Contents

Preface 3

1 Developer’s manual 4
  1.1 Requirements ....................................................... 4
     1.1.1 Targeted environment ........................................ 4
     1.1.2 Pre-existing requirements .................................... 4
     1.1.3 Extra requirements ........................................... 6
  1.2 Tape server architecture ........................................ 6
  1.3 Reference documentations ......................................... 7
     1.3.1 SCSI specifications ............................................ 7
     1.3.2 SCSI support in Linux ......................................... 8
     1.3.3 Unsorted CAStor docs ........................................ 9
     1.3.4 SCSI tape support in Linux (st driver) ...................... 9
  1.4 Tools used during development ..................................... 9
     1.4.1 Required tools for build ..................................... 9
     1.4.2 Tools used during development ............................... 9
     1.4.3 Code coverage using lcov ................................... 9
  1.5 Data transfer session step by step ............................... 10
  1.6 Software layout .................................................. 11
     1.6.1 SCSI structures, constants and endianness ................. 11
     1.6.2 Exceptions hierarchy and error handling strategy .......... 12
     1.6.3 Non-fatal warnings strategy ................................ 13
     1.6.4 The Tape::Drive object ..................................... 13
     1.6.5 The Tape::File class ........................................ 17
     1.6.6 FIFOs .......................................................... 21
     1.6.7 Disk client library .......................................... 21
     1.6.8 VDQM client library ......................................... 21
     1.6.9 VMGR client library ......................................... 21
     1.6.10 Stager/TapeGateway client library ......................... 21
     1.6.11 Logging system client library .............................. 21
1.6.12  Application architecture .................................................. 22
1.6.13  How to interpret the exit code of the mount-session child process .... 25
1.7    Compilation instructions ...................................................... 25

2  Administrator's manual .......................................................... 28
2.1    User and capabilities ......................................................... 28
2.2    Pending questions ............................................................. 28
Preface

The Tape server project is targeted at replacing the CAStor tape server with a new drop-in reimplementation. The reimplementation will replace a legacy implementation that is written in C.

The reimplementation will be done using the latest tools available to us in the current Scientific Linux distribution. The language will be C++, to group concept and variables in self-contained (and unit testable) objects.

The interface to the mounting daemons might still change with respect to CAStor 2.1.14 as the mounting daemons are being reviewed in parallel.

This documentation itself currently references the older tape server this project is intending to replace. The references will have to be removed as they become unnecessary. Likewise, the layout of the document will be adapted.

The tape drive primitives have now been developed, and the rest of the project’s plan is being laid out.
Chapter 1

Developer’s manual

1.1 Requirements

1.1.1 Targeted environment

CERN SLC5 and SLC6, 64bits. Although it should compile in theory, the 32 bits version is not tested. The unit test purposely returns an error when run on non-64 bits architecture.

1.1.2 Pre-existing requirements

The new tape server (software) will have to replace the software running on a tape server (computer). A previous analysis describes the current software stack of the tape servers. This new tape server will retain the same external interfaces as the old tape server, replacing the stack of daemons from the tape bridge down to the tape drive hardware.

The tape server will interface with the Volume and Drive Queue Manager daemon (vdqmd), the Volume manager daemon (vngrd), the Castor User Privilege Validation daemon (cupvd) and tape gateway daemon (tapegatewayd) for data transfer management and access control.

It will connect to the CAStor disk servers to transfer the data itself, using one of the supported protocols (current candidates are rfio, xroot and ceph) and it will use the services of the Remote Media Changer daemon (rmcd) to mount and unmount tapes.

1.1.2.1 Tape session triggering

The tape server acts as a server only on one occasion, when it receives a client info request from the VDQM. This call triggers a tape session, recall or migrate. The information received at that point is only client system information and drive information (request ID, host, port, DGN, tape drive name and user information (user id, etc...)).

The main thread of the tape server will then just forks (and maybe exec, to be decided (TODO)) a new process, which will handle the session and quit.

A new session starts with the tape server connecting with the client (it could be the tape gateway, or one of the command line commands of castor (currently packages in castor-tapebridge-client: readtp, writetp, dumtp)). In the case of the tape gateway, it can be either a read or a write session.

The collaboration diagrams of the previous version of the tape server (with all its sub components) can be found in dot format.

---

2 http://svn.cern.ch/guest/CASTOR/CASTOR2/trunk/castor/tape/doc/collaboration_diagrams/
The new sequencing of a session start, simplified from the internal component communication is shown here:

At that point, already two client libraries are in use in the tape server: the tape gateway protocol client library and the vdqm client library, and a simple server, answering only one type of requests:

- The client library for the tape gateway is implemented using the UML/umbrello based serialisation system. It is contained in the `castor::tape::tapebridge::ClientProxy` class in CASStor.
- The direct C client API in CAStor’s `h/vdqm_api.h`.
- The VDQM request handler is implemented in the server class `castor::tape::tapebridge::VdqmRequestHandler`.

1.1.2.2 Tape session startup

Upon reception of the session request, the tape server checks that it can continue with the session:

- with VMGR the volume status and block size. A disabled volume can only be mounted by a TP_OPER (this information is retrieved from cupv).
- In case of a write session, it queries the pool info know the owner. If user is not the owner or ADMIN, refuse the mount.
Then reports are sent:

- Reports to VDQM that the drive is assigned. (VDQM_UNITSTATUS).
- Requests work to be done from the client to prevent useless mounts (and start caching data in case of migration).
- Mounts the tape, rewinds and validates the volume label.
- Reports to VMGR that the volume is mounted (VMGR_TPMOUNTED).
- Reports to the VDQM that the tape is mounted (VDQM_UNITSTATUS).

