the tape mount can unnecessarily slow down the client's database with an overflow of requests.

The conclusions gathered during the first two approaches led to the final solution of this milestone. The best approach is to perform the Recommended Access Order query only once, with the first batch of files and then resume to the regular execution flow. In the optimal case, the number of files reaches the maximum supported UDS obtained right after the tape mount. A graphical representation of the final solution is visible in Fig. 11.

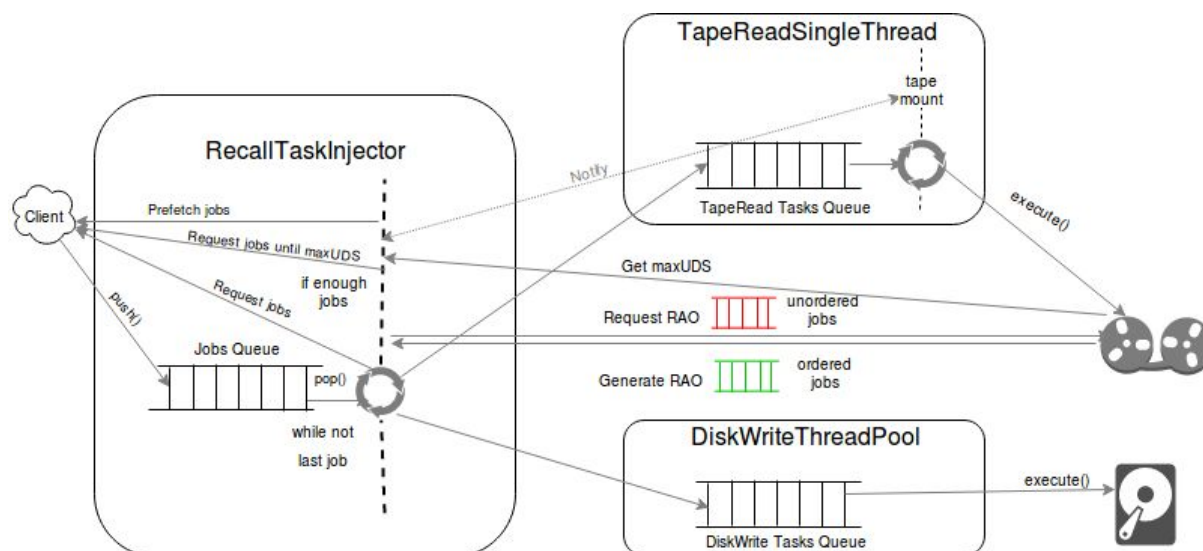


Fig. 11 Final solution: Reorder a single batch of maxUDS files

The current execution flow is the following: the *RecallTaskInjector* fetches a pre-established number of jobs before getting the notification for the completion of the tape mount (via promise-future mechanism). When it gets access to the drive performs an UDS query to discover the maximum number of files that can be sorted in one query. If necessary, requests more jobs from the client to reach that number. When the recall jobs batch is formed, it performs a RAO query on the jobs and sends them for execution. The recommended access order is disabled and the next files to recall will be accessed again in sequential order.

This approach complies with the highly-pipelined architecture of the project and introduces only one extra job request from the client, thus not altering the execution of other entities. This solution has been tested and Chapter 4 will present the results and observations.

4. Replacing the usage of *st* driver with *sg*

The low-level tape access implementation uses a mixture of calls to the *st* driver (SCSI tape) and direct SCSI access to the *sg* (SCSI generic) driver for functionalities that are not available in *st*. In theory, it is indicated to use the specialized driver for a particular device rather than the generic driver for the bus the device connects to. Using the *st* driver brings a series of advantages: from the developer's point of view, the system calls exposed by the *st* are simpler and easier to use because they require less parameters and are more intuitive. The *st* driver internally creates the SCSI command and auto-populates some of the command's parameters requiring from the user only one that can not be deducted. Also, it provides functionalities for querying over the tape status, which is not possible with the *sg* driver.

Although the above mentioned reasons might incline the balance in favor of the *st* driver, its usage brings a series of inconveniences. First, the status information is collected as a side effect of any other drive operation and returned when a request for it is received from userspace. In some situations, this metadata is cached and does not reflect the real tape status, requiring a device reopening to enforce the update. Another point of view taken into consideration is that the *st* driver performs additional security checks over the processes accessing its functionalities. The Linux capabilities[Reference] mechanism is a system for granting and checking permissions in the kernel in order to ensure the correct usage of the managed devices. Safely running the tape server as an unprivileged process causes it to be the subject of full permission checking based on its credentials such as effective UID and GID, group list etc. Therefore, specific permissions must be granted to this process in order to perform tape I/O operations or access special system files. However, the tape server instances in CERN tape archive software run in dedicated controlled machines and there is no threat of an unauthorized access to justify a permission based security system. Replacing the calls to the *st* driver with calls to the *sg* driver would eliminate this unnecessary feature and would simplify the tape server's current implementation and development.

