Fachhochschule Wiesbaden, University of Applied Sciences
Fachbereich Informatik

Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker (FH)

Redesign and Functional Extension of the Robotic Tape Queueing System within the CASTOR HSM

vorgelegt von Matthias Bräger
am 19. Dezember 2005

Referent:	Prof. Dr. Detlef Richter, Fachhochschule Wiesbaden
Korreferent:	Dr. Olof Bärring,
	CERN (European Organization for Nuclear Research), Genf

Abstract

High Energies Physics is a research area which provides several data-intensive scenarios at petabyte-scale level. Current simulations predict for the experiences of the Large Hardrons Collider [LHC], which is expected to be online in 2007, a data raising of fifteen petabytes per year. We are therefore, challenged to expertise new ways to catalogue, store, locate and retrieve in a timely manner.

In this thesis we present an extension of **C**ERN's **A**dvanced **STOR**age Manager [CASTOR], the hierarchical mass storage system developed by CERN. An introduction to the field is given and alternative mass storage solutions are outlined.

We elaborate a description of CASTOR that helps us to understand the context of the Robotic Tape Queueing System, which has been reimplemented.

An exact analysis of the problems is made and technical extensions explained to a relevant level. This is required to work out in detail the system design.

The implementation is discussed and significant code examples presented.

We conclude with an evaluation of the redesign and give an outline of potential future developments.

Contents

List of Figures

Chapter 1

Introduction

In this Chapter we give our motivation for doing a revision of the mass storage project at CERN, and an overview about the work presented of this thesis.

It has been developed within six month at CERN, enhanced of a three month internship at the beginning.

1.1 Motivation

The European Organization of Nuclear Research [CERN] is located on the border between France and Switzerland, close to Geneva. It is the most important laboratory for High Energies Physics [HEP] worldwide. Several accelerator facilities have been in operation during the last fifty years, such as the Large Electron Positron Collider [LEP] from 1981 to 2000 which performed electron-positron collisions.

The Large Hardrons Collider [LHC] is currently being built. This is a 26 km long accelerator facility, located 100 meters under the surface (see Figure 1.1). Its first run is expected in 2007.

In this framework, four different HEP experiments are presently being constructed to make use of the capabilities of the LHC accelerator:

- ALICE - A Large Ion Collider Experiment
- ATLAS - A Toroidal LHC ApparatuS
- LHCb - LHC study of Charge Parity [CP] violation in B-meson decays

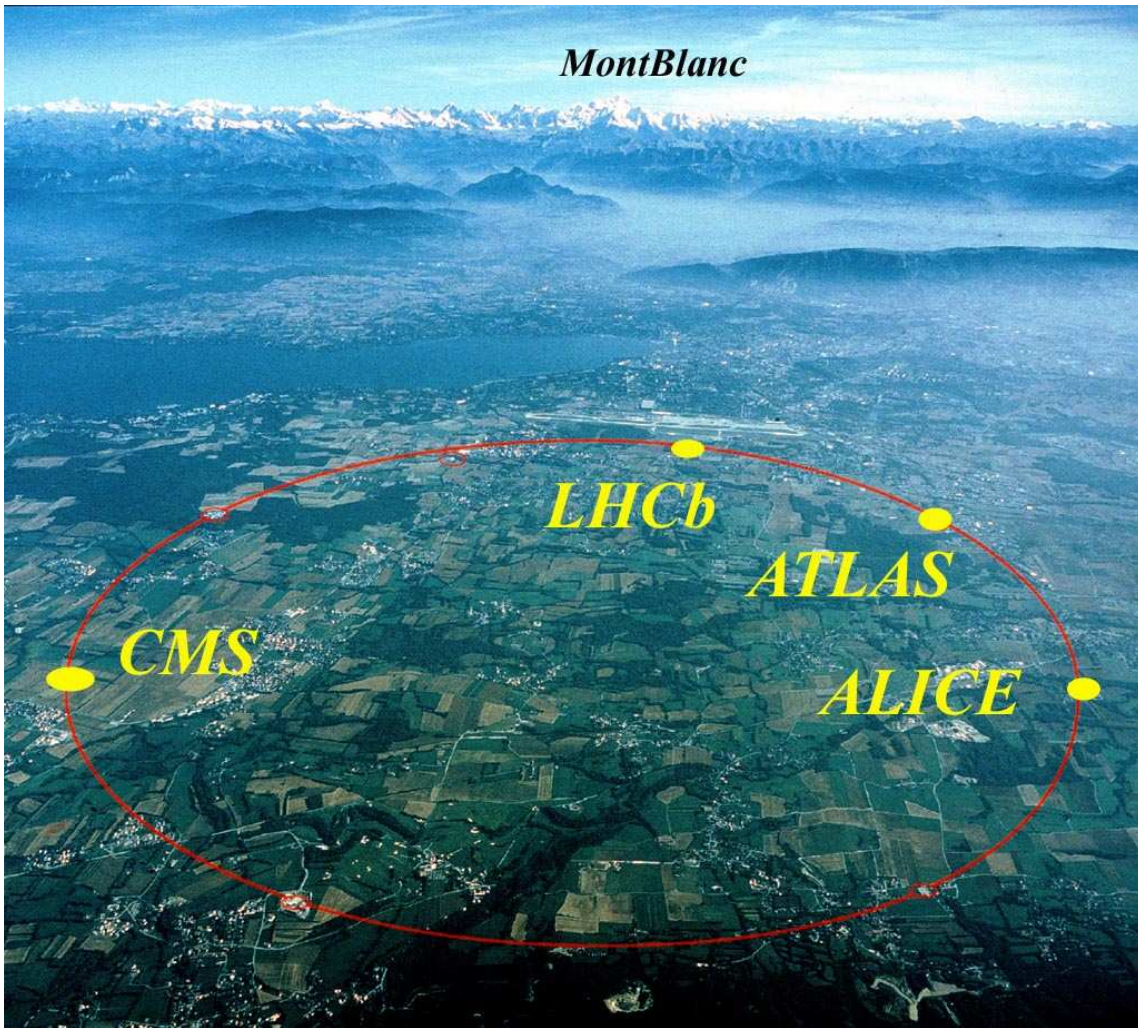

Figure 1.1: Aerial view of the CERN LHC accelerator [1]

- CMS - Compact Muon Solenoid

A detailed description of these experiments is out of the scope of this thesis and would involve multiple PhD thesis!

With the start of the HEP experiments in 2007, CERN will produce a total amount of fifteen petabytes of data per year. This represents a big challenge not only for the hardware and storage infrastructure, but also for the involved software components.

The CERN Advanced STOrage manager [CASTOR] is used to store this information to tape. CASTOR has been in production since 2001, but, due to the fast increasing amount of data it has reached its technical limits. Hence, a revision of the architecture is necessary to meet the challenge of the future tasks.

1.2 Thesis Overview

The intention of this work is to extend the robotic tape queueing system of CERN's mass storage application CASTOR2. Therefore, a reimplementation of the Volume and Drive Queue Manager [VDQM] component is necessary.

The key goals of the thesis are:

- Analysis of CASTOR's current robotic tape queueing system (VDQM)
- Information gathering about limitations and new requirements
- Design and implementation of an extended, backward compatible system

1.3 Thesis Structure

Chapter 2 describes the storage components, currently used and tested in the CERN Computer Centre. It provides an introduction to the fabric management tools which are used for maintaining the hardware. We present furthermore, two standard Grid Interface Protocols that are supported by CASTOR.

Chapter 3 presents alternative storage managers to CASTOR and explains the reasons why they are not used.

Chapter 4 gives an introduction to the CASTOR system. The collaboration of its main components is explained by describing the steps to store/retrieve files to/from the robotic tape library. Additionally, architectural details of the reimplemented core system are outlined.

Chapter 5 acquires a detailed technical analysis of the current robotic tape queueing system. The requirements of the reimplementation are presented and solutions for the major problems are evaluated.

Chapter 6 gives an overview about the use cases of each client component, supported by the reimplementation.

Chapter 7 describes in detail the design of the new architecture.

Chapter 8 summarise the implementation process and explains some code details.

Chapter 9 contains the conclusions of this work, and descriptions of practical extensions and future prospects.

1.4 Prerequisites

To understand in detail the work of this thesis, the readers should be familiar with basic network protocols, such as the Transmission Control Protocol [TCP] and the User Datagram Protocol [UDP]. This is particularly needed to understand the introduction to the Grid technologies and the communication protocols of the presented mass storage systems.

Since the application is written in an object oriented manner the readers should have expertise in modern software development paradigm, such as Software Design Patterns. Especially in the description of the detailed design, the technical terms are used without extensive explanation.

Additionally, this document contains a Chapter describing some bigger code examples. Therefore, it will be taken for granted that the reader has basic programming knowledge in C/C++.

To aid the reader, we highlight *class names* and *function names*, which are mentioned in the text in italics as shown. All acronyms used are explained at their first use and can be found in the Glossary. Further references are provided by reference number and can be found at the end of the thesis.

The contents of this document can be freely used and distributed providing the author and the source is referenced.

Chapter 2

Technical Basics

CASTOR is a modular system with multiple software components running on various machines. Due to an increasingly amount of soft- and hardware, the administration becomes a difficult issue. Tools are necessary to deploy, manage, and to monitor the machines and all accumulating logs and real data at CERN. This Chapter will give an overview about those which are used in combination with CASTOR.

Beside the software challenge, CERN has to upgrade their robotic tape libraries to face the expected data from the Large Hardron Collider [LHC] experiments in 2007. This is likely to amount to tens of petabytes.

Thus, the first Section will give an overview about the actual used storage hardware and the robotic tape libraries evaluation for 2007.

2.1 Storage Components

The first commercially successful tape drive appeared in 1952 and was the Industrial Business Machine [IBM] 726 Tape Unit for the IBM 701 Defense Calculator. The data were recorded at a density of 100 bytes per linear inch (byte/in) of half-inch-wide magnetic tape. The IBM 701 cost over $1 million, used magnetic core memory, was priced at $1 per bit, and required refrigeration. Since then a lot of new fabrications have been made [6].

In 2004, Toshiba introduced the first sub-1-inch tape drive with a 0.85-inch diameter, 3,600 rotations per minute [rpm] in both 3 GB and 4 GB capacities. The 400 GB Linear Tape Open 3 [LTO3] Ultrium tape drive IBM has been replaced by the recent announcement of a 500 GB tape drive by Seagate. IBM displayed the first native 1 TB tape cartridge containing 1,216 tracks and 775 meters of media [6].

Model	Capacity, native (uncompressed)	Average file access time (first file)	Data transfer rate, native (uncompressed)
T9840A	20 Gb	8 sec	10 Mb/sec
T9840B	20 Gb	8 sec	19 Mb/sec
T9840C	40 Gb	8 sec	30 Mb/sec
T9940A	60 Gb	41 sec	10 Mb/sec
T9940B	200 Gb	41 sec	30 Mb/sec
LTO Gen 1	100 Gb	86-96 sec	15-16 Mb/sec
LTO Gen 2	200 Gb	64-75 sec	32-35 Mb/sec
LTO Gen 3	400 Gb	72 sec	80 Mb/sec
SDLT 320	160 Gb	82 sec	16 Mb/sec
SDLT 600	300 Gb	79 sec	36 Mb/sec
3592 J1A	300 Gb	data not available	40 Mb/sec
T1120 (new)	500 Gb	data not available	100 Mb/sec

Table 2.1: Overview about the most common tape drive models [7, 8, 9]

Table 2.1 gives an overview about the characteristics of todays most common tapes drives. The tape drives are separated with a horizontal line from models with a different tape cartridge type. Newer tape drives are able to read tape models of older tape drives if they belong to the same cartridge family, but they can not necessarily write to them.

At the moment CERN has four Powderhorn 9310 robotic tape libraries from StorageTek [STK] , placed for safety reasons in two different buildings. Each library has 22 9940B tape drives and space for up to 6,000 tapes. This gives a maximum total storage capacity of 600-1,200 terabytes.

Due to the Large Hardron Collider [LHC] experiment starting in 2007, CERN will have to store over 15 petabytes of data per year and will need to support an average throughput transfer rate of four gigabytes per second. Apart from existing storage resources, - being insufficient, the currently installed Powerderhorn tape libraries can only provide a total throughput of one gigabytes per second, and this only if we disregard the idle time during tape exchange. To fullfill these needs in the near future, CERN evaluates currently the newest robotic tape library generation of IBM and STK.

StorageTek's system is the modular library system StreamLine SL8500. The basic library storage module [LSM] has space for 1,448 tape cartridges and 64 tape drives. It is possible to connect up to 32 LSMs to a single system and to reach a maximal configuration of 300,000 customer-usable slots and 2,048 drives [7]. Each LSM is directly connected to a library control unit [LCU], which controls the robotics motion and interfaces to the drives through the LCU-to-drive path [10]. The LSM components can be interconnected with a Pass-Thru Port [PTP] system. The PTP is used by the robotic arm to pass tape volumes from one LSM to another connected LSM. An advantage of this is that tape drives can be used more efficiently. If each experiment had instead its own library, some could be very busy and others idle.

IBM provides the TotalStorage 3584 Tape Library solution, with a maximum capacity of 6,260 tape cartridges and up to 192 tape drives. Compared to the STK system the dimension of this library is much smaller. On the other hand, IBM just presented the world fastest tape drive, the IBM TS 1120 with an average transfer rate of 100 MB/sec and a capacity of 500 GB on the IBM tape cartridge 3592.

The final decision about the robotic tape library setup at CERN will be made in the autumn of 2006.

2.2 Fabric Management Tools

In the last years of the European DataGrid [EDG] project, developers and service managers have been working to understand and solve operational and scalability issues of Grid technology.

The project was funded by the European Union to build the next generation computing infrastructure, provide intensive computation and analysis of shared large-scale databases.

In this period three sub projects were developed to administer and maintain huge clusters, which were summarised under the name "Extremely Large Fabric management system" [ELFms] [11] . This comprises:

- System Administration Toolsuite [Quattor] [12]
- LHC Era Monitoring [Lemon] [13]
- LHC-Era Automated Fabric [Leaf]

This report factors out the Leaf project, which is the hardware and state management system for the Large Hadron Collider Experiment [LHC]. For an understanding of CASTOR, this topic is not necessary.

2.2.1 The Quattor System Administration Toolsuite

Quattor is a system administration toolsuite for automated installation, configuration and management of clusters and farms running with Linux and Solaris operating systems [12].

Quattors development was started by EDG in 2001 and handed over to CERN (IT department) after two years in the scope of ELFms. It is now used to manage over 2600 Linux nodes, out of approximately 3400 nodes in the CERN Computer Centre. The managed nodes can be, for instance, tape servers, database servers or web servers.

The Quattor information model distinguishes between the *actual* state and the *desired* state. The desired state of the machines is registered in a fabric-wide Configuration Database [CDB] , using a special designed configuration language called PAN. The state information in CDB are hierarchically structured in building blocks called *templates*, which are validated and kept under version control in CVS . The hierarchy structure allows to build service structures as shown for instance in Figure 2.1.

There are different ways to create a template and store it into CDB. The most common way is to use existing scripts, which send the PAN file via the Service-Oriented architectural pattern [SOAP] interface to CDB. Another way, in the future, is to use a graphical user interface, which is not yet fully implemented.

This architecture makes it possible to detect conflicts of concurrent modifications of the same configuration information and to go back to a previous version of a template.

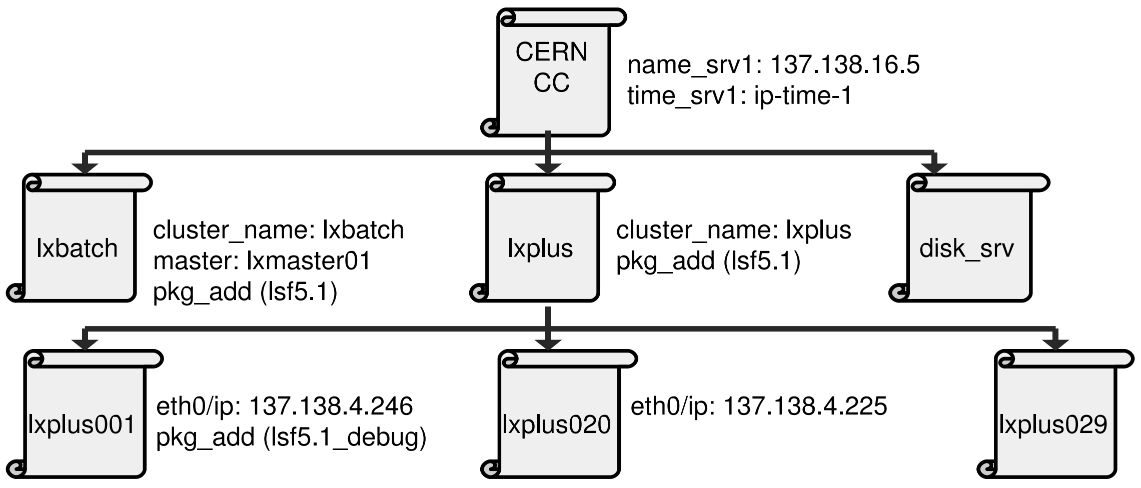

Figure 2.1: Example of a hierarchical service structure [2]

A special PAN compiler translates the configurations in CDB into XML-files, which are propagated to, and cached on, the affected nodes in the cluster.

Figure 2.2 gives an overview about this main architecture. As this shows, CDB provides an SQL interface, which enables it to run standard SQL queries on configuration information. This is used by Lemon to obtain information about the services of the fabric nodes.

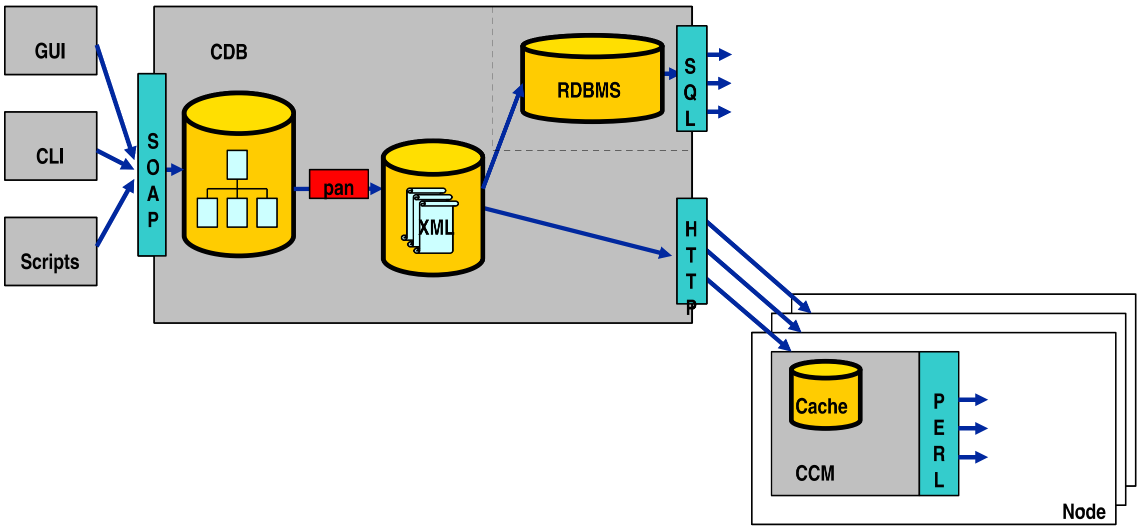

Figure 2.2: Quattor's configuration management infrastructure [2]

Each node has a Configuration Cache Manager [CCM] , which is responsible for downloading the information from CDB and storing it in the local cache. The main advantage of this architecture is the avoidance of demand peaks on CDB side.