This part adds the `vmgr client` and the `rmcd client`.

TODO: bad day scenario.

### 1.1.2.3 Work loop

The work loop will be a 4 step pipelined operation where:

- The work to be done FIFO gets topped up continuously by a thread (using bulk)
- The data FIFO(s) is(are) filled up by the reader (tape thread or disk thread).
- The data FIFO consumer (opposite)
- The results are reported by either the control thread or a specialized thread (using bulk).

### 1.1.2.4 Release tape

### 1.1.3 Extra requirements

Additional requirements, arising from the current practices of operators are:

- The tape server’s session should gracefully handle an unclean situation where the tape is left in the drive by a previously crashed session. The protocol is to clean anything left over before proceeding to the new session.
- A tape session should be preemptable by the operator. This is currently achieved by killing the tape process. Closing the session on a kill (-1) could be a solution.
- The operator should be able to specify values in different SCSI code pages in order to setup the tape drive. This setting will be defined differently for each tape drive type.

### 1.2 Tape server architecture

To fulfill the requirement for an ability to kill a session, the main tape server daemon will be simple, and just report its status to the VDQM and wait for requests from it on an open port.

When a tape mount should start, the process will fork a child process, which will reserve the memory and instantiate the tape mount machinery.

The layout of the main process is shown in figure 1.1. The layout of the child process, which contains all the complexity is shown in figure 1.2.

The data path will go to/from tape drive, through the generic SCSI interface of st driver (CAStor uses a mixture of both in the Tape::Drive class), then through the File structure support classes, as controlled by
the tape thread. The tape thread will communicate the data to (or get from) the disk threads via the data FIFO class. This class will in turn allocate the memory from a preallocated, pool of fixed sized blocks. The size of the pool will be controlled by the operators.

The file structure support classes will be arranged in two layers, with two distinct implementations for read and write. The lower layer, the session will be stateful, and validate that the conditions are correct before operations (correct tape is identified in the drive thanks to volume label, and proper position is confirmed thanks to last file trailer before writing). Positioning can also be optimized if file to file information is retained (mostly, we should avoid repositioning between consecutive files when reading).

Some libraries already exist in CAStor, and will be reused, either by copying or linking from pre-compiled packages. The main parts of the sessions spawner will be taken from the VDQM as well.

1.3 Reference documentations

1.3.1 SCSI specifications

The SCSI commands can be found in the SCSI section of Hackipedia.org. The most significant documentations for tape server development are the SCSI stream commands (SSC-3) and the SCSI primary commands (SPC-4).

1.3.1.1 Manufacturer's specificities

The SCSI specification allows for some flexibility for the manufacturers of tape drives, and each of them has differences. The details can be found in the following documentations:

- StorageTek T10000 Tape Drive
- Sun StorageTek T10000 Tape Drive Fibre Channel Interface Reference Manual
- IBM System Storage TS1120 and TS1130 Tape Drives and TS1120 Controller Operator Guide

http://hackipedia.org/Hardware/SCSI/3
http://hackipedia.org/Hardware/SCSI/Primary%20Commands/SCSI%20Primary%20Commands%20-%204.pdf
http://docs.oracle.com/cd/E19957-01/96174E/96174E.pdf
http://docs.oracle.com/cd/E19772-01/MT9259L/MT9259L.pdf
1.3.2 SCSI support in Linux

On the Linux side, the main references are the Linux 2.4 SCSI subsystem HOWTO, especially for its section 9.3 on the st driver, and the Linux SCSI Generic (sg) HOWTO.

More details regarding the Generic SCSI driver can be found on the SCSI subsystem maintainer’s web site.

The section on the SG_IO ioctl details the usage of the simplest ioctl for the generic SCSI driver, which allows the invocation of a SCSI command and the collection of the result in a single system call.

This ioctl is provided in the middle layer of the SCSI subsystem of Linux. All SCSI drivers, st included, fall back to the middle layer when encountering an unknown ioctl. This means there is no need to open the matching generic SCSI, unless we want to control command queueing with separate sending of commands and result collection, which requires the use of read and write calls from the generic SCSI (sg) driver.

---

11 http://www-01.ibm.com/support/docview.wss?uid=ssg1S7003248&aid=1
12 http://www-01.ibm.com/support/docview.wss?uid=ssg1S7003556&aid=1
13 http://mirrors.kernel.org/LDP/HOWTO/pdf/SCSI-2.4-HOWTO.pdf
15 http://sg.danny.cz/sg/
1.3.3 Unsorted CAStor docs

A collection of links to various documentations written in the past is available on one of CAStor’s web pages.

1.3.4 SCSI tape support in Linux (st driver)

Generic SCSI allows detailed control of the operations, but the bulk of them (including reading and writing) can be managed by the higher level SCSI tape (or st) driver provided by the Linux kernel. More information on the st driver can be found in the man page "st" and in Documentation/scsi/st.txt in the sources of the kernel.