This optimization aims to replace the calls to the specialized tape driver *st* with calls to the generic SCSI driver *sg* in order to gain more control over the commands sent to the tape drive, to gain more accurate status information over the device bypassing the caching performed by the *st* driver and to eliminate the unnecessary capabilities checking mechanism. The low-level drive abstraction of the CERN tape archive software supports a large series of operations over the tape media and the implementation is based both on the *st* and *sg* drivers. Table 3 shows precisely which functionalities are based on each particular driver. To be noted that all functionalities based on the *st* driver are part of the implementation of the generic tape drive whereas the *sg* driver is used in the implementation of the derived classes precisely for the flexibility of customizing the SCSI commands for drive's requirements. The goal of this milestone is to replace all calls to the *st* driver with calls to the *sg* driver.

Table 3

Driver usage of tape operations

st	sg
DriveGeneric::isTapeBlank	DriveIBM3592/DriveLTO::clearCompressionStats
DriveGeneric::setSTBufferWrite	DriveGeneric/DriveT10000/DriveMHVTL::getDeviceInfo
DriveGeneric::spaceToEOM	DriveGeneric::getSerialNumber
DriveGeneric::setSTFastMTEOM	DriveGeneric::positionToLogicalObject
DriveGeneric::fastSpaceToEOM	DriveGeneric::getPositionInfo
DriveGeneric::rewind	DriveGeneric::getTapeAlertCodes
DriveGeneric::spaceFileMarksBackwards	DriveGeneric::setDensityAndCompression
DriveGeneric::spaceFileMarksForward	DriveGeneric::setLogicalBlockProtection
DriveGeneric::unloadTape	DriveGeneric::getLBPIInfo
DriveGeneric::flush	DriveGeneric::setEncryptionKey
DriveGeneric::writeSyncFileMarks	DriveGeneric::clearEncryptionKey
DriveGeneric::writeImmediateFileMarks	DriveIBM3592/DriveT10000::isEncryptionCapEnabled
DriveGeneric::waitUntilReady	DriveGeneric::SCSI_inquiry
DriveGeneric::hasTapeInPlace	DriveT10000::getCompressionStats
DriveGeneric::isWriteProtected	DriveLTO/DriveIBM3592::getCompression
DriveGeneric::isAtBOT	DriveIBM3592/DriveT10000::getTapeWriteErrors
DriveGeneric::isAtEOD	DriveIBM3592/DriveT10000::getTapeReadErrors
	DriveGeneric::getTapeNonMediumErrors
	DriveIBM3592/DriveT10000::getVolumeStats
	DriveIBM3592/DriveT10000::getQualityStats
	DriveIBM3592/DriveT10000::getDriveStats
	DriveGeneric::testUnitReady

Table 3 presents all the functionalities based on the ioctl system call. Additionally, there are the read / write functions also have their kernel correspondence in the *st* driver. The functions presented in the table are the high level methods of the drive classes in CERN tape archive software and some of them map to the low-level SCSI commands follows.

Table 4

Mapping between high-level functions and SCSI commands

Function	SCSI command
spaceToEOM	SPACE
fastSpaceToEOM	
spaceFileMarksBackwards	
spaceFileMarksForward	
rewind	REWIND
unloadTape	LOAD / UNLOAD
writeSyncFileMarks	WRITE FILEMARKS
writeImmediateFileMarks	
flush	
isAtBOT	READ POSITION

As Table 4 shows, more functionalities map to the same SCSI command but not all functions have a corresponding SCSI command. The reason behind this is because the other functions access the meta-information cached by the *st* driver during the device usage while the tape drive is not queried.

Both drivers *st* and *sg* implement *ioctl* command but the function signature is the one which determines which of the kernel function will be called. The *st* driver exposes the *ioctl* signatures presented in Fig. 12 which is a simplified SCSI API built especially for tape media.

```
int ioctl(int fd, int request [, (void *)arg3]);
int ioctl(int fd, MTIOCTOP, (struct mtop *)mt_cmd);
int ioctl(int fd, MTIOCGET, (struct mtget *)mt_status);
int ioctl(int fd, MTIOCPOS, (struct mtpos *)mt_pos);
```

Fig. 12 *ioctl* formats exposed by the *st* driver

This API makes use of predefined data structures from *sys/mtio.h* listed in Table 5 to represent commands for the tape drive. In order to require the minimum number of parameters from the user, the *st* driver auto-populates some fields with tape specific repetitive information.