A locally running Software Package Management Agent [SPMA] handles matching the desired state of the node. With the aid of a system packager, like RPM or PKG , the SPMA installs or removes software packages, as in the information provided by the CCM over its PERL interface.

Another subsystem is needed to configure or reconfigure local system and Grid services. This subsystem is called Node Configuration Manager [NCM] , which like SPMA uses the local CCM cache.

2.2.2 The *wassh* Shell-Command

Wassh is used internally at CERN to run remote shell commands in parallel on sets of hosts via ssh. The degree of parallelism can be specified by the user. The output of each remote host is returned in random order and printed out to the standard output as if the shell command had run on each node sequentially. A desired side effect of the parallel execution is a high performance increase.

The syntax of wassh looks as follows:

```
wassh [<options>] [<targets>] <shell-command>
```

Usually it is just used like in this example:

```
wassh root@lxbatch <shell-command>
```

This command would run as root user a specified shell-command on all machines of the lxbatch cluster.

2.2.3 The Large Hardron Collider Era Monitoring System

"Lemon (LHC EDG monitoring) is a client/server based monitoring system." [14] It serves to provide monitoring information about the farms in Computer Centres as well as on a normal user PC. Moreover, Lemon can also be used to provide a framework for recovery actions and alarms. At CERN it is mostly used for general monitoring issues and for system administrators and programmers in debugging problems.

The clients are represented by monitoring agents on each single monitored node, which retrieves monitoring information and provides them over a push/pull protocol with sensors to the Lemon Server. The local cached information is forwarded to a central Measurement Repository.

For a better understanding of the whole Lemon structure and the following component specifications, please see the schema in Figure 2.3.

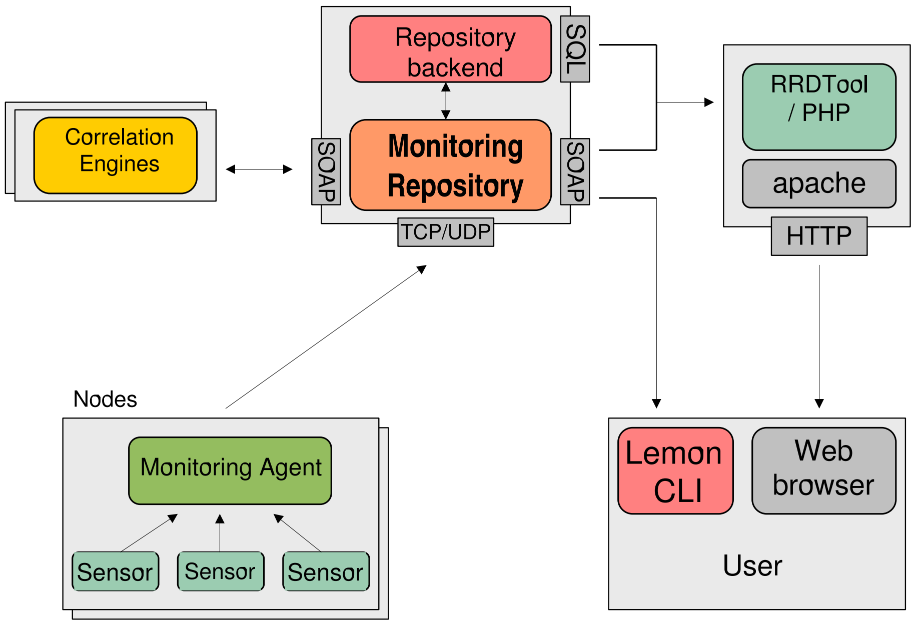

Figure 2.3: Lemon architecture schema [3]

Sensor

Sensors are applications, which collect individual metrics from hard- and software components and convert them into human readable values. They are used for instance to monitor remote entities like switches or power supplies. The measurements are based on the requests of the Monitoring Sensor Agent [MSA]. Sensors are easy to implement and to include into an existing system, so the users can always invent new ones. On average there are about 70 metrics measured on approximately 2600 nodes in the CERN Computer Centre. In total they collect about one gigabyte of data per day. Several sensors exist for database monitoring, performance measurements and soft- and hardware monitoring.

Monitoring Sensor Agent [MSA]

The Monitoring Sensor Agent [MSA] is a daemon which runs on each client machine and spawns the sensors to measure metrics in defined time intervals. The data are sent back from the sensors through a pipe and then forwarded asynchronously via an UDP or TCP connection to the Monitoring Repository on the Lemon server.

Monitoring Repository [MR]

The MR receives all collected samples from the MSAs and stores the full monitoring history in an Oracle database or alternatively into a flat file, having validated the data set. Data is never deleted but data of only historical interest is archived on the Tivoli Storage Manager [TSM] and CASTOR.

The data can be accessed using the SOAP interface, or directly from the repository backend via SQL.

MR allows to plug-in correlations accessing collected metrics and external information. Examples for correlation engines can be found for instance for Quattor CDB or LSF (Large Scale Facility) . Another feature of MR is the launch of configured recovery actions.

Lemon RRD Framework [LRF]

To transform the collected data from the client nodes to a format which can easily be used for virtualizations, another framework called Lemon RRD framework is needed [LRF]. LRF is based on the RRDtool project and preprocesses the data into RRD files which it gets over the SOAP interface of the MR.

RRD is the acronym for Round Robin Database [15]. It is a system to store and display time-series data, like those collected by Lemon. RRD stores the data in a very compact way that will not expand over time. Furthermore RRDtool provides an API to create graphs of the data samples. These graphs are then passed over the web interface for virtualization by a web browser.

The framework of LRF is generic enough to accept data from other sources than MR. It is capable of group the metrics of machines together and to provide summaries of each group.

Another way to access the monitoring data from MR is to use the Lemon command line interface [LCLI] . LCLI will connect to MR and retrieve the desired data set, which can be specified using several options.

2.3 Grid Interfaces

Scientific and engineering Grid applications require both the transfer of large amounts of data (terabytes) between geographically distributed storage locations and remote access to large data sets. There are already a number of storage solutions in use, but very often these were just designed to satisfy the individual needs of the users. Unfortunately, most of these don't use standardised communication protocols or don't publish them.

CASTOR2 supports two universal Grid data transfer and access protocols. This Section will outline their main features.

2.3.1 Grid File Transfer Protocol

"The Grid File Transfer Protocol [GridFTP] is a high-performance, secure, robust data transfer protocol optimised for high-bandwidth, wide-area networks." [16]

The protocol is defined by Global Grid Forum Recommendation GFD-R-P.020 [17], RFC 959, RFC 2228, RFC 2389, and a draft before the IETF FTP working group. GridFTP extends the standard File Transfer Protocol [FTP] protocol, which is most commonly used data transfer protocol on the Internet. An advantage of using FTP as basis is that it is a widely implemented and well-defined architecture.

On top of this GridFTP provides additional features, from which the most important ones are explained below:

Grid Security Infrastructure [GSI] and Kerberos Support

One of the biggest aims of GridFTP was to support the Grid Security infrastructure [GSI] and Kerberos authentication. This is very important because robust authentication, integrity and confidentiality are critical issues when transferring or accessing files. The goal was reached by implementing the GSSAPI authentication mechanism, defined the "FTP Security Extension" [RFC 2228]. GSSAPI is an acronym and stands for Generic Security Services Application Programming Interface.

Third-Party Control of Data Transfers

As shown in Figure 2.4 GridFTP permits the user to start, control and monitor a data transfer between two remote servers. This feature is useful, if the user wants to mangage large data sets for distributed communities. Furthermore it allows the use of dedicated transfer nodes without doing a log in, which simplifies the users's workflow.

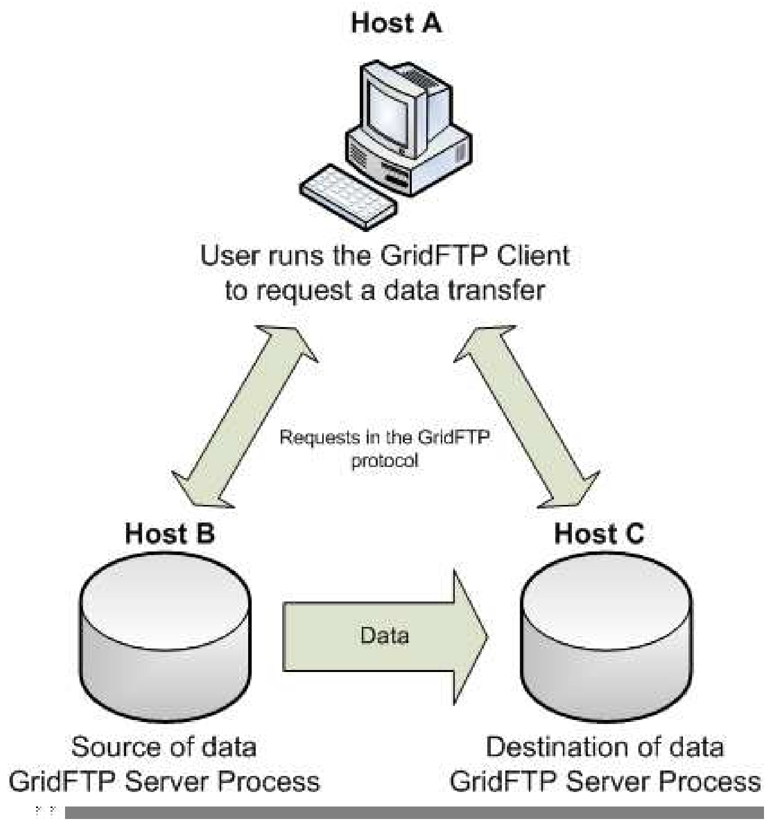
Host A
User runs the GridFTP Client to request a data transfer
Requests in the GridFTP protocol
Host B
Host C
Data
Source of data GridFTP Server Process
Destination of data GridFTP Server Process

Figure 2.4: Third party control of data transfer with GridFTP

Therefore the third-party authenticates on a local machine (Host A) and launches the GridFTP Client application. GSSAPI operations manages the authentication of the third-party on the source (Host B) and destination (Host C) servers for the data transfer.

Parallel Data Transfer

To achieve better performance results on wide-area links, GridFTP uses multiple TCP streams in parallel. This is even possible between the same source and destination machine or the same file. GridFTP supports the parallel data transfer through FTP command and data channel extensions.

At CERN, GridFTP is used to transfer data to/from the mass storage system CASTOR from outside of the CERN network. In the local network, these functionalities are provided by a specific client (see Chapter 4). To upload or retrieve files from outside of CERN several steps have to be performed, which is described in detail on the CERN IT Department homepage [18]:

1. The user has to have a valid personal certificate issued by a LCG recognised certificate authority. LCG is the LHC Computing Grid project, which is run from CERN.

2. For automatic mapping of user's distinguished name [DN] to the Unix group account on CERN's provided GridFTP service, the user has to register the certificate on one of the LCG virtual organisations, which originate from the CERN experiments and projects, like ALICE, ATLAS or CMS.

3. Once all necessary software has been installed and the required configurations are done, the user must generate a Grid proxy certificate by typing this command into his shell:

   ```
   > grid-proxy-init
   ```

   This is intended for short-term use, when the user is submitting many jobs and cannot be troubled to repeat his password for every job. The default expiration for the proxy certificate is twelve hours.

4. To upload files to CASTOR, the following commands have to be used, as shown in the following example:

   ```
   > globus-url-copy file:///tmp/testfile \
     gsiftp://castorgrid.cern.ch/castor/cern.ch/grid/\
     experiment/datafiles/testfile
   ```

   To retrieve files from CASTOR, the user has to type these commands:

   ```
   > globus-url-copy \
     gsiftp://castorgrid.cern.ch/castor/cern.ch/grid/\
     experiment/datafiles/testfile \
     file:///tmp/testfile
   ```

2.3.2 Storage Resource Management

The Storage Resource Management [SRM] is a middleware component for managing shared storage resources on the Grid. Its function is to provide dynamic space allocation and file management [4]. The advantage of SRM is the uniformed access to heterogeneous storage elements as shown in Figure 2.5. The use of SRMs simplifies file managing by abstracting the access to the storage, so that the user doesn't has to know where and how his data are exactly stored. This allows also designer to concentrate on function rather than on compatibility with all systems involved.

Additional features of SRM are protocol negotiation and dynamic transfer URL generation. This allows support for multiple transfer protocols. At the moment CASTOR has implemented the GridFTP protocol, so that there is not really a choice when using it together with SRM.

The abstract functionalities of SRM are specified in actual two existing protocol specifications:

- SRM v1.1 provides
  - data access and transfer
  - implicit space reservation
- SRM v2.1 adds to the first version of the protocol

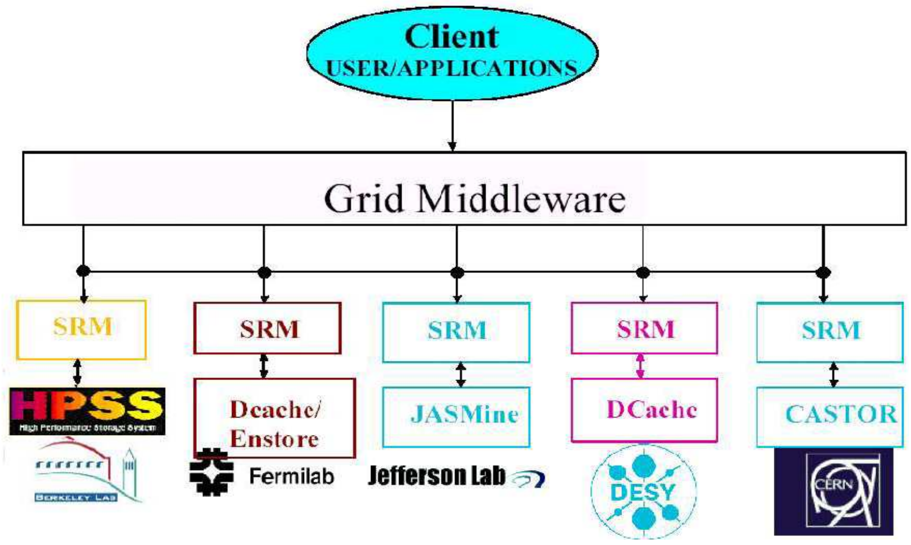

Figure 2.5: SRM as a uniform storage interface [4]

– explicit space reservation

– namespace discovery and manipulation

– access permission manipulation

SRM v1.1 was implemented and finalised in 2001 by LBNL[1], JLAB[2], FNAL[3] and CERN. It mostly consists of the definition of the data transfer functionality. The knowledge gained during implementation of the first version has allowed to define the more complete interface SRM v2.1. CASTOR supports at the moment only the first protocol version, because an implementation of the v2.1 protocol would necessitate bigger changes in the design structure of CASTOR.

Apart from that, the SRM interface allows direct file transfer from one mass storage system to another. This qualifies storage systems to be either the client or the server. Replication schedulers use this feature to take advantage of their knowledge of the system operating parameters, like the current network load or the availability of the system resources.

[1]Lawrence Berkeley National Laboratory

[2]Jefferson Lab, managed and operated by Southeastern Universities Research Association [SURA] for the U.S. Department of Energy [DOE].

[3]Fermi National Accelerator Laboratory

Chapter 3

Alternative Storage Managers

CERN has several reasons to implement its own Mass Storage System. One reason is to be independent from a single proprietary solution and licence costs. Looking more closely at existing products it is also very difficult to find a good solution, which fits all the needs of CERN's storage requirements and its huge amount of data. Ideally CERN was looking for a solution which can easily be adapted to the needs of users. Thus, a good solution has to be able to manage following needs:

- Storing a very high amount of data per second (over 1 gigabyte/s)
- Deliver data to other storage systems and Tier one institutions
- Being compatible with recent Grid standards, such as GridFTP and/or SRM
- Providing 99.999 percent of stored availability
- Being able to manage more than 15 petabytes of data per year and a transfer rate of 4 gigabytes per second
- Giving, at any time, access to all stored data
- Easily to be adapted to customers needs

The next sections will present alternatives to CASTOR and explain why they cannot realistically be used at CERN.

3.1 Disk Cache

The goal of the Disk Cache project (dCache) is to "provide a system for storing and retrieving huge amounts of data, distributed among a large number of heterogeneous server nodes, under a single virtual filesystem tree with a variety of standard access methods"[19]. The software was jointly developed by the Deutsches Elektronen-Synchrotron, DESY and the Fermi National Accelerator Laboratory, FNAL .

Two main features which are provided by dCache and are useful in a Grid environment are:

1. The reservation and guarantee of storage availability for large data sets.

2. A garbage collection to ensure that failed operations or missbehaving clients do not permanently freeze the storage resources and to prevent other user from accessing it.

As shown in Figure 3.1, dCache strictly separates the filename space of its data repository from the physical data set location. Internally the filename space is managed by a database. The location of a file can be on the hard disk of one or more dCache nodes or on tertiary storage system. These are contacted, if a requested file can not be found on the local disk storage.

Disk Cache provides for the users a native access protocol, called dCache Access Protocol (dCap). dCap is shell based and supports the basic file access functionalities. In addition to this native access, various File Transfer Protocols are supported, e.g. the Kerberos based Generic Security Services File Transfer Protocol [GssFTP] and Grid File Transfer Protocol [GridFTP] [20].

As mentioned, the storage of dCache is purely based on hard disks, which allows a fast access to the stored data. A disadvantage of this solution is that the costs of hard disk space is higher than tape space. This is a big issue, especially when it is necessary to store more than 15 petabytes per year.

The Disk Cache system is used at CERN in addition to CASTOR, to store user data which are frequently accessed. CASTOR is connected to dCache as a tertiary storage manager via the Storage Resource Manager interface.