1.4 Tools used during development

1.4.1 Required tools for build

- GCC/G++ (Basic SLC version)
- CMake (Basic SLC version)
- rpmbuild (Basic SLC version)
- Google Mock/Google test (GTest is provided in EPEL repository for SLC. GMock requires recompilation. The source RPMs can be found for newer versions of RPM based distributions, for example from rpmfind. For convenience, they are also available on AFS as a temporary solution.
- Valgrind (Basic SLC version)
- LATEX (Basic SLC version) to compile this document
- Doxygen for code documentation (Basic SLC version)

1.4.2 Tools used during development

- mhvtl for developing against virtual drives and libraries (to enable mhvtl kernel debug output to dmesg opts=3 have to be used for kernel module options, i.e. modprobe mhvtl opts=3).
- TeamCity for continuous integration
- NetBeans as an IDE, including for remote development

1.4.3 Code coverage using lcov

Although the code coverage is not integrated in the build process, it is straightforward to run on the code. The following recipe will deliver a set of web pages indicating which parts of the code are covered or not in the unit tests. The lcov package is required. It is only available on SLC6, and can be installed via yum.

- Change the main CMakeFiles.txt as in this diff:

---

17 http://castorwww.web.cern.ch/castorwww/links.htm
18 http://rpmfind.net/linux/rpm2html/search.php?query=gmock
19 /afs/cern.ch/user/c/canoc3/public/GoogleTest-Mock
20 https://sites.google.com/site/linuxvtl2/
1.5 Data transfer session step by step

The following sequence describes when and what the different components of castor communicate while a data transfer is occurring

1. Tape operator runs "readtp V12345" -n 1 or "writetp V12345 /etc/group"
2. readtp/writetp queries vmgrd for information about the tape with volume identifier V12345
3. readtp/writetp queries cupvd to determine whether or not the tape operator really has the TP_OPER privilege
4. readtp/writetp aborts if the tape is DISABLED and the tape operator does not have the TP_OPER privilege
5. readtp/writetp aborts if the tape is either EXPORTED or ARCHIVED
6. readtp/writetp binds a TCP/IP listening socket ready for callbacks from a mount-session child-process of tapeserverd
7. readtp/writetp requests vdqmd to allocate it a free drive in the same DGN as tape V12345 - readtp sends its callback port in the request
8. The RequestHandlerThread of vdqmd stores the readtp/writetp request in the database
9. The DriveSchedulerThread of vdqmd allocates a free drive to the readtp/writetp request
10. The RtcpJobSubmitterThread of vdqmd sends a job to tapeserverd
11. tapeserverd checks that the vdqmns hostname is in the ADMIN HOSTS parameter of /etc/castor/castor.conf
12. tapeserverd forks a mount-session child-process.
13. The TapeDaemon::forkMountSession() function of the child process asks the vdqmd daemon to assign the process ID to the drive.
   - VdqmProxy::assignDrive()
14. The child process creates the MountSession object and calls MountSession::execute().
15. mount-session asks readtp/writetp for the details of the mount (tapegateway::Volume)
16. mount-session sends the mount details (tapegateway::Volume) to the tapeserverd parent process
   - TapeserverProxy::gotReadMountDetailsFromClient()
   - TapeserverProxy::gotWriteMountDetailsFromClient()
17. If writing to tape and the client is the tapegatewayd daemon then the tapeserverd parent process checks with the vmgr daemon that the tape is marked as BUSY. A tape is marked as busy as a result of the tapegatewayd daemon calling vmgr_gettape().
18. mount-session asks readtp/writept for the first file to be recalled or migrated
19. mount-session requests rmcd to mount the tape - the request is synchronous
   - RmcProxy::mountTape()
20. mount-session checks with the drive that the tape has been mounted
21. mount-session notifies the tapeserver parent process that the tape has been mounted
   - TapeserverProxy::tapeMountedForRead()
   - TapeserverProxy::tapeMountedForWrite()
22. tapeserverd via MountSessionAcceptHandler::handleIncomingUpdateDriveJob() notifies the vmgr daemon that the tape has been mounted
   - VmgrProxy::tapeMountedForRead()
   - VmgrProxy::tapeMountedForWrite()
23. tapeserverd via MountSessionAcceptHandler::handleIncomingUpdateDriveJob() notifies the vdqmd daemon that the tape has been mounted
   - VdqmdProxy::tapeMounted()
24. mount-session recalls/migrates and requests more files until gateway::NoMore
25. mount-session unloads tape from drive
26. mount-session requests rmcd to unmount the tape - the request is synchronous
   - RmcProxy::unmountTape()
27. mount-session notifies tapeserverd the tape has been unmounted - currently this is not used
   - TapeserverProxy::tapeUnmounted()
28. mount-session terminates with the success value of 0
29. tapeserverd reaps the zombie mount-session
30. tapeserverd parent-process marks the drive as FREE
31. tapeserverd via TapeDaemon::postProcessReapedDataTransferSession() notifies vdqmd that it is releasing the tape
   - VdqmdProxy::releaseDrive(forceUnmount)
32. tapeserverd via TapeDaemon::postProcessReapedDataTransferSession() notifies vdqmd the tape has been unmounted
   - VdqmdProxy::tapeUnmounted()

1.6 Software layout

1.6.1 SCSI structures, constants and endianness

In order to make the code readable, and to avoid heavy mask-and-shift usage (which one would tend to code using literals in order to avoid many constants definitions), we use bit field structures. The unused fields can be left anonymous. The definition is shown in listing 1.1, and usage in listing 1.2. As there could be endianness issues, we limit this usage to within bytes. Fortunately, the SCSI standard nicely adheres to this rule.