Table 5

Description of *st* specific data structures

mtop	mtget	mtpos
/* Used for executing a custom operation */ struct mtop { /* operations defined below */ short mt_op; /* how many of them */ int mt_count; };	/* Used for querying status info */ struct mtget { long mt_type; long mt_resid; /* the following registers are device dependent */ long mt_dsreg; long mt_gstat; long mt_erreg; /* The next two fields are not always used */ daddr_t mt_fileno; daddr_t mt_blkno; };	/* Used for READ POSITION command*/ struct mtpos { /* current block number */ long mt_blkno; };

The same functionalities can be obtained by performing an *ioctl* to the *sg* driver which exposes a single generic signature for *ioctl* with a generic data structure *sg_io_hdr_t*, defined *scsi/sg.h*, which gathers both the SCSI command and the reference to the buffer containing the actual data sent to/from the SCSI device. The function's signature is the following:

```
int ioctl (int fd, SG_IO, (struct sg_io_hdr_t*) sgh)
```

The structure `sg_io_hdr_t` is less intuitive and not tape specific. Few key fields have been highlighted in Table 6 for a better understanding of its usage. Given that it is a generic structure, it requires creating in user space the specialized data structures corresponding to the SCSI commands and passing them along in the appropriate fields. Although the `sg` approach is more demanding, owning the control over the command and data formats offers more flexibility when adding support for tape media which has hardware particularities according to each vendor.

Table 6

Description of `sg_io_hdr` fields [20]

Field	Description
int interface_id	Mandatory has value 'S'
int dxfer_direction	Possible values: - SG_DXFER_NONE - SG_DXFER_TO_DEV - "WRITE" - SG_DXFER_FROM_DEV - "READ" - SG_DXFER_TO_FROM_DEV - SG_DXFER_UNKNOWN
unsigned char cmd_len	Length of cmdp
unsigned char mx_sb_len	Max length of sbp pointer
unsigned short iovect_count	The number of scatter gather elements from the array pointed to by dxferp; if equals 0, the array is not being used
unsigned int dxfer_len	The number of bytes moved during the transfer
void * dxferp	If iovect_count = 0 then points to a user memory of dxfer_len bytes. Otherwise, points to a scatter-gather array of size iovect_count placed in the user memory
unsigned char * cmdp	The command to be executed
unsigned char * sbp	Points to a user memory buffer where the SCSI sense information will be outputted. On success, no sense information should be returned
unsigned char sb_len_wr	Size of the sbp buffer

In CERN tape archive software we distinguish two types of functionalities: ***stateless***, where the command relying on the `st` driver can be easily be replaced with a similar `ioctl` call to the `sg` driver and ***stateful***, where the execution or the result depends on the internal information kept by the `st` driver. In continuation, each function will be discussed along with our replacement proposal.

4.1 Stateless commands

4.1.1 Space commands

The space command refers to repositioning the reading / writing head to an absolute location or to a distance relative to its current position. Its layout is described in Fig. 13 and the key parameters are *code* and *count*.

In SCSI terms, logical objects are accessible entities recorded throughout the tape and they split into two categories: logical blocks and filemarks. The logical block represents the basic units of transfer between the client application and the tape medium. There is no 1-to-1 mapping between the logical blocks and the physical blocks on the medium, this being the responsibility of the device server. Unlike logical blocks, filemarks do not contain user data and they are used to separate groups of client data from each other. The detection of the filemarks is also the responsibility of the device server. Each logical object has a unique identifier in the partition.

The end-of-data (EOD) is a format-specific indicator that no valid logical objects are stored after this position and the last accessible position of the partition. The representation of EOD depends on the vendor but often consists of a group of filemarks.

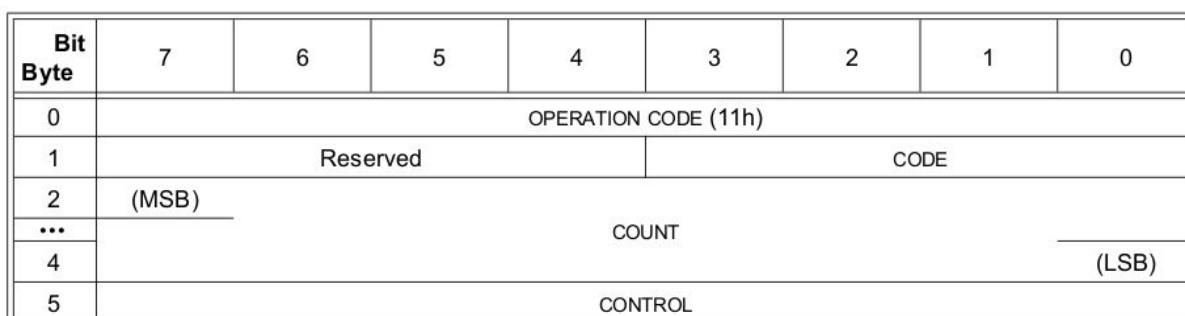


Fig. 13. SCSI SPACE command[18]

The *code* field indicates the type of spacing as shown in Table 7. The magnitude of the *count* field indicates the number of logical objects that will be spaced over in case of a relative repositioning. The spacing is possible in both directions: towards the beginning or end of the partition. In order to trigger a backwards spacing, the *count* field will contain a negative number in two's complement.