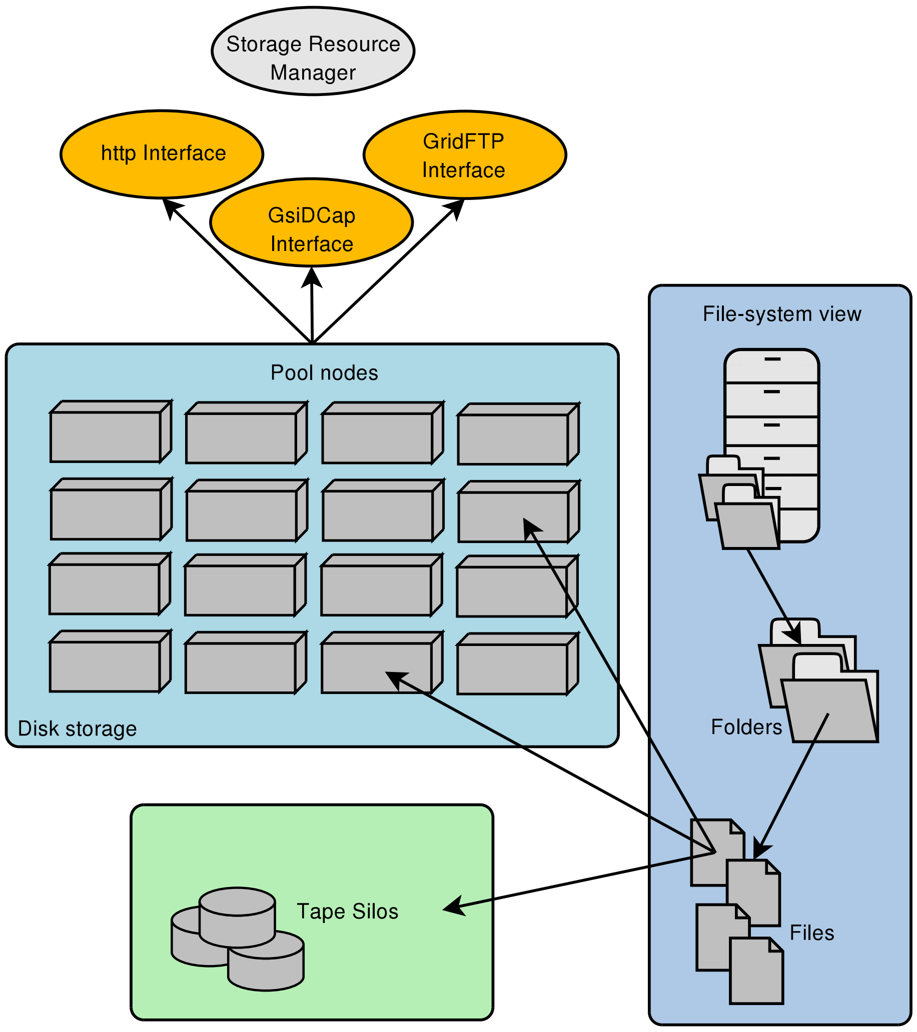

Figure 3.1: Overview of the dCache component architecture

3.2 Enstore

Enstore [21, 22] is a mass storage system developed and used at Fermi National Accelerator Laboratory (FNAL) to provide distributed access to data on robotic tape libraries. As a disk caching front end, FNAL uses dCache, Enstore just handles tape storages.

The project was started to support the FNAL Collider Run II experiment, which needed an aggregate storage requirement of 250 Mbytes/s with an uninterrupted throughput for one month. To reach aims to the preceded example, the system has to be robust enough to sustain failures of hardware elements and has to support the addition of new equipment to scale its capacity and rates.

The main components of the Enstore software is presented in Figure 3.2. It is based on a client-server architecture with a generic user interface. All components communicate over UDP based interprocess communications (IPC), which guaranties addressability under extreme load conditions.

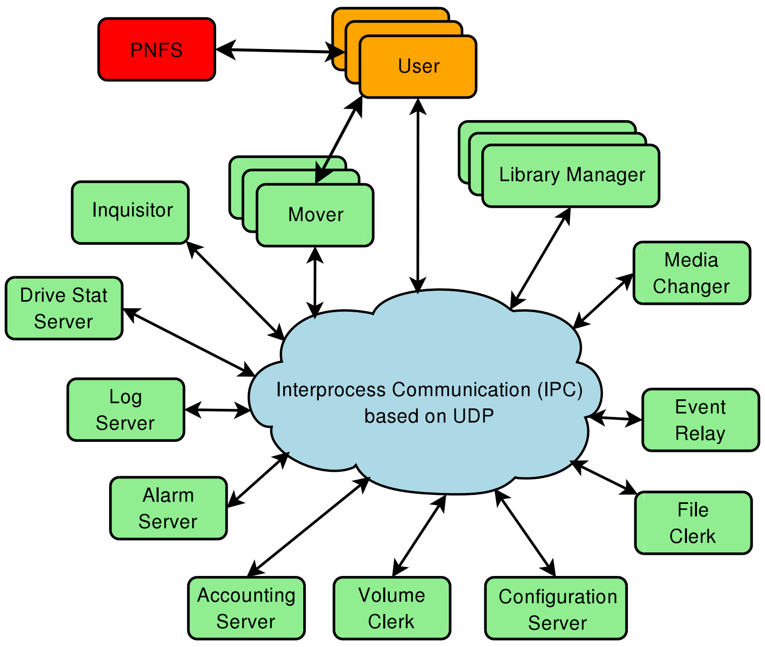

Figure 3.2: Enstore component overview

Following components are needed for the Enstore system:

- A configuration server, which provides system configuration information to the rest of the system. The information is stored in easy maintainable files.

- A volume clerk is responsible for the volume database management. It handles the declaration of new volumes, the assignment of volumes, user quota, and volume bookkeeping.

- The file database is maintained by a file clerk component, which assigns unique file IDs and keeps all required information about files stored into the robotic tape libraries.

- Movers transfer user data to tapes and handle read requests.

- Multiple distributed library managers provide queueing, optimisation, and distribution of user requests to assigned Movers. A Mover can be assigned to more than one library manager.

- A Media Changer handles the mounts and dismounts of tapes, which are requested by the Movers.

- Every component can send an event to the alarm and log servers to trigger the generation of an alarm or log message.

- An Accounting Server manages information about the failed and completed data requests in a database.

- Information about the tape drives is maintained by the drive status server [Drive Stat Server], which also makes use of a database.

- The actual state of the Enstore components is monitored by an Inquisitor.

- The Perfectly Normal File System, or just PNFS , is an independent piece of software developed at DESY . It provides a name space that externally appears as a set of Network File System. The users access files through PNFS names, which are maintained in a database together with some additional metadata information about the files.

- Events are used in Enstore as communication instrument to inform components about ongoing changes in the system. An Event Relay transmits these events to its subscribers.

Enstore has two grouping methods for data. First of all, it is possible to specify the amount of tapes reserved for an experiment. To delegate the incoming data to the right tapes, each request possesses a unique group ID. Additionally, a user can group files together in a so-called file family and has the feasibility to control the amount of simultaneous write transfers by a family. Enstore tries to store data belonging to the same file family on the same tape, which improves a later read access.

The user command interface, encp, is similar to the UNIX cp copy-command with some additional features. It allows users to specify processing commands such as priority, tape dismount time or number of retries.

In conclusion, Enstore is a very modular and robust system which, in combination with dCache, fits the needs of an experimental environment with a high data quantity. The reason not to use Enstore is that the amount of accumulated data at CERN will be four times higher than at FNAL, and the transfer rate will be 16 times higher! Another issue is that Enstore would have to be changed to support tape drives with a data transfer rate of 100 megabytes per second. At CERN Enstore would most likely reach its architectural limits.

3.3 High Performance Storage System

"The High Performance Storage System (HPSS) provides scalable hierarchical storage management (HSM), archive, and file system services."[23] Its implementation was started in 1992 by a collaboration of five laboratories of the United States Department of Energy and IBM.

HPSS is designed to manage very high amounts of data on disk and robotic tape libraries, in the range of petabytes. It emulates a virtually unlimited disk space, which uses one unified name space. For the users it hides the complex storage structure and appears as one single storage service. The system keeps recently used data on disk, while the rest is written to tape. Figure 3.3 illustrates the conceptual architecture of HPSS, which uses cluster and Storage Area Network (SAN) technology to aggregate the capacity and performance of many computers, disks, and tape drives.

The information about the data are stored on a database on metadata disks and are strictly separated from their physical location. To manage information, several

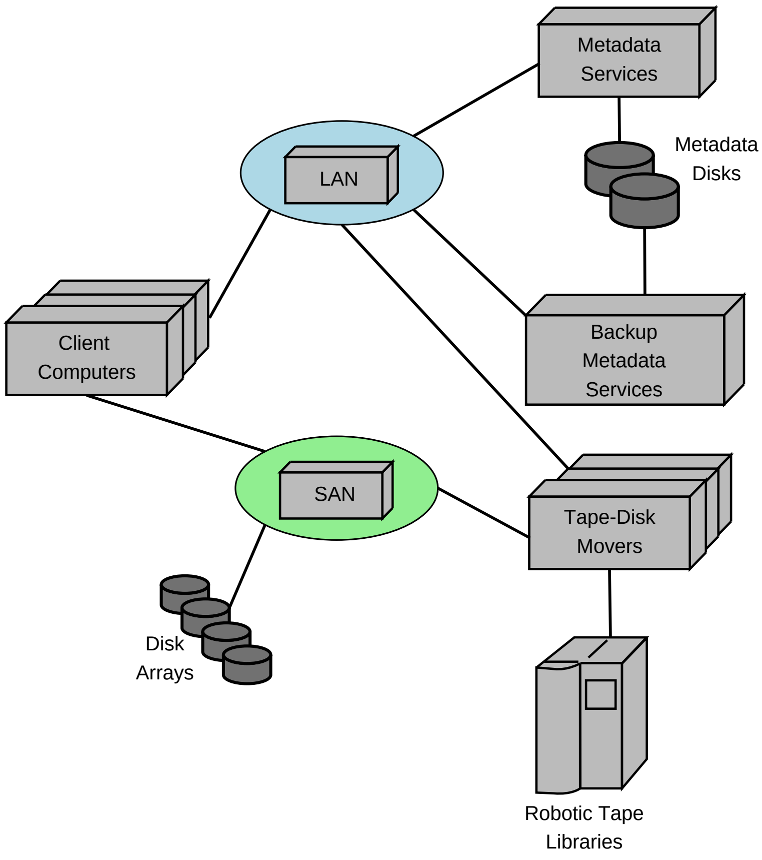

Figure 3.3: Abstracted HPSS deployment

robust metadata services are needed, which support atomic transactions, backup and recovery mechanisms, a scalable set of metadata structures, and scalable update and access algorithms.

The Tape-Disk Movers implement the main architecture to either transfer data from a source device or to redirect the input and output to Mover. For instance, to do a direct SAN transfer. A device can be memory, tape, disk, network or file system or any other physical or logical storage entity [23]. Apart of that, the Movers are also responsible to retry failed requests and to optimise the transfer.

For data handling, HPSS provides a variety of user and filesystem interfaces. Beside protocols like the Virtual File System (VFS), FTP, Samba and Network File System (NFS), its most powerful Client API is an extended Portable Operating System Interface (POSIX), called CLAPI. It provides, beside the POSIX semantics, a set of specific HPSS functions and can be used directly by end user applications or by data storage service applications. One of the advanced functionalities is for instance the stripping of disks and tapes to create parallel files. Through parallel access it is then possible to reach data rates much higher than the rate of a single disk array or tape can achieve.

In summary, HPSS is a robust and very scalable HSM solution, which provide an unified name space and the possibility to store data to disks or to robotic tape libraries. HPSS was used in 1998 to 1999 as the main mass storage system at CERN. To fit the needs of the experiments and users, CERN decided to switch to the self developed CASTOR HSM system, which is based on the SHIFT project started in the early 1990s.

Chapter 4

CERN Advanced Storage Manager

The CERN Advanced Storage Manager, CASTOR, is a hierarchical storage management [HSM] system developed at CERN for files that may be migrated between front-end disk and back-end tape storage hierarchy. Files in CASTOR are accessed using RFIO (Remote File Input/Output) protocols either at the command level or, for C programs, via function calls. The CERN Central Data Recording Service [CDR] uses CASTOR for the transfer of raw data from experimental areas to the central storage. CASTOR is continuously enhanced for the LHC experiment and, as such, has been integrated with Grid technologies.

As shown in Figure 4.1, CASTOR currently manages over 4,2 petabytes of data in over 40 million (users + production) files.

4.1 History and Limitations

CASTOR is an evolution of the SHIFT project started in the early 1990s, which allowed multiple tape, disk, and CPU servers to interact over high performance network protocols. SHIFT was based on RISC workstations and specialised networks.

In January 1999 CERN began to develop the first version of today's massive system which has thousands of Linux PC nodes linked by Gigabit Ethernet to hundreds of terabytes of automated tape storage cached by dozens of Terabytes of caches based on commodity disk components. CASTOR was then first deployed for

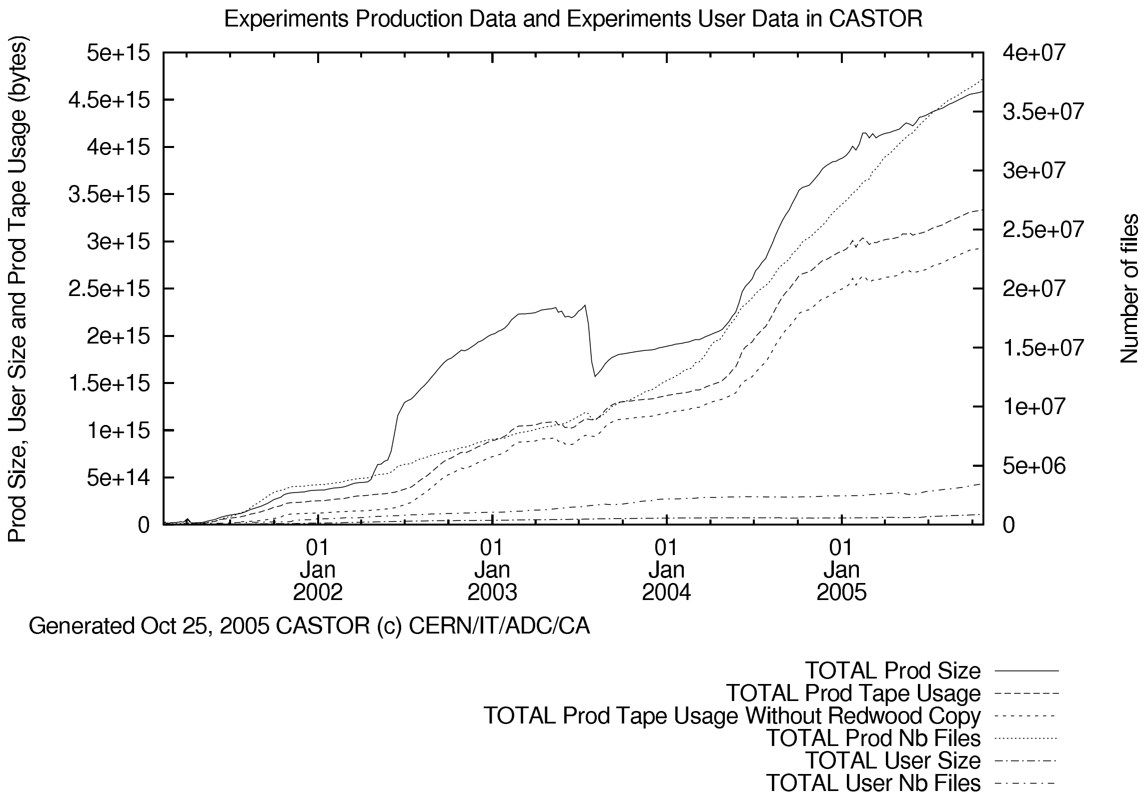

Figure 4.1: Statistical overview about CASTOR evolution

full production use in 2001.

Due to the large increase in numbers of disk servers and file systems at CERN, a main revision of the first realease was necessary in 2004. At this time, most of the components where stateful, without a database in the backround. The Stager component was the central data information handling unit of the files in the CASTOR disk pool. The growth of its catalogue caused an exeedance of the available physical memory. The temporally solution was to run up to 50 Stager instances in parallel on different machines.

Also the management flexibility became a problem, in particular its robustness and the lack of fault tolerance [24]. Other major problems were:

- scalability problems
- performance bottlenecks
- needs of proper resource sharing

- sub-optimal use of resources
- historical growth, hard to maintain code

Today, all information about the files in the disk pool are managed in a central database. It was necessary to reimplement the whole Stager architecture from scratch and to adapt some of the old components. Since then it is called CASTOR2, and is much more scalable, robust and flexible than before.

4.2 Architecture

CASTOR2 possesses a modular design with a central database for information handling.

The general interactions will be described based on the diagram in Figure 4.2. As examples will use a file put- and recall request. The grey rectangles in the diagram illustrate which components have to be installed on the same host.

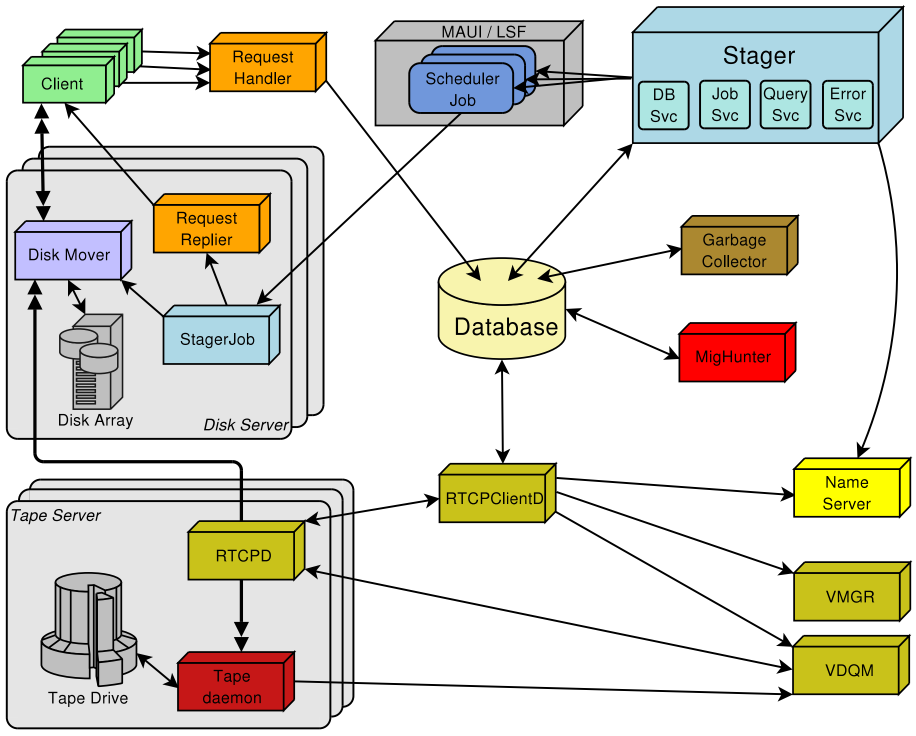

Figure 4.2: CASTOR2 architecture overview (based on the CASTOR presentation in 2005 [5])

Putting a File to CASTOR

1. The client calls through a client API the Request Handler [RH], which stores the file put request into the database. The supported client interfaces of CASTOR are GridFTP (see Section 2.3.1), the Remote File Input/Output Protocol [RFIO] and CERN's ROOT framework [25].

2. The Stager daemon looks regularly into the database and creates for each put request a basic file location entry in the name server and in the stager database file catalogue. The Stager catalogue handles only the information of the files in the disk pool.