The unit test resorts to shift and mask, once and only once, to validate the bit fields in another way. There is an example for this validation in SCSI/StructureTest.cc an excerpt is in listing 1.3.

Other common types in the SCSI specification are multi-bytes number, which are represented by unsigned char[2/* (or 4)*/] and handled by helper functions toU16() and toU32(). The helper functions conveniently use ntohl{1|s}, as SCSI and network orders are the same. The reverse is
namespace SCSI {
  namespace Structures {

    /* Inquiry data as described in SPC-4. */
    typedef struct {
      unsigned char perifDevType : 5;
      unsigned char perifQualifier : 3;
      unsigned char : 7;
      unsigned char RMB : 1;
      unsigned char version : 8;
      unsigned char respDataFmt : 4;
      unsigned char HiSup : 1;
      unsigned char normACA : 1;
      unsigned char : 2;
      ...
    } inquiryData_t;
  }
}

Listing 1.1: SCSI::Structures example

covered by setU16() and setU32(). Another helper function takes care of string extraction from fixed sized char arrays. See listing 1.4.

Those arrays are space-padded, and may not be 0 terminated. It is seen in listing 1.2. The helper function extracts the string, dealing with potential zeros at the end, and the fixed length. They keep the space-padding at the end of the extracted string.

To avoid literals in the code, which forces anyone reading it to do tedious lookups, the SCSI constants are also defined as constants in the code. See listing 1.5.

Finally all structures have a constructor, which at least zeroes all the data. Some structures (typically the CDBs, where the first byte is the operation’s code) automatically set the value of fields which can only have one value. Helper functions are created as needed, where accessing/setting the data in the structure requires non-trivial processing (and when the case is not covered by the common tools handling strings that endianness).

1.6.2 Exceptions hierarchy and error handling strategy

There is a small class hierarchy for exceptions: Tape::Exception inherits from std::exception, and Tape::Exceptions::Errnum inherits from the latter. Tape::Exceptions::Errnum manages the errnos. It collects the errno value and turns it into a string automatically at construction time.

Tape::Exception and all its heirs automatically generate a stack trace at creation time. This allows easy tracing of unhandled exceptions, as the stack trace is embedded in the content of the what() method. For the cases where the exception is indeed handled, a shorter version called shortWhat() allows the logging of the problem without bloating the logs with long stack traces.

Another exception class, SCSI::Exception, turns the SCSI status and sense buffer into a user readable string. In addition, a helper exception thrower function avoids code repetitions (ExceptionLauncher()).

Throughout the project, the error handling strategy is to throw an exception when any error condition occurs. This ensures that any returned value is valid, and prevents the calling function from testing for
error conditions. The default exception throwing is coming from a narrow set of exceptions types. This gives a crude exception handling capacity to the user of the functions. When finer grained exceptions will turn out to be required, we will add them on an as needed basis.

### 1.6.3 Non-fatal warnings strategy

We want to deliver an interface, preferably common, to most object where the non-fatal problems are recorded (with time of occurrence) and stored for further retrieval by upstream caller. This allow developers to deal with the logging interface only in the top "application" class which glues all the bricks of the project together.

A lower level failure (exception) could also be turned into a warning by a higher level retry.

TODO: define API.

### 1.6.4 The Tape::Drive object

This first deliverable is a tape drive object. This tape drive object abstracts all SCSI and technical details and provides a high level interface, to be used by the file structure layer.

It will provide as much data safety as possible by blocking writes in situations where they are not safe (to be defined in details, but the most obvious is right after positioning, as the file layer is expected to check the position by reading the trailer of the previous file before writing.

The SCSI commands and st driver’s functions used in previous software (CAStor’s taped/rtcpd) are:

- Individual SCSI commands sent using generic SCSI:
  - Read status (inquiry SCSI command used by posovl)
  - Read serial number (inquiry SCSI command, asking for vital product data page 0x80)
  - Locate (locate(10) SCSI command: 32 bits logical object identifiers) 21
  - Read position (read position SCSI command – short form): get the current logical object location (a.k.a. block ID).

21There is also a locate(16) command allowing 64 bis addresses. This might become necessary as tapes grow. Discounting the per-file overhead, with 250kB block, it still takes 1PB to get $2^{32}$ blocks.
namespace UnitTests {
TEST(SCSI_Structures, inquiryData_t_multi_byte_numbers_strings) {
    /* Validate the bit field behavior of the struct inquiryData_t, which represents the standard INQUIRY data format as defined in SPC-4. This test also validates the handling of multi-bytes numbers, as SCSI structures are big endian (and main development target is little endian). */
    unsigned char inqBuff [100];
    memset(inqBuff, 0, sizeof(inqBuff));
    SCSI::Structures::inquiryData_t & inq = *((SCSI::Structures::inquiryData_t *) inqBuff);
    /* Peripheral device type */
    ASSERT_EQ(0, inq.perifDevType);
    inqBuff[0] |= (0x1A & 0x1F) << 0;
    ASSERT_EQ(0x1A, inq.perifDevType);
    /* Peripheral qualifier */
    ASSERT_EQ(0, inq.perifQualifier);
    inqBuff[0] |= (0x5 & 0x7) << 5;
    ASSERT_EQ(0x5, inq.perifQualifier);
    [...]
}
}