Table 7

The CODE field	
Value	Description
0x00	Blocks
0x01	Filemarks
0x02	Sequential filemarks

For a more flexible spacing, the SCSI interface also supports a similar command but having 8 bytes for the *count* field instead of 4.

4.1.2 Rewind command

The rewind operation means positioning the logical unit at the beginning-of-partition within the current partition. Its SCSI format is presented in Fig. 14. The only configurable option is *immed* which indicates when the device status will be returned: after the command completion or at the receipt of the command.

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (01h)							
1	Reserved							IMMED
2	Reserved							
...								
4								
5	CONTROL							

Fig. 14 SCSI REWIND command[18]

4.1.3 Load / Unload commands

This SCSI command (Fig. 15) is highly customizable and can request either the mounting or unmounting of the volume or a retention function.

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (1Bh)							
1	Reserved							IMMED
2	Reserved							
3	Reserved							
4	Reserved				HOLD	EOT	RETEN	LOAD
5	CONTROL							

Fig. 15 SCSI LOAD/UNLOAD command[18]

Besides the omnipresent *immed* field, several bits indicate the type of the command and special settings as follows:

- **HOLD** - If set when mounting, volume will be mounted but not positioned for access
- **EOT** - If set when unmounting, device will be positioned at end-of-medium before unmounting
- **RETEN** - Perform a retention function
- **LOAD** - Specify if the command is a LOAD or an UNLOAD

4.1.4 Write filemarks command

As the name implies, this SCSI command serves for writing a specified number of filemarks on the tape either for separating the user data, special fields (see Fig. 7) or for marking the end-of-data. The number of filemarks to be written is represented on 3 bytes and the location is given by the current position in the current partition. There is a similar command including special bytes for indicating the partition and the position in the partition where the write should start. The *immed* field has the same meaning as in previous command.

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (10h)							
1	Reserved						Obsolete	IMMED
2	(MSB)							
...	FILEMARK COUNT							
4	(LSB)							
5	CONTROL							

Fig. 16 SCSI WRITE FILEMARKS command[18]

4.2 Stateful commands

Reimplementing the stateful commands in user space requires understanding the conditions in which the meta-information kept by the *st* driver changes and to port the entire mechanism in the drive model.

4.2.1 Determining position

Two functionalities provided by the drive model are checking if the cursor is placed at the beginning-of-partition(BOP) or end-of-partition(EOP). The corresponding *ioctl* calls to the *st* driver return internal status bits which describe if the cursor is currently placed in one of these key positions.

For BOP checking, the meta-information is extracted and cached from the result of a SCSI command READ POSITION. Given that in CTA this check is performed rarely, replacing the existing *st-ioctl* which returns the cached information with an *sg-ioctl* which performs a SCSI query to the drive will not impact the execution speed. The response of the READ POSITION, presented in Fig. X, contains a series of useful drive status information such as:

BOP bit - indicates if the logical unit is placed at the BOP at the current partition

FIRST LOGICAL OBJECT LOCATION field - this field specifies the logical object identifier associated with the current logical position. For an alternative check, if this field is set to 0, then the current position is BOP

Bit Byte	7	6	5	4	3	2	1	0
0	BOP	EOP	LOCU	BYCU	Rsvd	LOLU	Obsolete	BPEW
1	PARTITION NUMBER							
2	Reserved							
3	Reserved							
4	(MSB) FIRST LOGICAL OBJECT LOCATION (LSB)							
...								
7								
8	(MSB) LAST LOGICAL OBJECT LOCATION (LSB)							
...								
11								
12	Reserved							
13	(MSB) NUMBER OF LOGICAL OBJECTS IN OBJECT BUFFER (LSB)							
...								
15								
16	(MSB) NUMBER OF BYTES IN OBJECT BUFFER (LSB)							
...								
19								

Fig. 17 READ POSITION command[18]

Similarly, for EOP the *st* driver updates its internal meta-information at each command which performs tape access or repositioning. When reaching or approaching the EOP, the *sense* buffer (Table 6) is used to notify the user application as a side information from the actual command effect. Conceptually, there are more possible positions relative to the EOP. First of all, there is a physical end-of-partition (EOP) and a logical-end-of-partition (LEOP). The EOP is determined by the proximity to the end of the permissible recording region on the medium. The LEOP is determined by the device server's restrictions such as the maximum logical object identifier. Sometimes EOP and LEOP can coincide (Fig. 18). Additionally, the early-warning (EW) position is an early indication that the logical unit is approaching the end of the permissible recording area. EW is placed far enough from the EOP to permit writing any buffered objects to the medium.