   The Stager launches for each put request a scheduler job. This can be either MAUI [26] or LSF [27]. A scheduler is needed to manage the load balancing of the disk servers and to choose the most suitable one for the file transfer. To do this, it retrieves all important infomation about the machines from a global disk server monitoring component, such as free disk space, CPU usage, or provided file transfer protocols.

3. Once a disk server has been chosen, the scheduler creates a StagerJob process, which supervises the whole data transfer, starting from the user disk location up to the disk of the CASTOR disk pool. After having locked the requested amount of disk space on its local disk, the StagerJob contacts the Request Replier [RR] , which is responsible for forwarding the disk mover (GridFTP, RFIO, ROOT) to the client. The real transfer is then handled from the chosen file transfer protocol and has basically nothing to do with CASTOR. Once the transfer has been completed, the StagerJob process marks the job entry in the database as "finished" and adds the file location information to the name server [NameServer]. Even for the user the transfer to CASTOR seems now to be successful completed and he won't take notice from the remaining migration to the robotic tape library.

The procedure to get a file out of CASTOR is quite similar to the put request.

Getting a File out of CASTOR

A. The client calls through a client API the Request Handler [RH], which stores the file get request in the database.

B. The Stager checks whether the requested file exists in CASTOR by contacting the name server. Then, it looks into the Stager catalogue to know whether it resides already in the disk pool. If this is the case, the Stager starts a scheduler job to contact the client and to send him the file. On the other hand, if the file only exits on tape, it has to create a new entry in its catalogue and to reserve space for the file in the disk pool. The file entry is marked for the Remote Copy Client Daemon [RTCPClientD] in the Stager catalogue as ”to be retrieved”. The RTCPClientD is an individual process, which manages the file transfers between the disk pool and the robotic tape libraries.

The data transfer between the disk pool and the robotic tape libraries is an asynchronous process, which is completely uncoupled from the put or get requests of the clients. The involved CASTOR components are fully derived from the first CASTOR version or even from SHIFT. Only the Remote Copy Client Daemon has been adapted to work together with the new Stager and the database.

The Migration to the Disk Pool

4. The users, who put files into CASTOR have always to specify the service class [*SvcClass*] that should handle their request. Service classes are well known objects in the database, which provides the migration policies for a certain user group.

   Independent migration components [MigHunters] query and trigger the database for migration candidates and start queueing the file information to streams, when the amount of data waiting for migration in the disk pool is high enough. The MigHunters create as many streams as the administrator configured for the SvcClass. The migration policies of the MigHunters are specified in simple Perl scripts. The reason that it does not make sense to immediately write every single file to tape is, that the mount and dismount procedure of a tape takes too much time.

5. The Remote Tape Copy Client [RTCPClientD] checks whether streams are waiting for migration. If this is the case, it asks the Volume Manager [VMGR] which tape can for storing the data. VMGR manages a database, which holds information about the tapes, such as volume id, free available space, density and physical library location.

   RTCPClientD submits then a write access request with the required tape information to the Volume and Drive Queue Mangager [VDQM] . This component queues the tape request for a later assign to free tape drive. Therefore, it has to manage the states of all available tape drives. They are updated from the Tape daemons, which run on every single tape server.

   Once the request is eligible to be assigned, its information is sent together with the free tape drive data to the Remote Tape Copy Daemon [RTCPD] of the selected tape server. RTCPD is the CASTOR tape mover that handles the transfer of the files from the CASTOR disk pool to the tape drive. Therefore, it has to allocate physical memory (by default 128MB) for the data buffering on the local machine. This is needed, because a tape drive can store data much faster to tape than a single disk could provide them.

   For every incoming tape request it forks a child process, which contacts the RTCPClientD to receive the whole tape migration request. Now, it can start copying the files to the buffer. At the same time, RTCPClientD marks them in the database as selected to avoid their migration from other streams.

6. The child process of RTCPD handles not only the buffering of the files in the high memory. For the migration it also collaborates with the tape daemon [TapeDaemon] process on its tape server, which provides the Physical Volume Repository functionalities like mounting, positioning, unloading tapes or configuring drives. As soon as the requested tape is mounted, RTCPD starts then transferring the buffered files to the dedicated tape drive.

   At the end of a successful migration, RTCPClientD marks the files as "staged" in the database, and updates the name server information. An independent garbage collector component has now the right to clean up the disk space and to remove the disk location entry in the CASTOR database, if needed.

Recalling a File Back to Tape

C. RTCPClientD queries the Stager catalogue for files, that have been marked as ”to be recalled”. For each file recall request it asks the name server on which tape the file has been stored. Afterwards, RTCPClientD retrieves the tape information from the Volume Manager and sends a new tape request to VDQM.

D. The following steps are now quite similar to the migration process. VDQM looks into its internal list for a compatible free tape drive for the queued tape read request. As soon as a fitting tape drive is free, it sends an assign request to the dedicated RTCPD to start the mount process of the tape. As in the migration process, the RTCPD recalls all needed information about its job from the RTCPClientD and manages the whole file transfer to the reserved space on the disk pool.

   At the end of the file recall, the CASTOR catalogue is updated by RTCP-ClientD, to mark the file as ”staged” in the disk pool.

4.2.1 Database Generation and Handling

The database plays a central role in the CASTOR2 design. Its database schema is very complex and contains about sixty tables and several procedures and triggers. Different from traditional applications, CASTOR2 uses the database to store actual status information of ongoing processes, to have stateless components. Most of the table entries have only a very short lifecycle and are deleted as soon as their process is completed.

To deal with the database information in the C/C++ world, each table row can be represented by a C++ class, also called container class. These classes are pure data containers and do not implement any deeper logic.

In order to avoid having to implement this infrastructure manually, the CASTOR team created an automatic code generation facility, based on the open source software ”Umbrello UML Modeller [28]”. Umbrello is a Unified Modelling Language [UML] designing program for Linux systems, that allows you to create diagrams of software and other systems in a standard format. It uses the XML Metadata Interchange [XMI] specification to store the information of the designed objects in a human readable format.

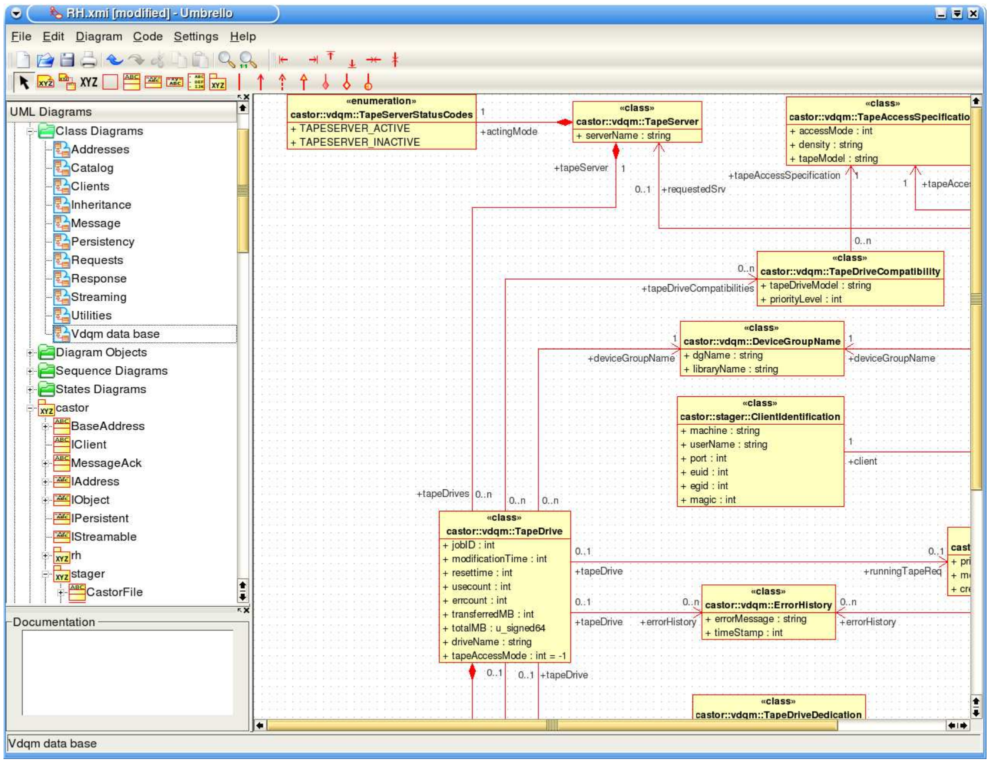

Figure 4.3: Snapshot of the Umbrello UML Modeller

All database tables for CASTOR are designed in Umbrello (see Figure 4.3). Each column of a table is specified by a parameter in the UML class. The adapted code generation programme parses afterwards the XMI file and creates for each UML class not only the C++ representation and several helper classes for retrieving process, but also two scripts to deploy and remove the all database tables. It is even able to resolve many to many relations between tables, by creating an additional join table. Supported databases are currently Oracle and MySQL.

Advantages of the CASTOR code generator are primarily time consumption and robustness. Errors in the generator code can be quicker detected and removed, since they affect nearly all generated classes.

4.2.2 The Database Interface

To include in a generic way the auto generated classes, the CASTOR team has developed a complex technique. Each required service and converter class will only be created when it is really needed. This ensures a minimum of memory load and a balanced performance.

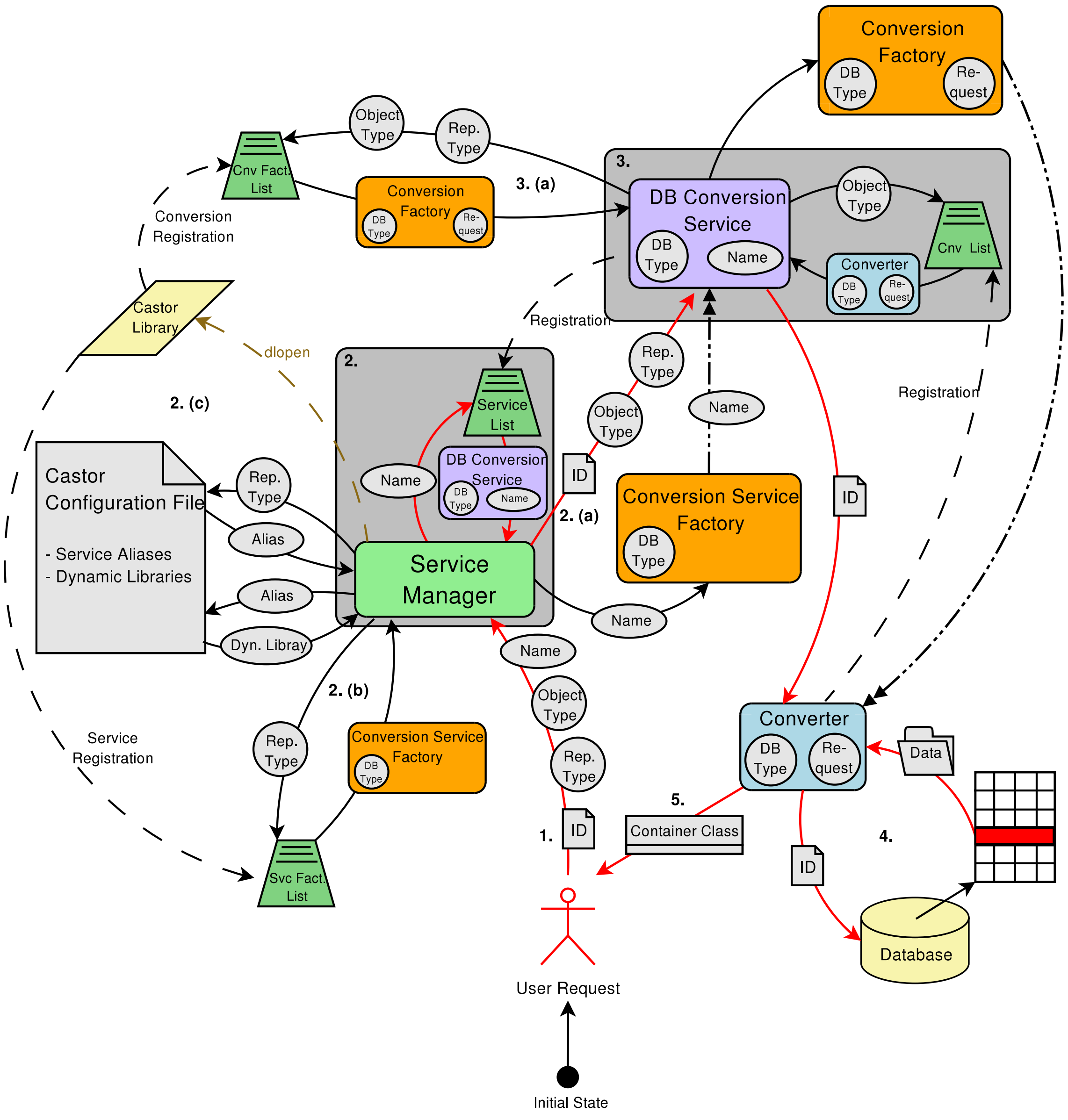

Figure 4.4: Retrieving data out of the database

The following text will describe in detail the diagram of Figure 4.4. It illustrates how the information in the database can be recalled by using the Service Manager facility.

1. To retrieve a container class with requested values out of the database, the user has to send all needed parameters to the Service Manager instance. The explicit parameters are:
   - the unique ID of the row
   - The type of the requested container object.
   - the name of the service to be used
   - the desired service type (Representation Type), in case the service does not already exist.
2. The service manager is responsible for the service objects and their factories. A user can ask for several service categories, such as the streaming service or like in this case the DB Conversion Service. The service manager handles the dynamic load of service factories and the instantiation of services, that are not yet registered in the Service List. The service objects can be selected with the given service name [Name].
   (a) If the service is already loaded and registered, the service manager gets it from the Service List and calls it with the ID, Object Type and Representation Type information.
   (b) If the service is not registered, the Service Manager looks into the Service Factory List for the right factory to create the desired service instance. For retrieving data out of the database, a Conversion Service Factory is used to create an instance of a DB Conversion Service. The Service Manager handles afterwards the registration in the Service List under the desired service name. Now, the DB Conversion Service can be executed as explained in (a).
   (c) If the needed Service Factory not listed, the Service Manager starts a final effort by looking in the CASTOR configuration file. The CASTOR configuration file is the main entry point for all CASTOR related configurations. Among other things, a dynamic library name can be found

associated to a service type. The CASTOR configuration file may also contain aliases, mapping services types to others. By using the Linux system call *dlopen* the needed CASTOR library can then be reloaded, to fill up the Service Factory List and the Conversion Factory List with missing objects (Please see Section 4.2.1). Afterwards, the Service Manager retries the failed lookup of (b).

3. Once the DB Conversion Service is called, it looks into its Conversion List to find the right converter for the requested Object Type. The converters manage the query to the database for their dedicated table. The converters are auto-generated by the code generator.

   (a) If the converter does not already exist in memory, the DB Conversion Service has to select the right Conversion Factory for the user request out of the Conversion Factory List. There, the Conversion Factory should normally be present for the specific Object Type and DB Type, since the list is filled with the objects of the dynamically loaded CASTOR Library. After having generated the right Converter, it is registered in the Conversion List for a later re-use.

4. The specific table Converter sends a select statement to the database to retrieve the information, stored in the row with the specified unique ID. If the row exist, the Converter puts the values into a designated Container Class.

5. Finally, the user receives a container class with the values, which he can now use for further processing.

To store information into the database, the user has to use the Service Manager. In this case he has to forward a filled container class, instead of the Object Type. The Object Type is not needed, because every container class provides a static constants, which determines its type. Thus, it is possible for the DB Conversion Service to select the right Conversion Factory.

4.3 The Distributed Logging Facility

Due to modular component design of CASTOR2 problems can be very complex. Thus, it is mandatory to have a powerful logging facility, which allows advanced message filtering.

CERN decided to implement an individual logging solution for CASTOR, which is called Distributed Logging Facility [DLF]. The purpose of DLF is to streamline and centralise the logging and accounting in CASTOR. The facility consists of a DLF server with an Oracle or MySQL backend and a client API for writing logging records. The standard Universal Logging Message [ULM] was selected for the log record format.

For each CASTOR component it is specified in the configuration file whether the log message are sent to the DLF server or stored in a local file.

The central message administration allows us to implement an independent web based user interface, which allows users to filter the messages by CASTOR specific parameters. It is for instance, possible to select all messages related to one file ID, or those which concerns a special tape.

A message can be sent with with ten different severity levels. The most used levels in CASTOR2 are listed below:

- DEBUG: For debug log messages
- USAGE: For tracing routine calls
- SYSTEM: The normal service message
- WARNING: Used for self-monitoring warning messages
- ERROR: Used, for all errors during normal operation

Chapter 5

Technical Analysis

This Chapter will explain the expectations of a reimplementation of the CASTOR Robotic Tape Queueing System (VDQM). Furthermore it will give a detailed technical analysis of the existing tape queueing system and about the main problems which have to be solved.

5.1 Requirements

The intention to create a complete redesign of VDQM has several reasons. One is that the old implementation is very difficult to understand, because of an undirected growth of C code and the use of macros. Of course, this alone doesn't justify a reimplementation. The main argument is, that the old design has two major problems:

- The old VDQM daemon stores the data in memory, which makes it stateful. This means a lost of all information at a system crash.
- Due to the increasing amount of tape requests the old system has a high memory load and can even cause an overload. This could be fixed by restricting the amount of acceptable requests, but then VDQM would become a bottleneck for the whole CASTOR system.

On the basis of these facts, a reimplementation of the full robotic tape queueing system has to achieve the following requirements:

- To be backward compatible, which means that communication with the other daemons and all administrative commands from the VDQM Client API still work with the new implementation
- To be stateless, so that no information is lost in case of an unexpected crash
- The design should be as robust as possible, which means that all unexpected external errors, like network problems, are handled and do not crash the system. Beside that, the daemon should not hang or slow down if an incoming request causes problems
- To be able to deal also with a very large number of tape requests.
- To use the existing DLF logging facility
- To solve the problem of the asymmetric tape drives
- To support tape drive dedication to specific users and/or tapes
- To be easily extendable, e.g. for new protocols

The following chapters will explain in detail these requirement and the possibilities to solve them.

5.2 The Communication Protocols

After having described the technical requirements, this Section will acquire a proper analysis of the communication protocols, used between the clients and VDQM. Because of missing protocol documentations, an analysis of the existing implementation was necessary.

5.2.1 The Request Accept Protocol

A major point is to still support the existing communication protocols of the old RTCPClientD, RTCPD and tape daemon components, as shown in Figure 4.2.