Listing 1.3: SCSI::Structures usage example

SCSI::Structures::uint32_t toU32(const char(& t)[4]);
SCSI::Structures::uint32_t toU32(const char(& t)[4]);

template <size_t n>
std::string toString(const char(& t)[n]);

Listing 1.4: SCSI::Structures helper functions

namespace SCSI {
    class Commands {
    public:
        enum {
            /*
             * SCSI opcodes, taken from linux kernel sources
             * Linux kernel’s is more complete than system’s
             * includes.
             */
            TEST_UNIT_READY = 0x00,
            REZERO_UNIT = 0x01,
            REQUEST_SENSE = 0x03,
            [...]
        }
    }
}

Listing 1.5: SCSI::Constants
- Log select (for clearing compression stats page. The function clear_compression_stats actually does a blanket reset of all statistics. It sets the PCR/SP/PC combination to 1/0/3. The basic SCSI specification states that the value pf PC is not important, but for the T10000 drives, the documentation recommends PC=11b, which we have for all drives.

- Log sense, to read the compression pages. This is device dependent. The code covers 5 blocks of device types: DAT, DLT-SDLT-LTO, IBM(3490, 3590, 3592), StorageTek RedWood(SD3), StorageTek(9840, 9940, T10000).

- Log sense for page 0x2E (tape alert, as defined in SSC-3) on all modern tape drives to detect tape alerts.

- Mode sense and Mode select was used in setdens called itself by mounttape. They get the drive parameters and set density and compression parameters based on the drive type and the density requested by the caller. On all modern tape drives, the compression page is 0x10. This will be replaced by the function `Tape::Drive::setCompressionAndDensity()`.

- st driver's commands, leading to internal variables setting or SCSI actions:
  - Get internal driver state via the MTIOCGET ioctl (for drive ready, write protection, get some error condition, when MTIOSENSE failed, to get the EOD, BOT bits (readlbl)). This functionality is covered by `Drive::getDriveStatus`.
  - Try and get the sense data for the last-ish command with MTIOSENSE. This relies on a CERN-made patch. As the patch is not available in SLC6, MTIOSENSE will not be used in this project. This is also covered `Drive::getDriveStatus`.
  - Setup the driver’s parameters (MTIOCTOP/MTSETDRVBUFFER) for (un)buffered writes and asynchronous writes (in confdrive, a child process of taped). This option is currently not set in production environments.
  - Jump to end of media (before rewinding, as a mean to rebuild the MIR) (MTIOCTOP/MTEOM, with some MTIOCTOP/MTSETDRVBUFFER before, in repairbadmir). The setting of the driver buffer is used to set the boolean flag MT_ST_FAST_MTEOM to 0. If not, the mt driver uses a nasty trick asks the device to skip 0x7fffffff files forward. The comment in the CAStor code claims it’s 32k files, but $2^{23} - 1$ is indeed 8M files. Anyway, after turning off the option, the st driver reverts to telling the SCSI device to space to end of data. This behavior is documented in the IBM’s operator manual mentioned in 1.3.1.1, on page 53 for tape alert 18 (Tape directory corrupted on load).

It is not mentioned for other tape server’s documentations. Specifically, StorageTek only lists operator-initiated methods for MIR rebuild. Nevertheless, we will still issue this operation in all drives as it is not known if it works in practice for StorageTek drives (or others).

- Rewind (MTIOCTOP/MTREW, in rwdtape).
- Skip to end of data (MTIOCTOP/MTEOM, in skip2eod, without the trick of repairbadmir).
- Skip n file marks backwards (MTIOCTOP/MTBSF, in skiptpfb).
- Skip n file marks forward (MTIOCTOP/MTFSF, in skiptpff).
- Skip n file marks forward (MTIOCTOP/MTFSF, in skiptpfff). skiptpfff and skiptpff differ only by error reporting. Both functions exists since CAStor has been put in SVN (20/07/1999)
- Skip n blocks backwards (MTIOCTOP/MTBSR, in skiptprb).
- Skip n blocks forward (MTIOCTOP/MTFSR, in skiptprf).
- Unload the tape (MTIOCTOP/MTOFFL, in unldtape).
- Write synchronous file mark(s) (tape marks in CAStor jargon) (MTIOCTOP/MTWEOF, in wrttpmrk).
- Write immediate (asynchronous file marks (MTIOCTOP/MTWEOFI, also in wrttpmrk).
- Clear the EOT condition by calling MTIOCGET. This is done in wrtrtlbl, 3 times. In MTIOCGET, indeed, a member of the scsi_tape structure called recover_reg is reset to 0. This clearing is used to properly report errors in label writing functions. The usefulness of this function is dubious and it is not included in the current API.
namespace Tape {
    class Drive {
        public:
        Drive(SCSI::DeviceInfo di, System::virtualWrapper & sw);        /* Direct SCSI operations */
        virtual compressionStats getCompression() throw (Exception);
        virtual void clearCompressionStats() throw (Exception);
        virtual deviceInfo getDeviceInfo() throw (Exception);
        virtual std::string getSerialNumber() throw (Exception);
        virtual void positionToLogicalObject(uint32_t blockId) throw (Exception);
        virtual positionInfo getPositionInfo() throw (Exception);
        virtual std::vector<std::string> getTapeAlerts() throw (Exception);
        virtual void setDensityAndCompression(unsigned char densityCode = 0,
            bool compression = true) throw (Exception);
        virtual driveStatus getDriveStatus() throw (Exception);
        virtual tapeError getTapeError() throw (Exception);
        /* ST driver based operations */
        virtual void setSTBufferWrite(bool bufWrite) throw (Exception);
        virtual void fastSpaceToEOM(void) throw (Exception);
        virtual void rewind(void) throw (Exception);
        virtual void spaceToEOM(void) throw (Exception);
        virtual void spaceFileMarksBackwards(size_t count) throw (Exception);
        virtual void spaceFileMarksForward(size_t count) throw (Exception);
        virtual void spaceBlocksBackwards(size_t count) throw (Exception);
        virtual void spaceBlocksForward(size_t count) throw (Exception);
        virtual void unloadTape(void) throw (Exception);
        virtual void sync(void) throw (Exception);
        virtual void writeSyncFileMarks(size_t count) throw (Exception);
        virtual void writeImmediateFileMarks(size_t count) throw (Exception);
        virtual void writeBlock(const unsigned char * data, size_t count) throw (Exception);
        virtual void readBlock(unsigned char * data, size_t count) throw (Exception);
    } // namespace Tape