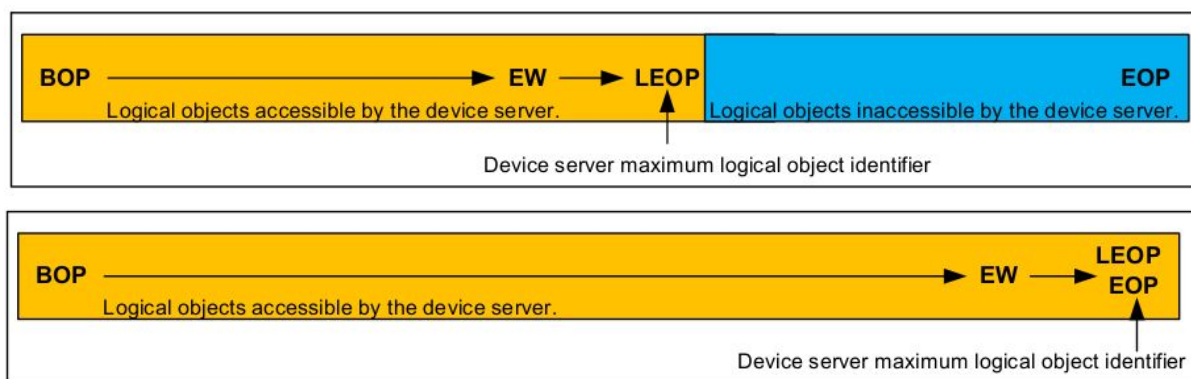


Fig. 18 Examples of placement of EW, LEOP and EOP [18]

In order to avoid the statefulness of this command, a suitable replacement it is performing a READ POSITION command and checking the **EOP bit** which, if set, indicates if the logical unit is placed between EW and EOP.

4.2.2 Test unit ready

In order for a user application to validly perform access or repositioning operations the tape must be mounted and the logical unit must be ready. Several reasons might prevent it from being available:

- logical unit not supported
- logical unit does not respond to selection
- medium not present (tape not mounted)
- logical unit is in process of becoming ready
- initializing command required
- manual intervention required
- format in progress
- unreported cause

The above listed causes are reported within the *sense* buffer when performing any SCSI operation. Consequently, the *st* driver caches the error reporting and, when a query is performed, only retrieves the stored value. In CTA, whenever a session is created the tape is mounted, process which lasts for 1-3 min depending on the hardware. This command is used to poll the drive if the tape is mounted and available to be accessed. However, due to caching, the *st* driver is not able to detect properly the change of state and the file descriptor of the tape must be reopened in order to enforce the update.

The proposed alternative is to perform a SCSI command TEST UNIT READY via the *sg* driver as method of checking if the logical unit is ready. This is a simple SCSI command with no input parameters performed to populate the *sense* buffer with the status of the logical unit.

4.3 Special commands

All operations discussed previously are performed via the *ioctl*/system call. However, some commands are sent via other system calls among which relevant to mention are *read* and *write*. Both *st* and *sg* drivers implement a kernel read and write but, since the *st* driver is the one matched by the SCSI subsystem with a physical tape, the system calls will be directed to it. The *st* implementations consider the block format of the data stored on tape medium, checks the errors reported through the *sense* buffer and updates the stateful meta-information. The SCSI commands performed behind are READ BLOCK and WRITE. Although both have a specific operation code and format that can be send via *ioctl* an system call, the actual data transfer is performed via DMA which would be difficult to reimplement in userspace. The alternative would be enforce the SCSI subsystem to match the special system calls with the *sg* kernel implementations. This task is still under analysis.

5. Results

5.1 Speedup of adding RAO support

The purpose of implementing Recommended Access Order in CERN tape archive software is to take advantage of the proximity of the files on tape in order to minimize the seek time between them at recall. To test the newly added functionality, a tape server instance has been installed on a Linux machine and two physical tapes have been used as samples. The tapes are written at their full capacity with random sized files in CASTOR format (Fig. 7) - format kept in CTA. They are different in size, number of files and maxUDS (maximum number of files that can be sorted with the Recommended Access Order).

For each test, a specific number of file indexes is generated and then requested from the tape server with the RAO feature both enabled and disabled. If RAO is enabled, the tape server performs a RAO query to the drive and then recalls the files in the given order. If RAO is disabled, the tape server recalls the files sequentially, in the same order they were requested (the files are sorted increasingly by index to simulate the stager's behaviour). The results are presented in Tables 7 and 8.