The VDQM code analysis shows, that the communication of VDQM with the tape daemon and RTCPClientD is based on the same protocol. VDQM listens to a predefined network port and handles each accepted incoming request in its own thread. A VDQM message consists of two main parts, a header and a body. The header includes three values:

- a unique so-called ”magic number” to identify the used protocol
- the VDQM message type constants
- the length of the body message

Depending on the incoming message, the body can include the volume request of the RTCPClientD or tape drive information sent by the Tape daemon. The following information is sent with the request:

- Tape request (sent by RTCPClientD):
  - The client information, such as port number, user id, group id and name
  - The requested volume id [vid]
  - The tape access mode: write- or read access
  - The device group name of the volume
  - A field, to specify a specific tape server, if requested by the client.
  - A number for the priority of the request
- Tape drive request (sent by the tape daemons):
  - The status of the tape drive
  - The name of the tape drive
  - The name of the tape server where the tape drive is installed
  - The device group name of the tape drive
  - The job id, in case of a tape assignment
  - The transferred megabytes. This is only sent with the *release* message.
  - A pattern matching string, to send usage restrictions for the specified tape drive. As we will explain in Section 5.5, this string is not anymore used.

The device group name is needed in both cases to realise the assign of tapes to compatible drives.

VDQM stores the incoming request in two lists in its memory:

- Tape request list
- Tape drive list

The request information is stored in C structs, which are concatenated as a linked list. To reach a stateless process, these two lists have to be managed in a database. Figure 5.1 shows the communication steps for incoming request.

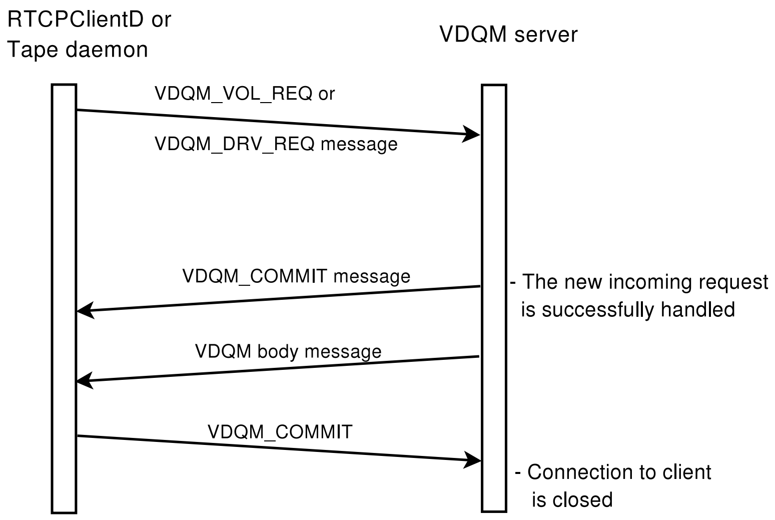

Figure 5.1: Communication protocol for incoming requests

After having verified the magic number, VDQM decides how it proceeds with the message body, given the specified message type in the header. VDQM distinguishes between 18 different message types, including administrative ones.

Every successfully handled message closes with a simple handshake. Therefore, VDQM sends to the client a commit message and then a message body that includes the latest status information of the tape request or tape drive. The client terminates the connection by sending a commit receipt for the last message.

If VDQM can not interpretate the client's message, or if an error occurs during processing, it sends a header back to the client with an error constant in place of the message type. As shown in Figure 5.2, VDQM then does an internal rollback of all made changes and closes the connection.

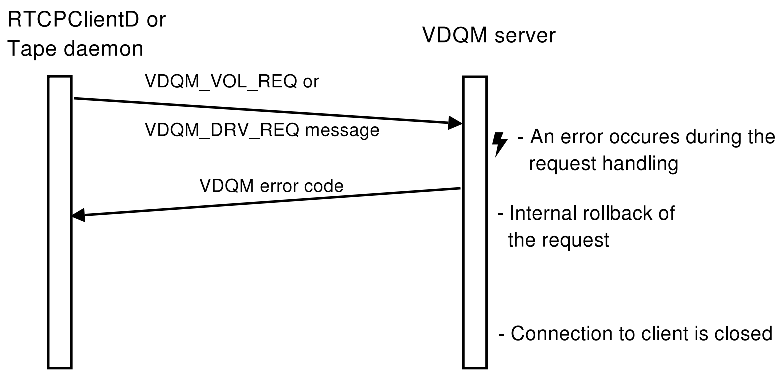

Figure 5.2: Error handling behaviour in the incoming request protocol

Only some of the 18 messages are still relevant for the reimplementation. A lot of them were for administrative remote commands, which for historical reasons are no longer in use or would not make sense in a reimplementation. For instance it is not necessary to provide in the new system a shutdown function. A stateless process can be killed at any time from the shell. Table 5.1 gives an overview of the relevant message types.

5.2.2 The Tape to Tape Drive Assign Protocol

When VDQM has a free tape drive in its list, it looks for the next suitable tape in the tape request queue. If it finds a matching couple, it sends all requested assign information to RTCPD. The assign candidate has to be in the same device group as the tape drive. This restriction is needed to avoid VDQM attempts to dedicate an incompatible tape request.

Beside the magic number and the correct message type, RTCPD needs to know the client information, the tape drive name, the robotic library and the tape request id. Given this information it is then able to contact the RTCPClientD process for retrieving all details about the tape request.

Figure 5.3 illustrates the communication protocol. Different from the described protocol between VDQM and RTCPClientD or the tape daemon, this protocol does not include a handshake phase. If an error occurs during the tape assignment, RTCPD sends an additionally error message in its response.

VDQM message type	**User**	**Message description**
VDQM_DRV_REQ	tape daemon	To send the actual tape drive status of one of their dedicated drives
VDQM_DRV_REQ	RTCPD	To Send the amount of transferred data to VDQM
VDQM_VOL_REQ	RTCPClientD	To add a new tape request to the queue
VDQM_DEL_VOLREQ	RTCPClientD, Administrator	To delete a tape request from the queue
VDQM_PING	RTCPClientD, Administrator	To return the queue position for a particular tape request or just to check whether VDQM is still alive
VDQM_GET_VOLQUEUE, VDQM_GET_DRVQUEUE	Administrator	Is used in relation to the administrator command "showqueues", which returns information about tape request queue and tape drive queue
VDQM_DEL_DRVREQ	Administrator	To delete a specific tape drive from the VDQM tape drive list
VDQM_CLIENTINFO	VDQM	VDQM uses this message type to send information for a tape to a tape drive assignment to RTCPD
VDQM_COMMIT	ALL	Used in the handshake phase, to inform the receiver about the success of the request

Table 5.1: The relevant messages, used in the VDQM protocol

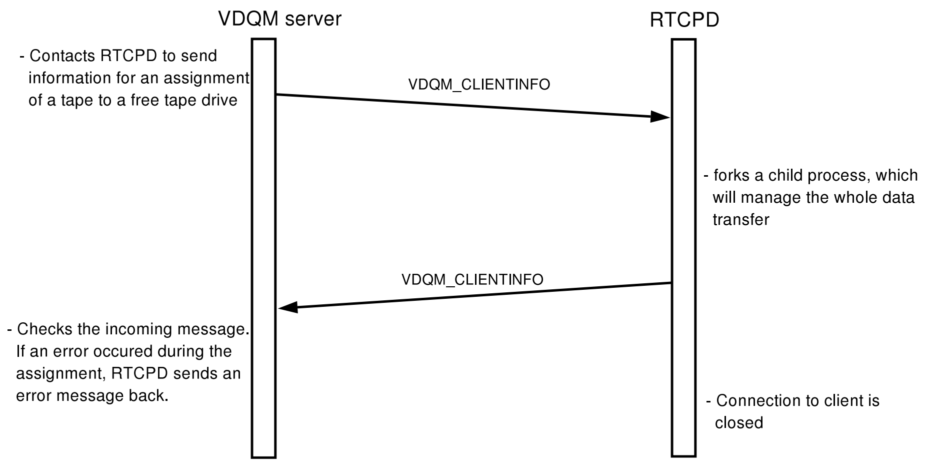
VDQM server
RTCPD
- Contacts RTCPD to send information for an assignment of a tape to a free tape drive
VDQM_CLIENTINFO
- forks a child process, which will manage the whole data transfer
VDQM_CLIENTINFO
- Checks the incoming message. If an error occured during the assignment, RTCPD sends an error message back.
- Connection to client is closed

Figure 5.3: Tape to tape drive assignment protocol

5.3 State Analysis of the Volume and Drive Queue Manager

One of the main requirements of VDQM is the administration of the different states of the tape drives. In the old VDQM system, the status information is a combination of bit flags, stored in a single integer variable. Each status has an assigned bit, that can be set to 0 for off, or to 1 for on. This method makes it very difficult to extract valid bit combinations, to understand and to debug the existing code.

From the technical analysis of the code arises the knowledge that there are only a few valid tape drive states and some temporary states, which are used for the internal algorithm logic. The resulting state diagram for tape drive status lifecycle is shown in Figure 5.4. The states in round brackets stand for temporary states, which are not stored or sent to the client. The arrow identifiers specify the received tape drive status information of the tape daemon.

The following paragraph will explain the standard lifecycle of Figure 5.4, without going into the details of the VDQM error handling:

1. To assign a tape to a tape drive it is mandatory that only the status flags UP and FREE are set. If RTCPD sends a positive reply for a tape assignment back to VDQM the status is changed to UP and BUSY. In addition, the tape drive is internally connected to the tape Request, to avoid a second assignment of the same tape request to another tape drive.

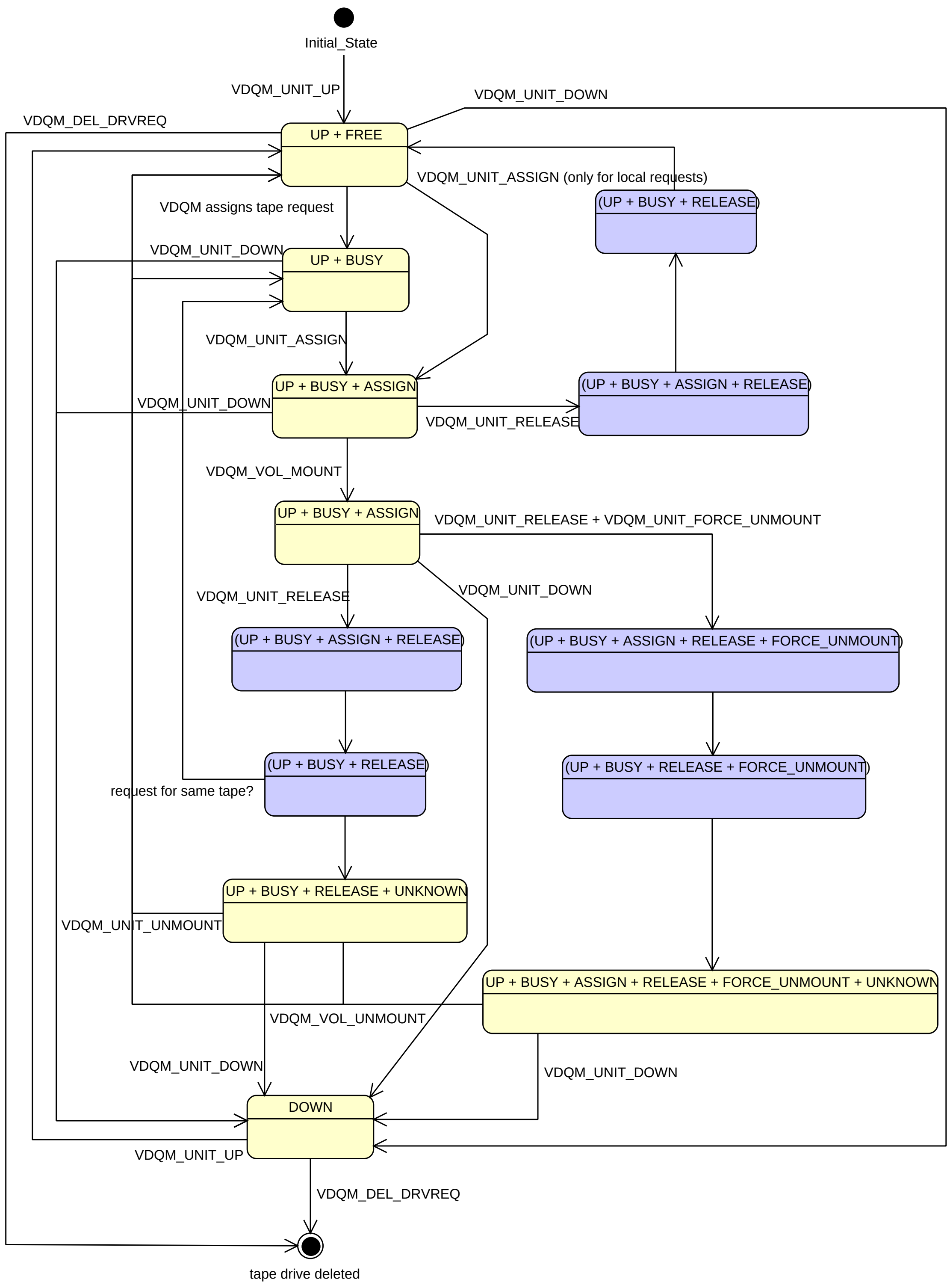

Figure 5.4: The different states for a tape drive in the old VDQM system

2. From now on, VDQM receives all changes of the tape drive status from the responsible tape daemon process, which has been informed by RTCPD. Each data transfer request receives a job ID, which is provided by the tape daemon. When the job ID is forwarded to VDQM, the tape drive information is updated and an additional ASSIGN status flag is set. The tape now resides in the tape drive, but is not yet mounted.

   An administrator can also avoid the automatic assignment of VDQM and manually assign a transfer job to a tape drive. In this case, VDQM goes directly to this state.

3. The next tape drive message is usually the information, that the tape has been mounted and is wound to the right position. From that moment RTCPD starts the data transfer. The tape daemon sends the volume ID of the mounted tape, which should correspond to the volume ID of the tape request. The volume ID is a unique ID to identify the tape. VDQM does not change the status flags for the incoming mount message, but adds the volume ID to the tape drive information.

4. After RTCPD has finished the data transfer, it orders the tape daemon to rewind and to release the tape. Secondly, VDQM is informed about the amount of transferred data. The according status message is then forwarded by the tape daemon to VDQM, which looks into its tape request queue, whether there exists, for the same tape, another request. If so, VDQM sends immediately a new assign request to RTCPD and switches the tape drive status flags to Up and BUSY. Otherwise it waits for the unmount information with the status flags UP, BUSY, RELEASE and UNKNOWN.

   In case of errors during the transfer, the tape daemon can force at any time an unmount of the tape. VDQM sets the tape drive immediately into a waiting status for the unmount message, without checking the tape request queue.

5. As soon as the volume has been put back into the robotic tape library and the tape drive is idle the tape daemon sends an unmount message to VDQM. Thereupon, it resets all information about the last request on that tape drive

and switches the status back to UP and FREE. The tape drive is now ready to accept a new tape request.

VDQM is a passive system and tries always to realise the incoming status messages of the tape daemons, which are of course always right. If an incoming status is not the expected one for the specific tape drive, VDQM tries to handle it and would, in the worst case, set the status of the tape drive to UNKNOWN, until the tape daemon sends the actual and valid tape drive status.

In the reimplementation, the bit combination of the status flags are replaced by seven clearly defined status constants. Figure 5.5 shows the revised state diagram. As we can see, the reimplementation abandons the use of intermediate states, which makes the code much easier to understand. The only status code, which does not appear in the diagram, is STATUS_UNKNOWN. It is used in case the tape drive status cannot be exactly determined.

5.4 Design Decisions for a Stateless Application

As mentioned in Section 5.2.1, the best way to reach a stateless application is to use a database to store the request messages. The same strategy was realised with good results in the new stager implementation. Another point, which has a lot to commend it is the reuse of the CASTOR code generation facility (see Section 4.2.1). Moreover, it is then possible to implement later on helpful administrative web services.

In the old VDQM implementation, the dedication of a tape request to a free tape drive was based on C functions, with no possibility to change the tape request dequeuing algorithm on the fly. Due to the use of a database it is possible to write a query statement, which can be changed on a running system.

Every row in the database possess a unique 64 bit identification [id] number. At the reimplementation has to be attended, that the id has to be downcasted to a 32 bit value to fit the old protocol. Because of the short lifecycle of a tape request, it will never happen that two 64 bit id numbers have the same 32 bit downcast value. From this it follows that tape requests can always be exactly determined.

The downcast makes no problem for the tape drives list, because a drive can always be clearly identified by its name and its tape server. These values are sent with every tape drive status request, as it was stated in Section 5.2.1.

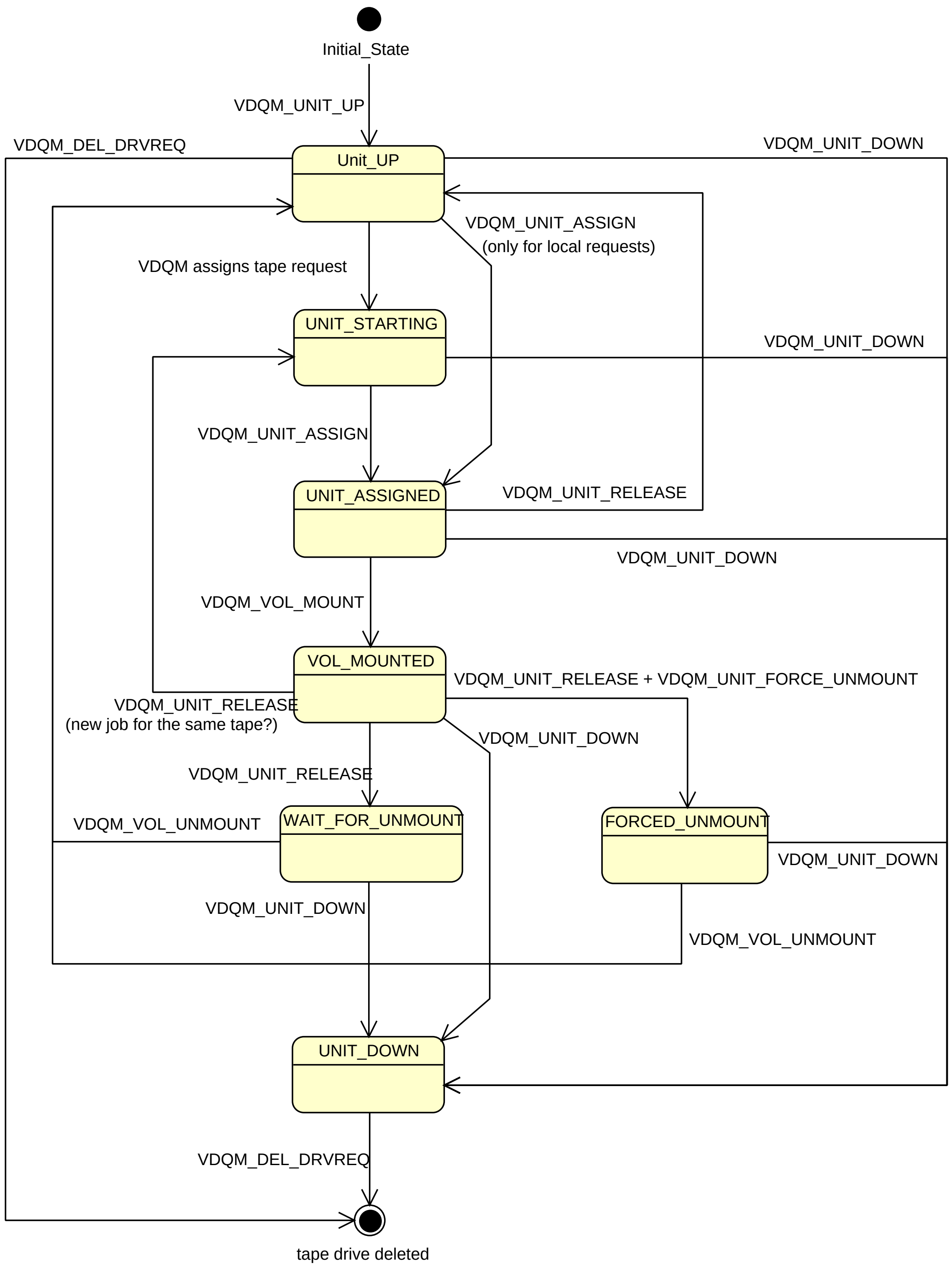

Figure 5.5: Revised tape drive states for the new VDQM implementation

5.5 Tape Drive Dedication

The old VDQM implementation was already able to reserve a tape drive for specific tasks. Therefore, the administrator had to specify a pattern matching string with all the restrictions for a specified tape drive. VDQM then dedicated the drive to clients matching this dedication expression. The disadvantages of this solution are:

- A special parser has to read the string and to check its correctness.
- By the protocol, the length of the pattern matching string is limited to 1024 character array.
- The requests are difficult to display in an easily readable format for the administrators.