Listing 1.6: Tape::Drive interface

- Write is used in 2 places only: twrite and writelbl (which is a specialized function to write 80 bytes blocks). twrite is not checking the size of blocks, which is determined in the calling functions.

- Read is used in tread, which is used in a single place of TapeToMemory. It is also used in readlbl. The latter uses a trick to detect that a tape is blank. This could be turned into a specialized function.

The interface is shown in listing 1.6.

TODO: define end of tape behavior for write (create an exception, and throw it).

TODO: define how detect a blank tape.
1.6.5 The Tape::File class

1.6.5.1 CAStor file format

Over time, CAStor used several file formats, but as of 2013, only one file format is used, called AUL. This format is described on an old CERN web site \(^{22}\), and the general description of the ANSI fields can be found in IBM’s z/OS documentation \(^{23}\).

The AUL format consists of volume label, header blocks and trailer blocks. All those descriptors are contained in tape blocks of 80 bytes. All data is in ASCII nowadays and empty bytes are spaces.

<table>
<thead>
<tr>
<th>Table 1.1: AUL label format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOL1</td>
</tr>
<tr>
<td>one data file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1.2: AUL prelabeled tape with one HDR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOL1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1.3: The structure of the volume label</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOL1</td>
</tr>
<tr>
<td>Bytes</td>
</tr>
<tr>
<td>0-3</td>
</tr>
<tr>
<td>4-9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11-23</td>
</tr>
<tr>
<td>24-36</td>
</tr>
<tr>
<td>37-50</td>
</tr>
<tr>
<td>51-78</td>
</tr>
<tr>
<td>79</td>
</tr>
</tbody>
</table>

CAStor example for the beginning of the tape:

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


### Table 1.4: The structure of the HDR1, EOF1 labels

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

CASStore example for the second file on the tape:

```
00000000 48 44 52 31 31 32 41 31 36 30 33 38 20 20 20 |HDR12A160C38 |
00000001 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 |V5200100010|
00000002 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 |00200001000120410|
00000003 31 32 32 34 31 20 30 30 30 30 30 30 30 30 30 30 |12041 000000CAST|
00000040 4f 52 20 32 2e 31 2e 31 32 20 20 20 20 20 20 |OR 2.1.12 |
```

CASStore example for the empty tape with PRELABEL and one HDR1 is used:

```
00000000 56 4f 4c 31 35 32 30 30 30 20 20 20 20 20 20 |VOL1V52001 |
00000010 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | |
00000020 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 |root |
00000030 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | |
00000040 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 |3 |
00000050 48 44 52 31 35 52 45 4c 41 42 45 4c 20 20 20 20 |HDR1PRELABEL |
00000060 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 |V5200100010|
00000070 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 |0010001000132340|
00000080 31 33 32 33 34 20 30 30 30 30 30 30 30 30 30 30 |13234 000000CAST|
00000090 4f 52 20 32 2e 31 2e 31 32 20 20 20 20 20 20 |OR 2.1.13 |
```
Table 1.5: The structure of the HDR2, EOF2 labels

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

CASTor example for the first file on the tape:

```
00000000 48 44 52 32 46 30 30 30 30 30 30 30 30 30 30 30 30 30 30 20 |HDR2F00000000000 |
00000010 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | |
00000010 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | |
00000030 20 20 30 30 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | 00 |
00000040 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | |
```
### Table 1.6: The structure of the UHL1, UTL1 labels

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

CASStor example for the second file on the tape:

```
00000000 55 48 4c 31 30 30 30 30 30 30 30 30 30 30 30 30 |UHL100000000200|
00000010 30 30 32 36 32 31 34 34 30 30 30 30 32 32 32 31 |00262140002621|
00000020 34 34 43 45 52 4e 20 20 20 58 43 32 44 45 |44CERN_LXC2DE|
00000030 56 35 44 32 53 54 4b 20 20 20 20 20 54 31 30 30 |V5D2STK_T100|
00000040 30 30 42 20 58 59 5a 5a 5f 42 31 20 20 20 20 |00B_XZZY_B1|
```

#### 1.6.5.2 File block management

Some files tapes have mixed block sizes, some files used to have mixed block sizes. Current proposal is to have a fixed block size per tape, and to have operators choose the optimal block size for drive performance (too small blocks reduce performance).