Tape1 - 30061 stored files

Table 7

Results for Test1 - Tape1			
FilesNo	SEQ	RAO	Speedup
50	30m2.076s	7m39.418s	3.922
100	48m53.257s	12m15.970s	3.985
150	62m39.904s	15m25.419s	4.062
200	69m55.510s	17m11.807s	4.066
250	79m57.389s	20m12.145s	3.957
500	98m26.789s	32m31.074s	3.027
750	108m5.068s	46m37.452s	2.318
1000	116m8.242s	55m38.017s	2.087
2000	140m7.878s	90m24.740s	1.549
2730 (maxUDS)	150m56.380s	109m24.892s	1.379
3500	159m39.288s	132m26.958s	1.205

Tape2 - 54803 stored files

Table 8

Results for Test1 - Tape2			
FilesNo	SEQ	RAO	Speedup
50	30m25.712s	6m12.958s	4.895
100	52m32.936s	10m16.068s	5.117
150	81m16.307s	14m52.061s	5.466
200	96m42.822s	17m19.344s	5.583
250	122m17.238s	20m5.954s	6.084
500	169m14.619s	43m13.255s	3.915
750	212m0.068s	64m30.018s	3.286
1000	226m45.786s	85m15.312s	2.659
2000	279m34.291s	165m39.512s	1.687
3000 (maxUDS)	317m9.936s	316m58.124s	1.000
3500	329m58.503s	330m51.513s	0.997

The recall time for a series of files is composed both of the seek time between the files and the time for the actual reading. The latter one is not affected by the Recommended Access Order but the times presented in Tables 7 and 8 emphasize how it impact the seek time. It can be also noticed that the speedup obtained by applying RAO is not constant. For a sufficiently big number of files (exact number depending on tape) the seek time is a small portion of the total time. Therefore, the optimization is unnoticeable. For a repack (recall of all files from the tape), RAO is irrelevant because the optimal order is identical with the sequential order. As it can be observed in Table 8 for a number of 3500 files, the RAO time is higher than the SEQ time because of the additional querying and reordering. Fig. 19 and Fig. 20 present a visual representation of the results.