The reimplementation solves this problem using an extra table in the database. This is then parsed by the query that dedicates the tape request to a tape drive. To enter new rows into the table a new Perl script has been implemented.

5.6 The Asymmetric Tape Drive Problem

In a robotic tape library resides a lot of different tape and tape drive generations. New tape drives often use new tape versions with higher densities, but with the same cartridge size as their predecessor models. Older tapes can some times only be read by the new tape drives, but not be written to. This is called ”the asymmetric tape drive problem”.

At the moment, VDQM is not able to dedicate an older tape to an asymmetric tape drive, which could still read the data. The problem has been ignored, due to its complexity.

To solve it with a minimal amount of changes in the old components, the tape daemons have to send for the registration of their tape drives supplementary the device model. This is necessary, because all tape models have to be put together into the same device group as their compatible tape drives. The Remote Tape Copy Daemons [RTCPD] don’t assign tapes of a different device groups. Such a change signifies that newer and older drive devices would belong to the same device group. Then it is not possible to distinguish them via the device group name.

Furthermore, VDQM needs information about the tape models, tape densities, and the library locations in its database. As mentioned in the migration example of Section 4.2, VMGR administrates all information about the tapes in the libraries. Hence, an independent small application is able to initialise the VDQM database with these static tape specifications.

The effort has to be done to make for each tape drive model an access priority list the of its compatible tape models. Thus, the tape request dedication algorithm (see also Section 5.2.2) can consequently prioritise new tape models on their proper devices. This avoids a jam in the request queue for newer tapes.

The asymmetric tapes have to be filled in manually in the priority list because it is impossible to get these information from other CASTOR2 components.

Figure 5.6 shows the extended communication protocol for an incoming new tape request. The volume id, which is sent with every tape request enables us to retrieve the missing tape information, such as volume density and tape model, from the volume manager. They are needed to associate the new queue entry with the static information in the database.

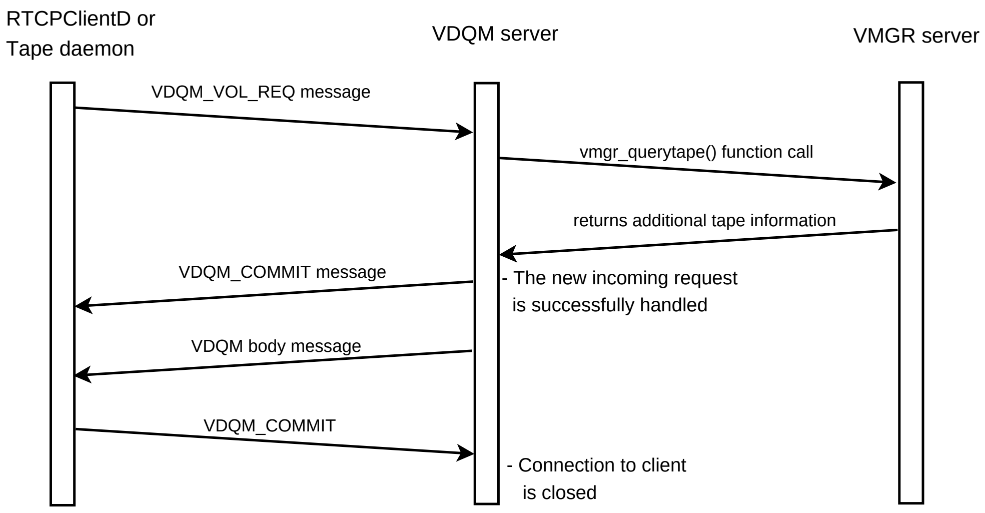

Figure 5.6: Extended communication protocol for incoming tape requests

A detailed description of the concrete realisation will be given in Chapter 7.

Chapter 6

Preliminary Design

This Chapter will present the use cases of the clients, which sends requests to VDQM. But first of all, the new interactions of VDQM will be described in Section 6.1.

6.1 Scenario

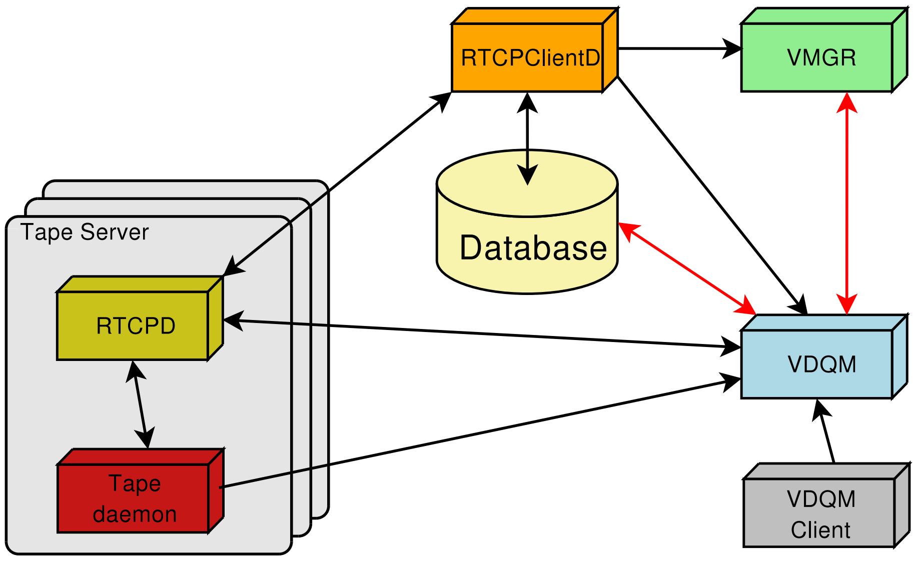

Figure 6.1: Interactions of the new VDQM implementation with other CASTOR2 components

The reimplementation of VDQM uses the database to store all incoming information from the clients and to handle the assignment of tape requests to tape drives.

A lot of database queries are also needed to manage the asymmetric tape drive problem. Therefore, it is also required that VDQM recalls additional information from the Volume Manager. Figure 6.1 shows the new scenario of VDQM with the other components of CASTOR2. The new relations are marked with red arrows.

The *VDQM Client* in the diagram represents the administrator commands, which will be explained in more detailed in the next Section.

6.2 User Interfaces

This Section will define an exact use case description, and provide both functional- and non-functional requirements. An exact requirement description is necessary for designing and implementing an adequate application.

6.2.1 Administrator and Remote Tape Copy Client Commands

The administrator of VDQM has the ability to retrieve information from VDQM through two shell commands, called *vdqm_admin* and *showqueues*. The use case diagram of Figure 6.2 shows, which commands are still supported for the administrators and the RTCPClientD by the VDQM reimplementation.

Retrieving the Actual Queue Position of a Tape Request

Both administrator and RTCPClientD send a command to retrieve the actual queue position of a specified tape request. The administrator can use the VDQM_PING message also to check if the VDQM server is still responding. Hence, it is mandatory that the command gives a quick reply.

Appending a New Tape Request to the Tape Request Queue

The VDQM_VOL_REQ message is used to queue new tape requests in VDQM. To do so, the sent information have to be mapped to the new structure of the database.

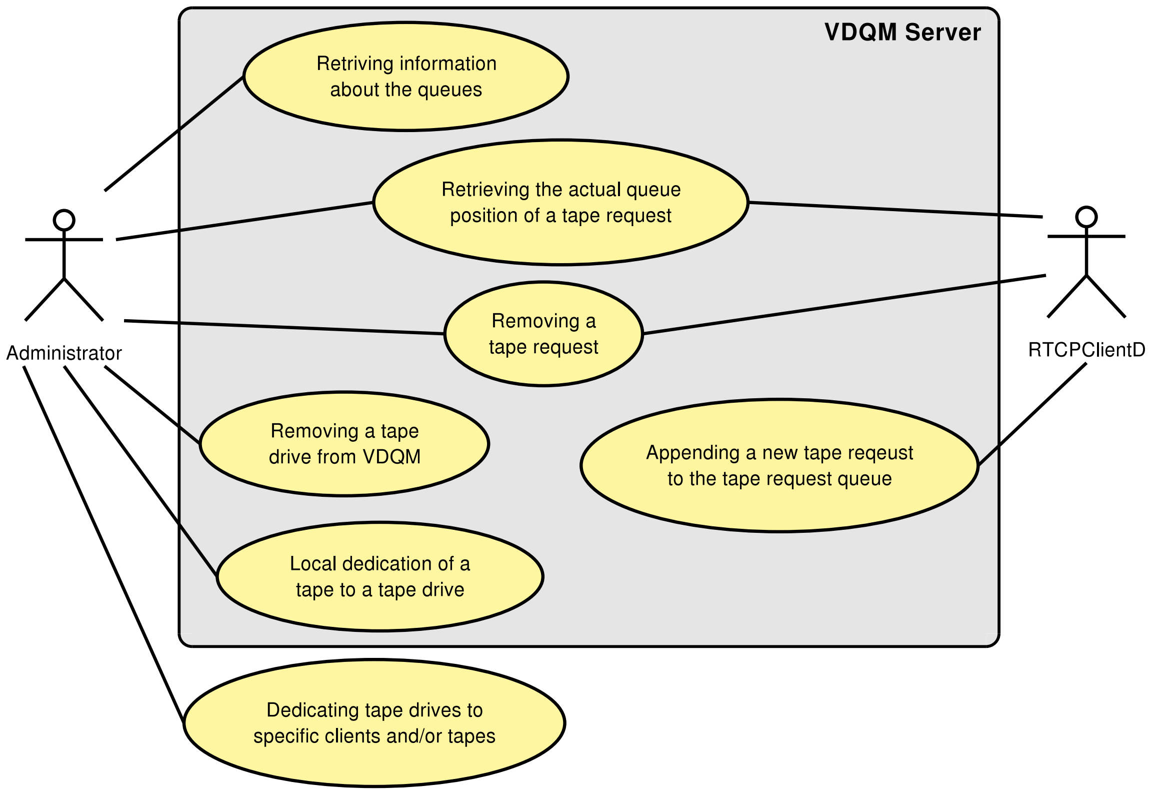
VDQM Server
Retriving information about the queues
Retrieving the actual queue position of a tape request
Removing a tape request
Removing a tape drive from VDQM
Appending a new tape reqeust to the tape request queue
Local dedication of a tape to a tape drive
Dedicating tape drives to specific clients and/or tapes
Administrator
RTCPClientD

Figure 6.2: Use case of the administrator and the Remote Tape Copy Client Daemon [RTCPClientD]

Removing a Tape Request from VDQM

Normally the deletion of a tape request is not difficult. Due to the fact that the protocol is restricted to 32 bit identification numbers, the new VDQM application has to downcast for the old clients the new 64 bit id representation and, of course, it has to take care that it is resolved later in a proper way.

Local Dedication of a Tape to a Tape Drive

An administrator has the ability to dedicate a tape request directly to a tape drive. In this case a free tape drive has directly to be switched from the UNIT_FREE status to UNIT_ASSIGNED, as illustrated in Figure 5.2. The passing of the normal tape request dedication logic is only accepted, if the command is sent locally from the tape server machine. The implementation of this command is a simple additional check in the request handling of the tape drive status message. Hence, it will not be explained further.

Removal of a Tape Drive from VDQM

When VDQM accepts the VDQM_DEL_DRVREQ message it deletes the tape drive information in the database. Therefore, the administrator has to send the drive name and the server name. A basic requirement of CASTOR is that a tape drive can clearly be specified with this information.

Dedicating Tape Drives to Specific Clients and/or Tapes

As already mentioned in Section 5.5, the dedication of tape drive is no longer solved inside the VDQM code, but with an additional Perl script, which directly accesses the database.

Retrieving Information about the Queues

The user can acquire information about the status of the drives and the tape request with the "*showqueues*" shell command of the VDQM Client.

Therefore, the complete tape request queue has to be sent to the client when VDQM receives a VDQM_GET_VOLQUEUE request message. A similar event has to be done for the tape drive information when the VDQM_GET_DRVQUEUE request message has been received.

6.2.2 Tape Daemon Commands

Figure 6.3 shows the use case of the tape daemon.

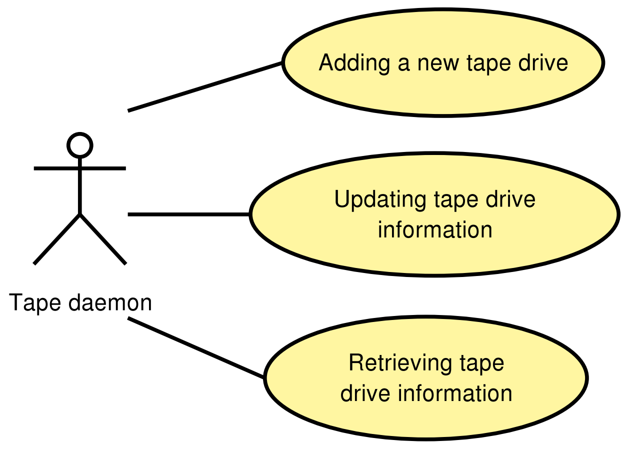
Adding a new tape drive
Updating tape drive information
Tape daemon
Retrieving tape drive information

Figure 6.3: Use case of the tape daemon

Adding a New Tape Drive

To register a new tape drive to VDQM, the tape daemon has to send a VDQM_DRV_REQ status message. The initial status of a drive has to be either Up or Down.

Updating Rape Drive Information

When the status of a tape drive changes, its responsible tape daemon has to inform VDQM with a VDQM_DRV_REQ status message about it. Which internal status bit message flags have to be sent to do the appropriate status update has been discussed in Section 5.3.

Retrieving Tape Drive Information

Due to an internal status bit flag, which has to be sent with an VDQM_DRV_REQ status message, it is possible to recall the stored information from the specified tape drive. For this reason, but not exclusively, the reimplementation of VDQM has to have a translator function, which maps the new status codes back to the old ones.

6.2.3 Remote Tape Copy Daemon Command

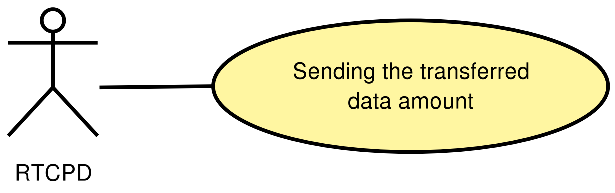

Figure 6.4: Use case of the Remote Tape Copy Daemon

During the whole life cycle of a tape request the Remote Tape Copy Daemon [RTCPD] has only in two interactions with VDQM:

1. When VDQM wants to assign a tape to a tape drive
2. When RTCPD sends the information about the amount of transferred data to VDQM.

VDQM uses this information to update the counter of the total transferred megabytes in the database for the concerned tape drive. As the tape daemon, RTCPD uses the VDQM_DRV_REQ message to forward the data amount. Even though this message has no effect on the status of the tape drive in VDQM.

Chapter 7

Detailed Design

After the basic structure was defined in the previous Chapter, this Chapter will describe the structure that is to be implemented subsequently. A low-level design will be outlined, and design decisions that were made will be discussed.

As it is required to support the old protocol, parts of the application were derived from the existing VDQM. Nevertheless, the new volume and drive queue manager was designed completely from scratch, but the old code structure was used as guideline.

Sequence diagrams are going to be used to illustrate the interactions between the several classes. Their time flow is always top down.

A complete class diagram of the new architecture is given in Figure A.1, which can be found in the Appendix to this report. It should serve to give a better understanding.

7.1 Information Handling in the Database

The database diagram in Figure 7.1 gives an overview about the tables which are needed to store the relevant information for VDQM. Due to the code generation with the Umbrello UML Modeller (see Section 4.2.1), the database is designed with the syntax of an UML class diagram. The table names correspond to the names of the classes without their namespaces. In this schema, *enumeration* classes are only relevant for the C++ code generation and represent in the database point of view just the valid numbers of one column. A diamond association, such as among the

TapeServer- and the *TapeDrive* table, express that the connected rows have to be removed together with the row of the table with the diamond.

The column names and column types of a table can be derived from the appropriate variable names and the belonging associations. An association column contains the unique row identification number [id], which is stored in a 64 bit *unsigned integer* variable. The column for the unique id is automatically added to the table and does not have to be explicitly inserted into the schema. An example for the table columns and their types is given in Table 7.1.

Table name	**Column name**	**Column type**
ErrorHistory	id	64 bit unsigned integer
	errorMessage	string
	timeStamp	64 bit unsigned integer
	tape	64 bit unsigned integer
	tapeDrive	64 bit unsigned integer

Table 7.1: Example: The columns of the ErrorHistory table

All time values are stored in seconds, which we count up from the first January 1970.

The *TapeAccessSpecification*- and the *DeviceGroupName* table only contain static information, which are filled by an independent small application, called *vdqmDBInit*. Therefore, it calls the Volume Manager [VMGR], which administrates the tape information, the list of device group names and library names. These additional information are needed to handle the asymmetric tape drive problem (see Section 5.6).

At the moment, the *ErrorHistory* table is not in use. The plan is, to extend the tape daemon protocol, so that it is possible to send an error message to VDQM, in case of problems during a tape request handling. This will enable to do analysis of the failing behaviour of specific tapes and drives.

For a better comprehension of the meaning, the tables will be explained in reference to the main tasks.