Currently 256kB is used everywhere, so hardcoding this block size for writing to this value is an acceptable for the time being. On the long run, this should be a configurable parameter by the operators.

Ideally, only the Tape::File class should handle all aspects of cutting the disk file, which is a continuous stream, into fixed size blocks. But this would have the downside of having the Tape::File class a client of the FIFOs, and potentially have its own thread, which is far beyond the scope of this class. Therefore, it is the duty of the caller to provide the file cut into fixed size blocks. The Tape::File class will require pre-declaration of the block size, and enforce it.

#### 1.6.5.3 Responsibilities of the class

This class will have the responsibility to check file structure and content, including checksum, block sizes and header/trailer content. In case of non-fatal errors, the warnings will be reported through the warning interface described in 1.6.3.

#### 1.6.5.4 Checksums

The checksum in CASStor uses the Adler32 checksum. Adler32 can be computed incrementally on a stream of data. The zlib contains an implementation of adler32 24. The checksum will be computer automatically when writing or reading the file to tape. Reading a file with a wrong checksum will throw an exception. TODO: define writing behavior (is the checksum pre-declared?).

24http://www.zlib.net/manual.html#Checksum
1.6.5.5 Tape::File API

TODO.

1.6.6 FIFOs

FIFOs will be used to synchronize the data transfer between the tape thread and disk threads. The Tape thread will manage the block-to-stream transformation. The FIFO might not always be able to provide blocks in one piece at chunk boundary. The first attempt solution for this case will be a copy of the cut block. With a chunk size significantly bigger than the block size, the event should be rare enough to not affect performance. FIFOs will probably need some thread safety, but as they will be single user, single consumer, some parts might possible be lockless.

1.6.7 Disk client library

Castor is dropping rfio and is moving to xroot. The effects on the code are to be discussed with Sebastien.

1.6.8 VDQM client library

TODO: describe how we will link with the VDQM client library. The VDQM is also the initial client which triggers the tape sessions. It carries a feature where the tape drive can recycle a tape mount. This is not very useful today, and the first release of the TapeServer will not support it. All sessions will be force-closed by the TapeServer.

1.6.9 VMGR client library

1.6.10 Stager/TapeGateway client library

1.6.11 Logging system client library

In case of failure, operators need to have logs files as much as possible. Thus, castor has an very aggressive logging strategy and every step of execution within the process is monitored.

Each thread has its own log::LogContext, which is holding a set of variables that have to be log in every case with every message. That way, we do bother only once with the variables to log (when adding them to the log context) and can focus on the log message.

The variables are stored within the log::LogContext as log::Param, which is roughly speaking a pair of string for the name of the parameter and its value.

Adding variables is done through LogContext::pushOrReplace, removing them through LogContext::erase and the logging through the log member function. To avoid this cumbersome add/remove, you can use LogContext::ScopedParameter, which is a RAII-class for registering a variable into the LogContext and removing it automatically when its goes out of scope. Using LogContext::ScopedParameter is

```cpp
log::Logger log;
log::LogContext lc(log);
{
  lc.pushOrReplace(cta::log::Param("Name var 1",var1));
  lc.log(LEVEL,message);
  lc.erase("Name var 1");
}
```
1  
2    cta::log::LogContext::ScopedParam sp(lc, cta::log::Param("name var1",var1));
3    cta::log::LogContext::ScopedParam sp2(lc, cta::log::Param("name var2",var2))
4        ;
5    lc.log(LEVEL, message);
6  }

still a bit clunky. To get rid of that, there is the **ScopedParamContainer**, which acts as RAII container for parameters.

```cpp
1  
2    cta::log::ScopedParamContainer sp(lc);
3    sp.add(var1, "name1").add(var2, "name2");
4    lc.log(LEVEL, message);
5  
```

### 1.6.12 Application architecture

#### 1.6.12.1 Memory management and threading architecture

Like the previous version in rtcpd, the tape server will pre-allocate a fixed number of memory blocks for the whole duration of the tape session, and circulate them between the data producers and consumers.

The data flow is organised around block passing, from queue to queue. Each queue output is processed by a thread of thread pool.

The overall layout of a migration mount is shown in figure 1.3. The recall mount, shown in figure 1.4, is almost symmetric. The report packer triggers on threshold instead of tape flushes, and

#### 1.6.12.2 Session spawner

#### 1.6.12.3 Session process

#### 1.6.12.4 Memory chunk manager

The memory block manager allocates (usually all at once) a large chunk of memory. This memory is then shared between the various FIFOs in the system. Deallocation of memory on exit will allow memory leak checks.