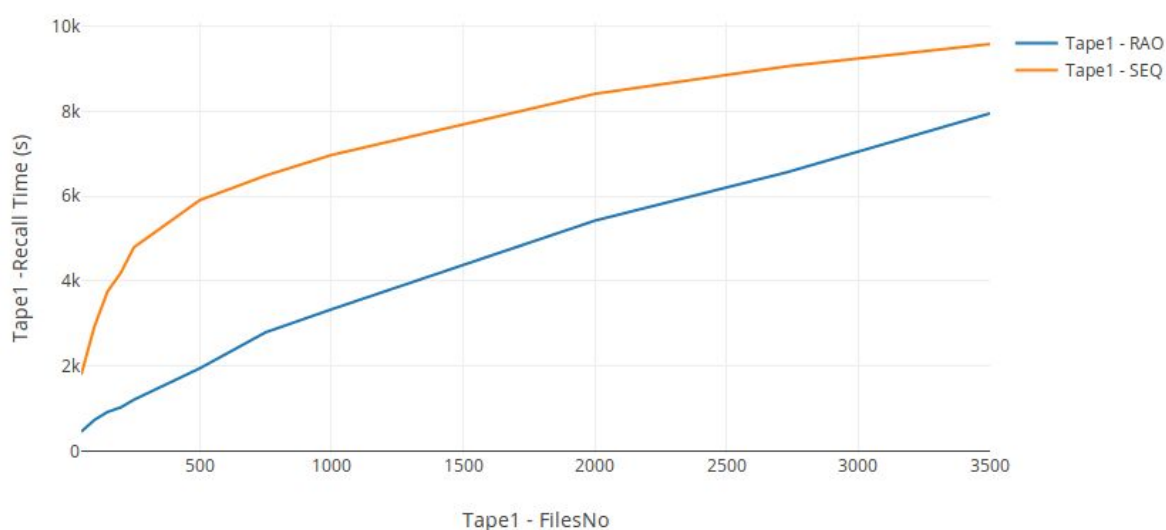


Fig. 19 Tape1 - Recall times for an increasing number of files

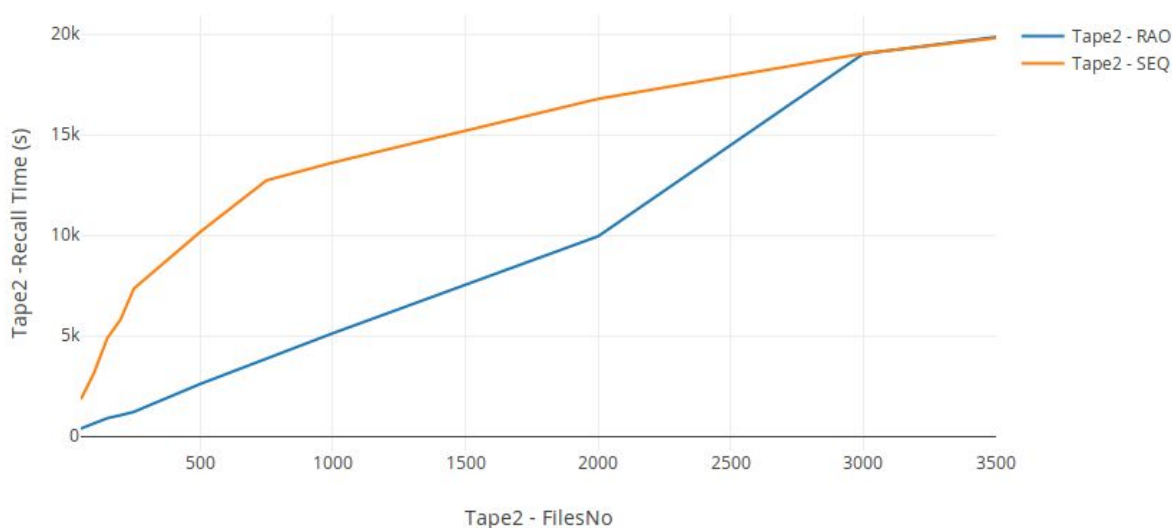


Fig. 20 Tape2 - Recall times for an increasing number of files

We deduct that the best speedups are obtained for a number of files equal with the number of wraps. If the files to be recalled are placed one per each wrap, a standard sequential access would require proceeding each track from the beginning to the end in order to obtain the requested files. This implies that the greatest portion of the total recall time is the seek time which can be easily optimized with RAO. The test data (the file IDs) used for the tests are randomly generated so we can assume that they are uniformly scattered across the tape wraps.

In order to validate our theory, the second test varies the number of files between numbers close to the number of wraps. Depending on tape, the number of wraps varies between 150 and 200. Tables 9 and 10 contain the results of the second test. As it can be seen, the speedup for applying RAO peaks around 140 requested files for Tape 1 and 160 requested files for Tape2.

Table 9

Results for Test2 - Tape1			
FilesNo	SEQ	RAO	Speedup
140	57m36.144s	11m54.456s	4.837
150	62m39.904s	15m25.419s	4.062
160	64m33.852s	14m39.362s	4.405
170	70m52.517s	16m11.810s	4.375
180	67m9.022s	16m42.734s	4.018
190	71m55.529s	16m58.557s	4.236
200	69m55.510s	17m11.807s	4.066

Table 10

Results for Test2 - Tape2			
FilesNo	SEQ	RAO	Speedup
140	67m31.076s	11m21.201s	5.946
150	81m16.307s	14m52.061s	5.466
160	84m53.203s	13m47.413s	6.155
170	80m20.236s	13m28.781s	5.959
180	87m35.630s	15m6.406s	5.798
190	95m20.767s	15m42.779s	6.067
200	96m42.822s	17m19.344s	5.583

A comparative analysis over the speedup is shown in Fig. 21. Since Tape2 has a bigger storing capacity the relative distance between files is higher, thus the seek time is higher. For this reason, the RAO algorithm provides a better speedup.