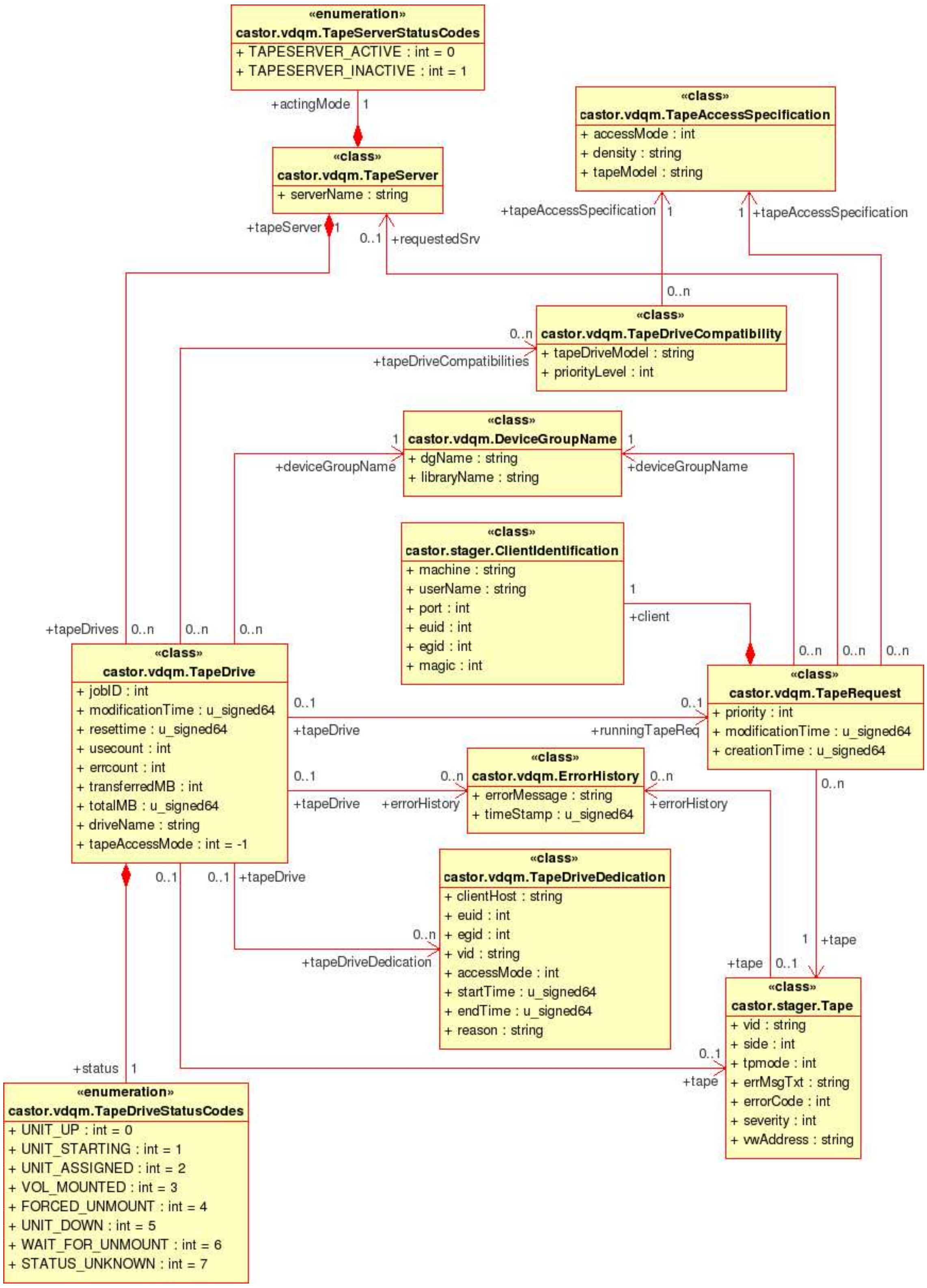

Figure 7.1: VDQM database overview

7.1.1 Storing or Updating of a Tape Drive's Information

When a tape daemon is starting, it connects to VDQM and sends a "status down" message for its tape drives. This is done to register the drives and to reset all temporary information in VDQM.

If a status message for a new tape drive arrives, VDQM adds a new row to the *TapeDrive* table. The sent information has been discussed in Section 5.2.1.

An initial row contains the drive name, the status value and the association to its tape server. If the data entry is missing then a new row is added to the *TapeServer* table. The *actingMode* of a tape server is by default set to active. This status flag is only important for the tape request dedication algorithm.

Right from the start, each *TapeDrive* row is also connected to the appropriate device group name in the static *DeviceGroupName* table. Thus, it is possible to ascertain the name of the robotic library of the tape drive.

Finally it has to be checked, if there are already existing entries in the *TapeDrive-Compatibility* table for the model of the new tape drive. This table contains the priority lists of the different drive models for their compatible tape models, that are specified in the static *TapeAccessSpecification* table. If the tape drive model name has not been sent with the request, then VDQM uses instead the supported cartridge model name. The information can be retrieved from the Volume Manager with help of the specified device group name. Unfortunately, this case implicates that asymmetric tape drives cannot be supported, because there is no possibility to distinguish the different drive models.

Due to the fact that it is impossible to obtain from the other components a list of compatible tape models for a tape drive, VDQM is only feasible to associate tape models of the same device group. Additional entries for the asymmetric tape access have to be inserted manually. The write access to tape is by default privileged to the read access. This guarantees a better performance during the data backup of the experiments. The highest priority number is zero.

Figure 5.5 shows the possible tape drive states of the new VDQM design. Table 7.2 outlines for each of these states the changes in the database during a normal lifecycle. The whole procedure has been showing in Section 5.3.

The table does not mention explicitly, that the modification time fields of the involved tables are updated every single time. It is also take for granted that the *status* field is set to the actual status of the tape drive.

Tape drive status	**Table name**	**Changes**
UNIT_UP, UNIT_DOWN	*TapeDrive*	Reset of the following fields: *jobID*, *tapeAccessMode*, *runningTapeReq*, *tape*
	TapeRequest	Deletion of the old tape request and its *ClientIdentification* entry, if necessary
UNIT_STARTING	*TapeDrive*	Connection to the assigned tape request row
	TapeRequest	Connection to the assigned tape drive row
UNIT_ASSIGNED	*TapeDrive*	Storing of the received job id
VOL_MOUNTED	TapeDrive	connection to the tape specification in the *Tape* table
WAIT_FOR_UNMOUNT, FORCED_UNMOUNT	TapeDrive	Update of the *transferredMB* and *totalMB* fields. Reset of the following fields: *jobID*, *tapeAccessMode*, *runningTapeReq*, *tape*
	TapeRequest	Deletion of the old tape request and its *ClientIdentification* entry
STATUS_UNKNOWN	TapeDrive	No changes, except of the status

Table 7.2: The changes in the database for each tape drive status

7.1.2 Storing of a Tape Request

With every new tape request, RTCPClientD sends via the request accept protocol (see Section 5.2.1) the required information (see Section 5.2.1) to handle the assignment of a tape to a compatible tape drive. For this purpose, the basic request information is stored in new rows in the *TapeRequest* table and in the *ClientIdentification* table. If the client wishes that its tape request is handled by a specific tape server the new *TapeRequest* row is additionally connected to the appropriate row in the *TapeServer* table.

The information about the volume identification number [volume id] and the tape access mode is used to connect to the right entry in the existing *Tape* table, which is provided by the stager catalogue. A new entry is created by specifying only this two values, in case that the table do not comprise these information.

It is also mandatory to connect the new tape request with a static entry of the *TapeAccessSpecification* table. This necessitates the retrieval of the tape model and density from the Volume Manager. The *TapeAccessSpecification* table includes for every tape model and its appropriate density a separate entry for the read- and write access mode. That way, it is possible to associate to the corresponding access specification of the tape request.

Like the tape drives, every tape request is also connected to the static *Device-GroupName* table.

In case of an assign to a tape drive, the tape request is additionally concatenated with the right row of the *TapeDrive* table.

7.1.3 Tape Drive Dedication

The *TapeDriveDedication* table is used to reserve time windows for specific actions on a tape drive. The dedication of the tape drives are checked with the tape request to tape drive assign procedure, which will be described in the next Section.

Additional rows can be added with a small command line script. It is mandatory that the administrator therefore specifies the drive, the time frame and information about the client, such as user id, group id and client host. If it is required, a tape and its access mode could be indicated as well.

7.1.4 Assignment of Tape Requests to Tape Drives

A first outline of this problem has been given in Section 5.2.2. As the algorithm is implemented in a database procedure, it will now be acquire in view of the database (see Figure 7.1).

To dedicate a tape request to a free tape drive, the following preconditions have to be fullfilled:

- The selected row of the *TapeDrive* table has to have in status, UNIT_UP
- The tape server of the selected tape drive has to be in status TAPESERVER_ACTIVE
- The row of the *TapeRequest* table must not be associated with a tape drive
- Both tape drive and tape request have to belong to the same device group. This can be verified using their association to the *DeviceGroupName* table.
- The tape model of the requested tape has to appear in the priority list of the drive model. The priority list is handled in the *TapeDriveCompatibility* table and has been discussed in Section 7.1.1.
- The *TapeDriveDedication* table has to be checked, whether there are at the moment some usage restrictions for the selected row of the *TapeDrive* table (see Section 7.1.3).

On the C++ side, only the information of the first matching couple is sent for an assign to the Remote Tape Copy Daemon [RTCPD]. For this reason, the query has just to select the first found free tape drive and the best result of the tape request filtering. The logic for the dedication is implemented in the *TapeDriveDedicationHandler* class.

The matching rows of the *TapeRequest* table are sorted ascending to their priority in the *TapeDriveCompatibility* table, and subsequent descending to their modification time. The modification time value differs only to the creation time, if a previous assign trial with RTCPD failed.

Due to the fact that no other database query is manipulating the entries of free tape drives and waiting tape request, the selected rows in the involved tables do not have to be locked, which increases the performance and avoids deadlocks.

7.2 The Protocol Facade

After the detailed description of the request handling in the database, this Section will focus on those components of the new architecture, which are needed to handle the incoming messages.

All older protocols, which were developed for the first CASTOR version, initially send of all their magic numbers. Given the magic number it is possible to determin the protocol used. In the new VDQM architecture, the *ProtocolFacade* class determines the protocol and calls the correct processes for handling. The intention is to provide a higher-level interface that makes the subsystem easier to use. The class was named after its structural "Facade Pattern", that was defined by the "Gang of four [29]".

The sequence diagram in Figure 7.2 points up the included steps to read out the old protocol, which is used by the tape daemons and RTCPClientD. In the future it is planned to reimplement also those components. In this case, they would use a different protocol, which can easily be included in the *ProtocolFacade* class. Then of course, an additional protocol- interpreter and handler would have to be implemented.

- The *VdqmServer* class accepts a client socket connection and assigns it to a thread in its thread pool. The basic C socket is embedded in the *VdqmServerSocket* class, which provides more abstract functions to handle the read and write requests to the socket.

- The assigned thread calls then the *handleProtocolVersion()* function of the *ProtocolFacade* class, which manages the correct handling of the message in the buffer of the socket. For the *VdqmServer* class purposes, the protocol handling is hidden. The *VdqmServerSocket* object is forwarded to all classes, which have to access to the socket.

- The *ProtocolFacade* class uses the *VdqmServerSocket* to read out the first four bytes of the socket, that corresponds to the magic number of the protocol. On this basis, it selects the right protocol interpreter to unmarshall the remaining message, and to store the information into the corresponding data objects. The *OldProtocolInterpreter* acts as a translator towards the *ProtocolFacade* and provides all functions to handle the old protocol.

  The *OldRequestFacade* class handles then the client request. As the *ProtocolFacade* class it offers a higher-level interface to hide the complex subjacent structure, which will be described in detail in the next Section.

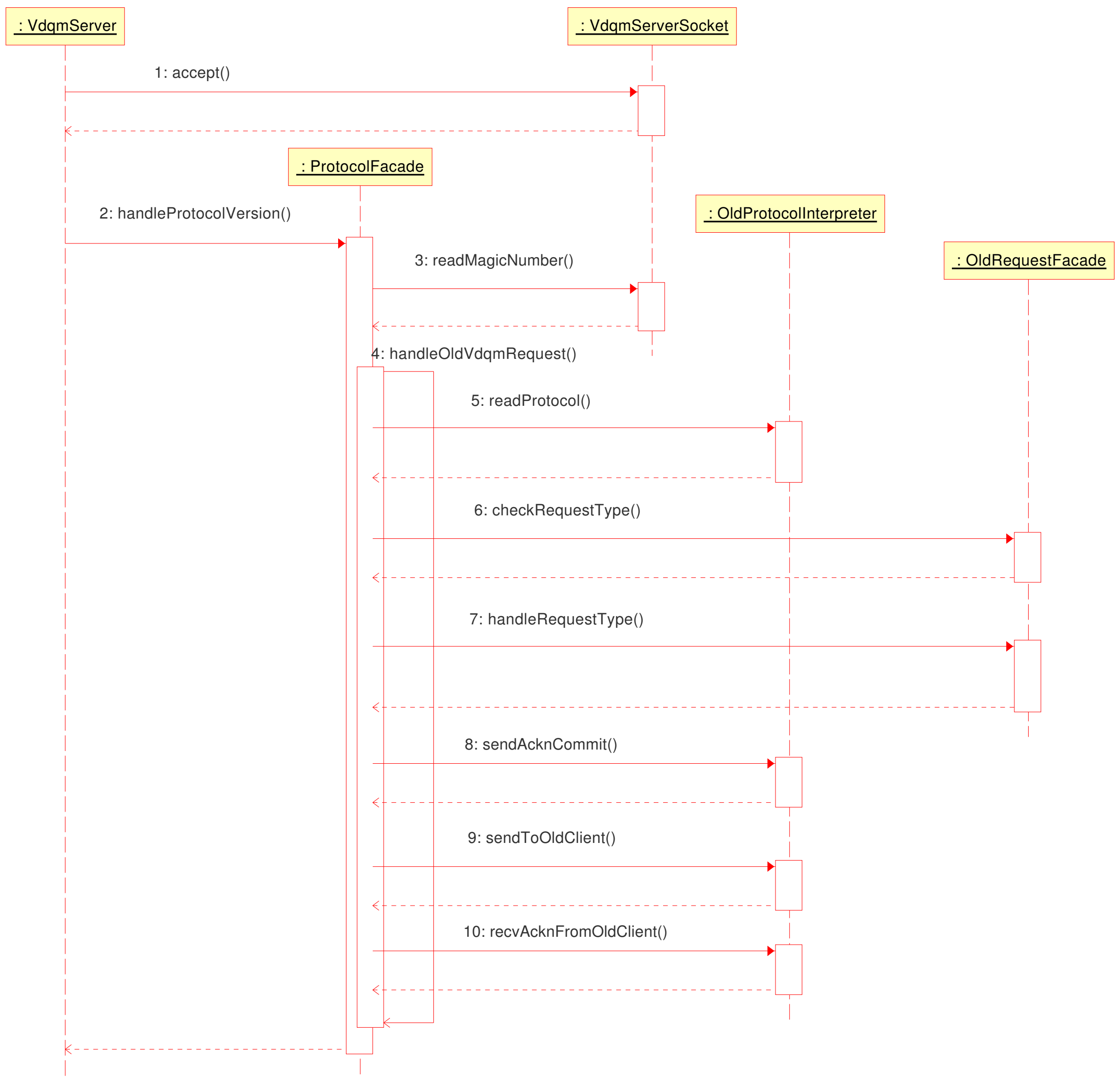

Figure 7.2: Sequence diagram of the Protocol Facade

- In case where the request handling was successful, the *ProtocolFacade* performs the handshake phase, as described in Section 5.2.1. Different than in the old implementation, we have to do an explicit "commit" at the end of the protocol. This is needed to store all changes in the database which have been done during the request handling.

  Figure 7.2 does not explicitly show the rollback behaviour of the *ProtocolFacade*, but it uses therefore also helper functions of the *OldProtocolInterpreter* class.

7.3 Handling of Incoming Messages

As explained in the last Section, the handling of the accepted request message is hidden behind the *OldRequestFacade* class. Due to the VDQM message type, which is sent with the message header (see Section 5.2.1, 5.3), the *OldRequestFacade* is able to choose the right function of the two handler facilities. The handler classes implements the logic, which realises the handling of the information and associations in the database, as described in Section 7.1.

All handler classes derives from the same *BaseRequestHandler* class (see Figure A.1), which provides functions to add, remove and update rows in the database. These functions are based on existing provided CASTOR2 API.

Moreover, it handles instantiation of the *IVdqmSvc* interface, which is used by the handlers to launch queries on the database. The association with the *IVdqmSvc* interface depends on the specification of the VDQM database in the CASTOR configuration file. Unfortunately, at the moment CASTOR supports only the Oracle database.

7.3.1 Handling of the Remote Tape Copy Client Messages

The *TapeRequestHandler* class manages the defined use cases for the Remote Tape Copy Client Daemon [RTCPClientD] , which have been elaborated in Section 6.2.

The sequence diagram in Figure 7.3 illustrates the interactions with the database interface of the new VDQM architecture, which are necessary to store a new tape request.

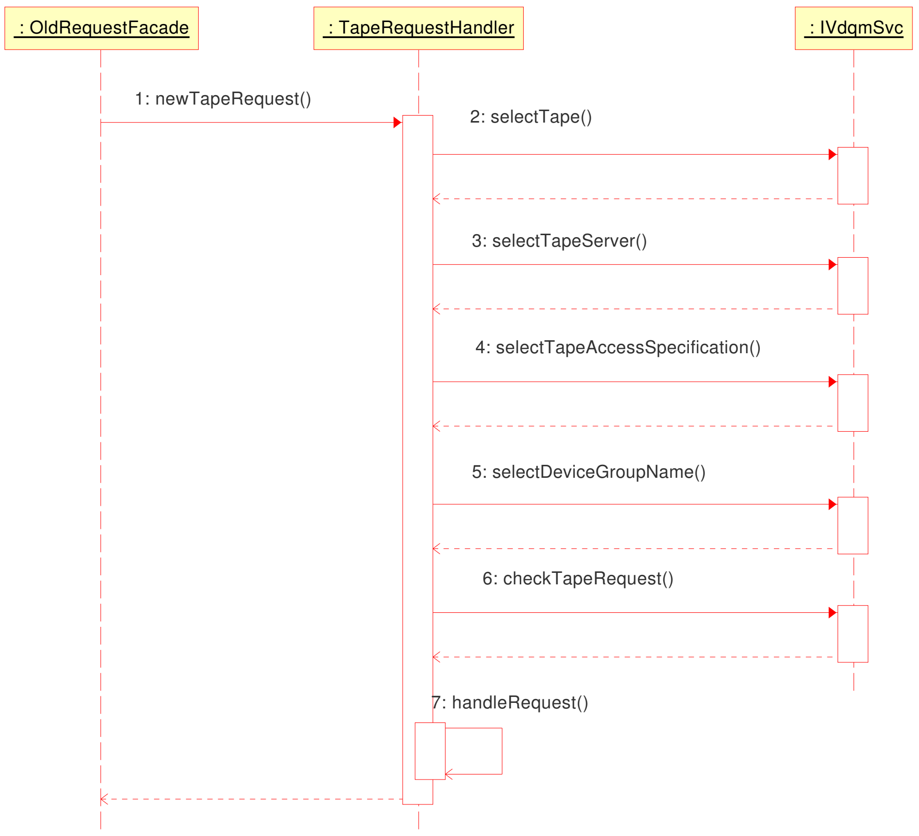

Figure 7.3: The handling of a new tape request

Description of the sequence diagram of Figure 7.3:

- If the message type corresponds to a new incoming tape request message [VDQM_NEW_VOLREQ], then the *OldRequestFacade* class calls the *newTapeRequest()* function of the *TapeRequestHandler*.