#### 1.6.12.4.1 Recall

In case of a recall, the memory manager is pretty much an empty shell. Memory blocks are sequentially pulled by the current tape read task and pushed back to the memory manager once written to the disk

#### 1.6.12.4.2 Migration

In case of a migration, he memory manager has an active role. He has a list of clients that are waiting for some memory blocks. It pushes to them

#### 1.6.12.5 Tape read thread

Behavior : Beginning :

- Load tape
Figure 1.3: Tape server child process: layout of the memory management in the case of a migration mount

- Mount tape
- wait for drive to be ready
- Check the label and position the tape

Run

- Pop a queued task (and ask the task injector to put more tasks in)
- execute

At the end :

- Signal the end to the global status reporter
- Signal to the task injector that he can stop
- attempt of cleaning : unmount and unload the tape
**Figure 1.4: Tape server child process: layout of the memory management in the case of a recall mount**

### 1.6.12.6 Tape write thread

**Behavior**: Beginning:

- Load tape
- Mount tape
- Wait for drive to be ready
- Check the label and position the tape

**Run**

- Pop a queued task
- Execute
- Report to the client if a threshold has been crossed

At the end:
• Signal the end to the global status reporter
• attempt of cleaning : unmount and unload the tape

1.6.12.7 Disk read thread pool

It is an empty shell owning and running a set of disk read threads. Each thread will pop a task and execute it. While popping a task, we ask the task injector to provide more task if we cross on the thresholds. The last thread alive will report the end of the session to the client

1.6.12.8 Disk write thread pool

It is also an empty shell owning and running a set of disk write threads. Each thread will pop a task and execute it. The last thread alive will report the end of the session to the client

1.6.12.9 Global status reporter

1.6.12.10 Task injector

1.6.12.10.1 Recall

1.6.12.10.2 Migration

1.6.12.11 Loopback system

1.6.13 How to interpret the exit code of the mount-session child process

An exit code of 0 means the underlying drive is free to be used again and the tapeserverd parent-process should therefore move the state of the drive to UP (idle in tpstat and FREE in showqueues).

A non-zero exit code means the tapeserverd parent-process should move the state of underlying drive to DOWN. An operator will take a look. Under no circumstances should the tapeserverd daemon launch a Leon/Victor process to clean the situation.

If the mount-session child-process terminates without an exit code (it received a terminating signal) then the Leon/Victor process should be launched.

1.7 Compilation instructions

Install SLC 6 (or 5) Install cmake and git if they are not installed out of the box

```bash
$ yum -y install python-debian cmake git
```

```bash
$ cd
git clone http://git.cern.ch/pub/CASTOR castor
mkdir build && cd build
cmake -DPackageOnly:bool=true ..;/castor/
make castor_rpm
```

The last command will give you a list of missing packages that are needed. Most of them can be installed with a simple yum install when epel repository is enabled.
yum install -y libattr-devel binutils-devel python-devel xfsprogs-devel gtest
gtest-devel globus-gridftp-server-devel globus-common-devel cppunit-devel
e2fsprogs-devel openssl-devel krb5-devel curl-devel zeromq3 zeromq3-devel
protobuf-compiler protobuf protobuf-devel valgrind

But you need get and install manually from swrep at least the following packages :
- gmock-1.5.0-3
- gmock-devel-1.5.0-3
- libmemcached-0.53-1
- libmemcached-devel-0.53-1
- oracle-instantclient-basic-11.2.0.3.0-5
- oracle-instantclient-precomp-11.2.0.3.0-5
- oracle-instantclient-devel-11.2.0.3.0-5
- stk-ssi-devel-2.3-3.cern
- stk-ssi-2.3-3.cern

Ceph is also need. The following command should take care of that :

You may install ceph-debuginfo (2.8 Go) if you need to.

After that, you can run again
You should have no errors (if some packages are still missing, install it) otherwise it should compile castor
and create the rpm package.

A classic compilation is done with the usual make. To run a set of tests make test has to be used.

1.7.0.1 Compiling on SLC6

On SLC6, there is clash between oracle-* and postgresql-devel packages about sqlca.h. The package
postgresql-devel provides a new file sqlca.h, which has several significant differences with the one provided
by the oracle-* packages, thus leading the compilation to fail.

http://swrep/swrep/
```bash
make castor_rpm
```

## 1.7.0.2 Compiling on SLC5

On SLC5, an empty line is required at the end of each file.
Chapter 2

Administrator’s manual

2.1 User and capabilities

Castor used to run as root, which is not the best safety policy. New version should be run by stage:st (even if still own by root), because the first action will be to drop the root’s privileges to move to stage:st. But accessing /dev/nst* for writing data requires to either be root (on SLC5) or to have the capability CAP_SYS_RAWIO set on. Here the incriminated piece of code into the st driver:

```c
switch (cmd_in) {
    case SCSI_IOCTL_GET_IDLUN:
    case SCSI_IOCTL_GET_BUS_NUMBER:
        break;
    default:
        if ((cmd_in == SG_IO ||
             cmd_in == SCSI_IOCTL_SEND_COMMAND ||
             cmd_in == CDROM_SEND_PACKET) &&
            !capable(CAP_SYS_RAWIO))
            i = -EPERM;
        else
            i = scsi_cmd_ioctl(STp->disk->queue, STp->disk, 
                               file->f_mode, cmd_in, p);
        if (i != -ENOTTY)
            return i;
        break;
}
```

The cleanest way to do it seems to allowed the one we need on the main-binary and the acquired them in the forked process.

2.2 Pending questions

- Is the option ST_BUFFER_WRITES from castor.conf still used?
- Why does /etc/castor/TPCONF have a tape density column? In order to ject incompatibel tapes from being mounted.