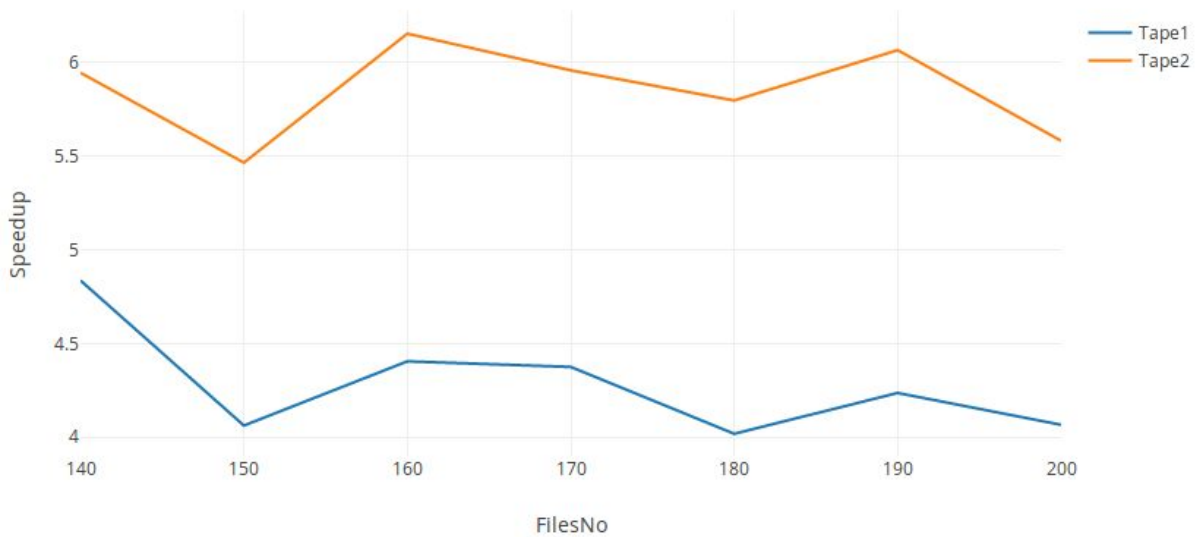


Fig. 21 Recall times for a number of files close to the number of wraps

5.2 Status of replacing *st* with *sg*

Currently the stateless functionalities have been completely ported to an *sg* based interface and the stateful functionalities are under development. The *st* meta-information is being ported into userspace by recreating the state automaton from the kernel. However, in order to perfectly determine the status, all tape commands which change the state of the drive (add or remove data, reposition the logical unit, etc.) must be tracked and the error buffer must be analysed. The challenge of this milestone is to provide a solution for porting the read / write functionalities into the *sg* interface while keeping the data access accuracy of the tape specific kernel module. The advantages of this optimization can be obtained only after the complete removal of the *st* based interface.

6. Conclusions and Further Work

Within this project, the CERN tape archive software (CASTOR and CTA) has been extended with support for the Recommended Access Order for files' recall. The optimization has been tested and validated, offering speedups up to 6x for generic test data and will be included and monitored in production. Experimentally, we have concluded that the best speedups are obtained for a number of recalls equal with the number of wraps, which is usually around 150. Conveniently, the number of files requested simultaneously in CERN archive software is of few hundreds, which makes RAO a suitable optimization. However, in some situations, such as repacks or simply recalls of a large number of files, the RAO algorithm can even introduce a slowdown. Consequently, the usage of RAO was created as a configurable setting. The RAO implementation introduced an additional reordering step while keeping intact the pipelined architecture of the project and also took into consideration the recovery mechanism in case of failure. Moreover, the extension of the low-level SCSI implementation to support RAO queries has been done for the generic drive abstraction so the algorithm will be applicable for various tapes from different vendors if the drive has hardware support for RAO.

The second optimization, of replacing the usage of the *st* (SCSI tapes) kernel module with an *sg* (SCSI Generic) based interface is currently under development. The aim is to eliminate the caching and the unnecessary permission checking mechanism (Linux capabilities) performed by the *st* driver which complicate both the code and the execution flow. The stateless *ioctl* based functionalities have been completely ported to the *sg* driver and the next challenge is to port the stateful meta-information normally kept by *st* to userspace in order to obtain the stateful functionalities. However, for a complete state automaton to be implemented, the other system calls such as read, write must be correctly ported to *sg* and their result must be tracked. Further development requires finding a suitable and efficient solution for this challenge.

BIBLIOGRAPHY

- [1] Monitoring Evolution at CERN, P. Andrade, B. Fiorini, S. Murphy, L. Pigueiras, M. Santos, CERN, 2015
- [2] CTA: CERN Tape Archive Overview and architecture, V. Bahyl, G. Cancio, E. Cano, M. Davis, C. Moraru, S. Murray, CERN, 2017
- [3] “What is LTO Technology?”, Ultrium LTO [\[link\]](#)
- [4] Tape drive attributes, IBM [\[link\]](#)
- [5] STORAGETEK SL3000 Modular Library System Datasheet
- [6] IBM 3592 tape cartridge, High-capacity tape media engineered for the enterprise, IBM DataSheet
- [7] IBM tape drives [\[link\]](#)
- [8] Tape driver (st), The Linux SCSI Generic [\[link\]](#)
- [9] stinit, Linux man page
- [10] Generic driver (sg), The Linux SCSI Generic [\[link\]](#)
- [11] Experiences and challenges running CERN's high capacity tape archive, G. Cancio, V. Bahyl, D. Kruse, E. Cano, S. Murray, CERN, 2015
- [12] Data Storage at CERN (presentation), J. Iven, CERN, 2014
- [13] Disk storage at CERN, L Mascetti, E Cano, B Chan, Espinal, A Fiorot, H Gonzalez Labrador, J Iven, M Lamanna, G Lo Presti, JT Mościcki, AJ Peters, S Ponce, H Rousseau and D van der Ster, CERN, 2015
- [14] Increasing the efficiency of tape-based storage backends, N. Bessone, G. Cancio, S. Murray, G. Taurelli, CERN, 2009
- [15] The new CERN tape software - getting ready for total performance, E. Cano, S. Murray, D. Kruse, V. Kotlyar, and D. Come, CERN, 2015
- [17] Streamlining CASTOR to manage the LHC data torrent, G. Presti, X. Curull, E. Cano, B. Fiorini, A. Ieri, S. Murray, S. Ponce, E. Sindrilaru, CERN, 2013
- [16] CTA Documentation
- [18] Information technology - SCSI Stream Commands, T10 Working Draft
- [19] Linux capabilities, Linux man page [\[link\]](#)
- [20] The sg_io_hdr_t structure in detail, The Linux SCSI Generic [\[link\]](#)