- Before the new tape request is stored in the database, several preparations have to be made to handle all mandatory associations in the database (see Section 7.1). For this purpose, a *TapeRequest* container class is filled with all corresponding object representations of the database table rows. The retrieve of the database information, as described in Section 4.2.2, is handled from the concrete instance of the *IVdqmSvc* interface.

- If the desired tape request representation could successfully be assembled with the container classes, then the *checkTapeRequest()* function controls, that the request does not yet exist in the database. Subsequently, the tape request can be inserted into the database with the inherited *handleRequest()* function of the *BaseRequestHandler*.

- The protocol requires, that VDQM sends back the assigned tape request id to RTCPClientD. Since the unique row id in the database is represented by a 64 bit integer value, it is not possible to give the exact id, because the protocol supports only a 32 bit value. The problem is solved through a simple downcast of the value, as it was analysed in Section 5.4.

Each tape request is handled separately by a child process of RTCPClientD. These child processes contacts regularly VDQM with the VDQM_PING message to check whether the connection can still be established, and to receive the queue position of their tape request. For this purpose, they have to send in the body message the id of the tape request. The internal handling of this request is illustrated in the sequence diagram of in Figure 7.4.

Description of Figure 7.4:

- As mentioned before, only the downcasted request id can be sent through the protocol. To obtain now again the 64 bit value, a database query has to downcast all row id's of the *TapeRequest* table and to compare them with the

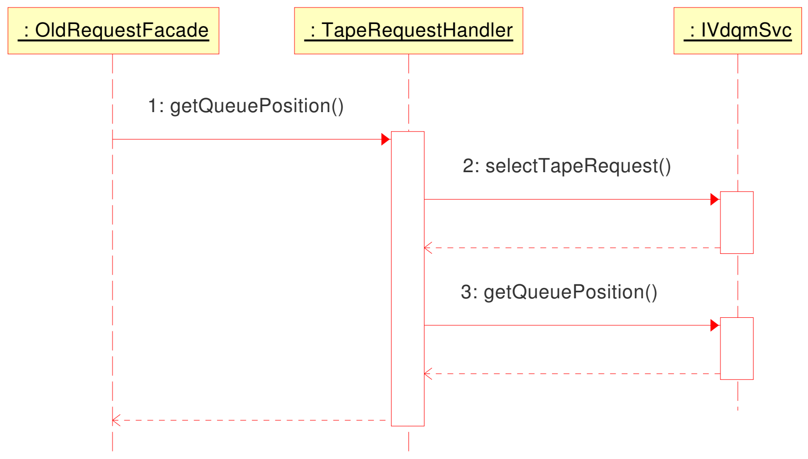

Figure 7.4: Handling of the queue position message for a tape request

sent value. This query is included in the *selectTapeRequest()* function of the concrete *IVdqmSvc* instance, which returns then a container class with the information of the determined row.

- The queue position is zero, if the tape request is associated to a tape drive, which testifies that the request is actually executed. Otherwise, the *IVdqmSvc getQueuePosition()* function computes an approximate queue position.

  Of course it would be possible to determine the exact number, but it is not necessarily needed. This algorithm would be strongly connected to the assign algorithm of a tape to a tape drive. Since it is recommended that the assign procedure can be changed during the running application, it would be quite exhausting to change each time the queue position algorithm, too. Hence, the approximate algorithm adds just those tape request up, which belongs to the same device group and have at the same time an older modification time. The asymmetric tape requests are so not considered, but this makes the query much faster, which is in this case also a requirement.

The removing of a tape request from the database works quite simple, as the illustration in Figure 7.5 shows:

- The Remote Tape Copy Client has to send with the VDQM_DEL_VOLREQ request only the 32 bit id representation of the tape request, which has to be deleted. Again, the *selectTapeRequest()* function is used to retrieve on basis of this value the right TapeRequest from the database.

- If the tape request is not associated to a tape drive, then the information in the *TapeRequest*- and *ClientIdentification* table can be removed by means of the inherited *deleteRepresentation()* function of the *BaseRequestHandler* class.

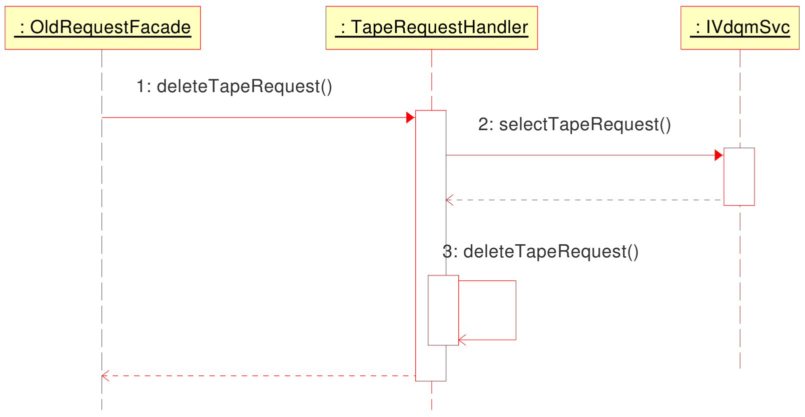

Figure 7.5: Removing a tape request

7.3.2 Handling of Tape Daemon Messages

The tape daemon contacts VDQM to update the status of its tape drives or to remove a drive from VDQM. The status message is included in the body message, which includes of course also all needed information to determin the specific tape drive. The sequence diagram in Figure 7.6 visualise the several steps, which are necessary to handle the VDQM_DRV_REQ message. The *newTapeDriveRequest()* function consists of four main steps:

1. The determination of the tapedrive and the retrieve of the information from the database
2. The verification of the tape drive status consistency
3. The operational discharge of the necessary changes in the database
4. The update of the database

The database changes for each status have been listed in detail in the Table 7.2 of Section 7.1.

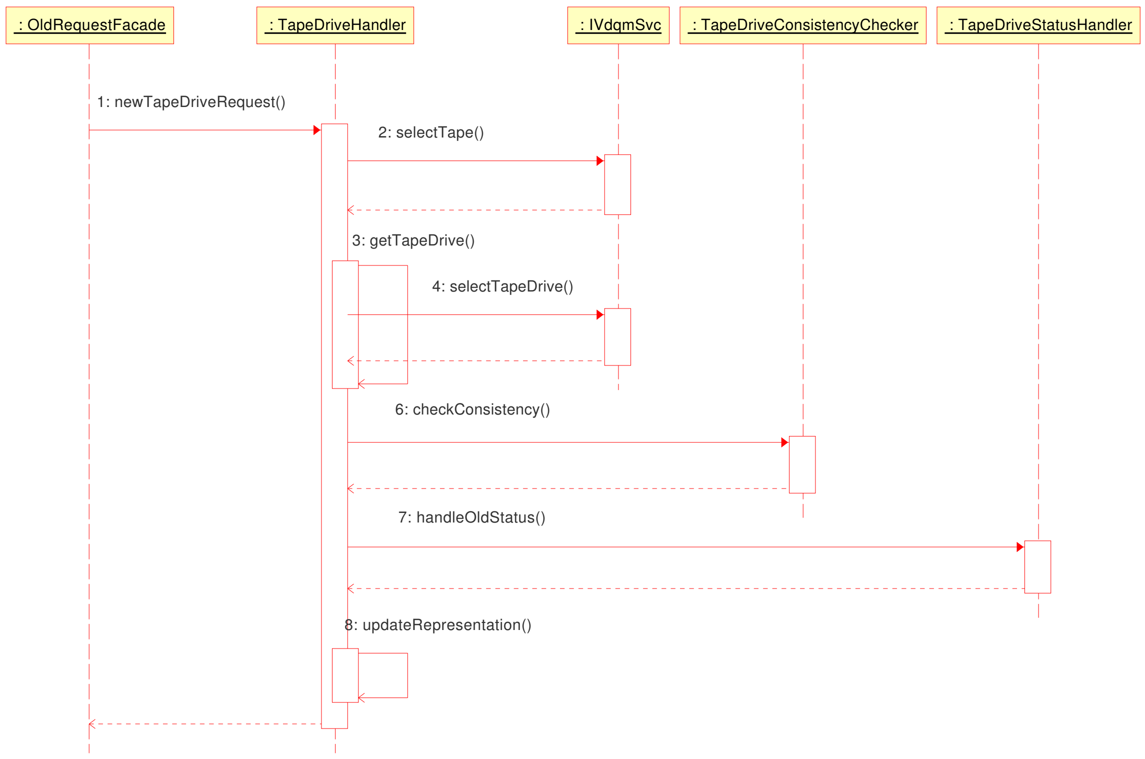

Figure 7.6: Handling of a status update message for a tape drive

Detailed description of the sequence diagram in Figure 7.6:

- A tape drive is clearly defined through its name and the its belonging tape server. Due to this information it is easy to find the right row in the *TapeDrive* table. Hence, it was not needed to implement a work around for 64 bit unique id's, as for the tape request handling.

  The *selectTapeServer()* function handles the creation- and/or selection of the tape server row. Then, the internal *getTapeDrive()* function is used to manage the recall or initialisation of the tape drive. For a better overview, the sequence diagram shows only the case, that a tape drive already exist the database.

  If the message of the tape daemon is for an unknown tape drive, VDQM creates a new entry in the *TapeDrive* table and manages the association to the DeviceGroupName. Furthermore, the *TapeDriveCompatibility* table has to be initialised as described in Section 7.1, if it contains no entries for its tape drive model.

- The *TapeDriveConsistencyChecker* class verifies, that VDQM can switch the tape drive to the status, which has been sent by the tape daemon. For it the *TapeDriveConsistencyChecker* has to check, whether the sent status has been expected as next for the tape drive. Since the tape daemon is always to trust, it tries anyhow to adapt its drive information, so that the request can be handled by the *TapeDriveStatusHandler* class. In the worst case, the tape drive status is marked as unknown until the tape daemon reverifies it.

- After a successful consistency check, the *TapeDrive* container object, that has been created before out of the values in the database, is ready to be forwarded to the *TapeDriveStatusHandler* class. This facility manages the manipulation of the information in the container class, depending on the sent status. The complexity of this method is hidden behind the *handleOldStatus()* function call, which returns at the end the updated *TapeDrive* container.

- To update the tape drive row and its associations, the *newTapeDriveRequest()* function simply forwards the container object to the *updateRepresentation()* function of the parent class.

7.3.3 Handling of Administration Requests

Removing a tape drive

When the administrator uses the VDQM Client interface to remove a tape drive from the database, the *OldProtocolFacade* class has to call the *deleteTapeDrive()* function of the *TapeDriveHandler*. As for the handling of the tape drive status messages, the selectTapeDrive() function is used to recall the drive information of the database. Due to the unique row id, the inherited *deleteRepresentation()* function is able to remove the information from the *TapeDrive* table.

The described function calls are illustrated in the sequence diagram of Figure 7.7.

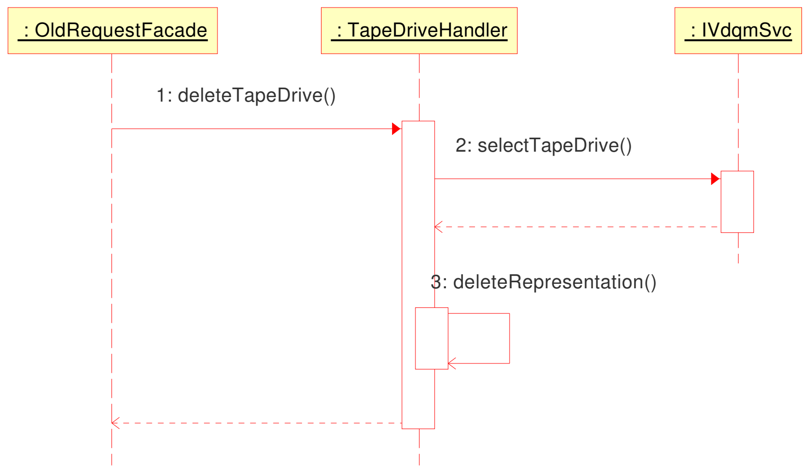

Figure 7.7: Removing a tape drive

Handling of the *showqueues* Command

The *showqueues* command is handled in two steps, which processes in similar ways. The sequence diagram of Figure 7.8 illustrates the transferring of the tape drive queue to the client.

First, the VDQM_GET_DRVQUEUE message is sent, which is handled from the TapeDriveHandler class in the sendTapeDriveQueue() function. There, the information from the *TapeDrive* table are recalled from database over the *IVdqmSvc* interface. The information are then mapped to the old protocol, which is done sequentially for each tape drive by the *sendToOldClient()* function. At the end, the client expects an empty tape drive information message with a -1 specified as tape drive id.

As the sequence diagram in Figure 7.9 shows, exists a corresponding structure for the tape request queue in the *TapeRequestHandler*, which will in this context not further be explained.

7.4 Thread Pool Management

The new architecture possesses about two independent thread pools, to decouple the handling of incoming request from the assignment of a tape request to a free tape drive. Two thread pools ensure, that there are always enough resources for

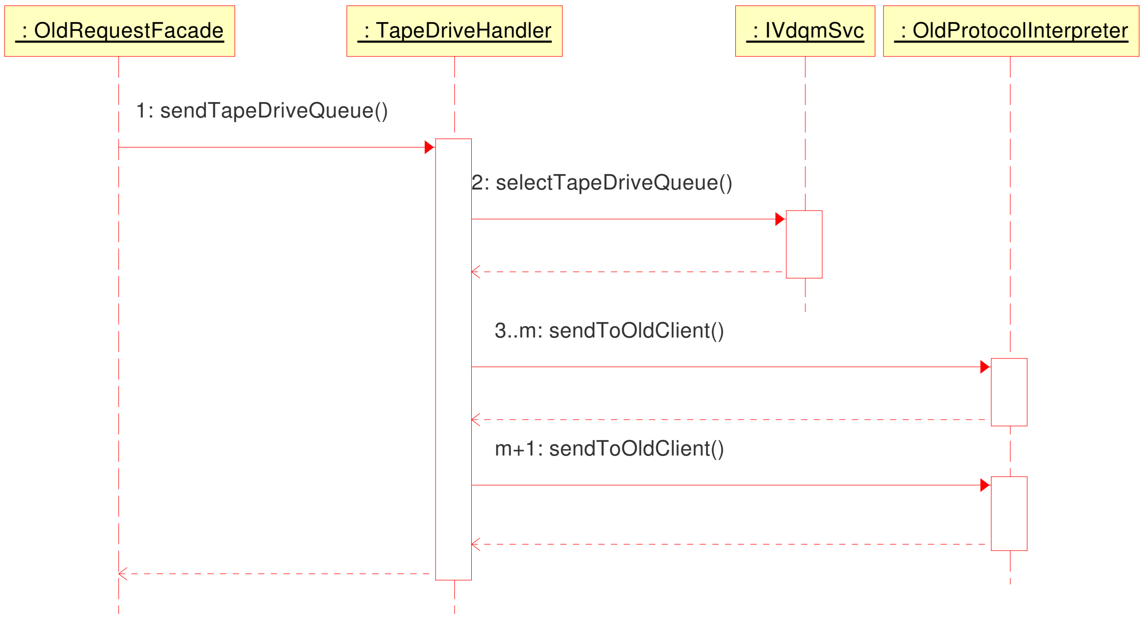

Figure 7.8: Requesting the tape drive list for the *showqueues* command

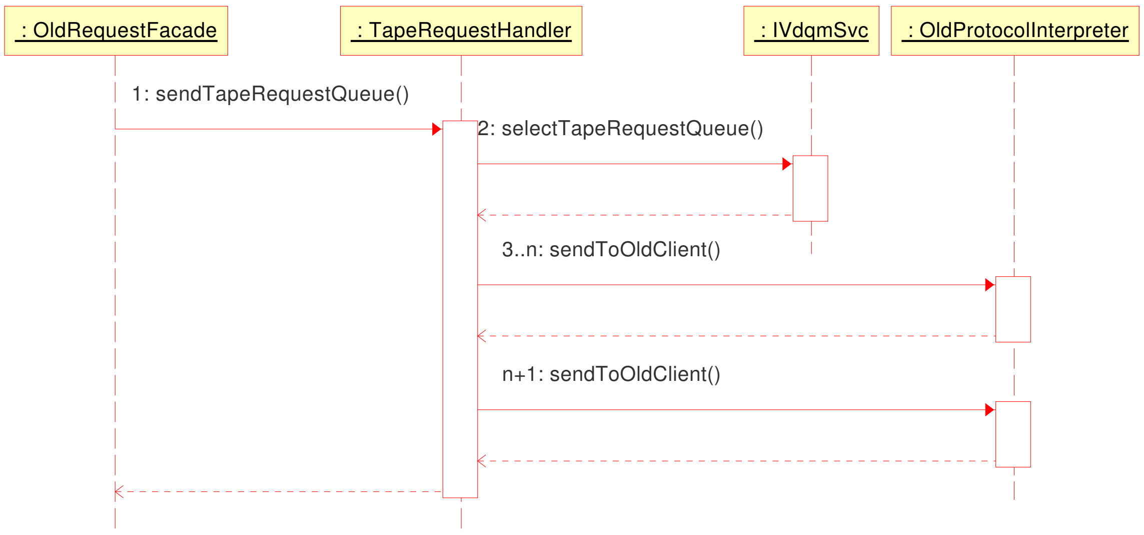

Figure 7.9: Accessing the tape request queue for the *showqueues* command

both tasks. Furthermore, the amount of available threads in a pool can be chosen separately.

The activity diagram of Figure 7.10 illustrates the thread pool handling of the new architecture.

1. When the new VDQM server starts, it has first of all to initiate the messages for the DLF logging facility (Please, see also Section 4.3).

2. A special class, called the *TapeRequestDedicationHandler*, is handling the assignment of queued tape request to free tape drives. Therefore, it queries the VDQM database tables, as discussed in Section 7.1.4.

   The VDQM server calls the *TapeRequestDedicationHandler* in an extra thread, which manages its own thread pool. This pool is needed to do the calls to Remote Tape Copy Daemons independently to the database query process. The details of the protocol to RTCPD are implemented in the *RTCopyDConnection* class, which is instantiated in each single thread.

   The *TapeRequestDedicationHandler* class is designed with the Singleton pattern, to ensure that only one instance is in use.

3. After having forked the *TapeRequestDedicationHandler* in an extra thread, the VDQM server opens a socket to accept incoming requests from RTCPClientD or the tape daemons. Also there, each accepted connection is handled in an own thread. This avoids a slow down of the connection accept loop of the *VdqmServer* class.

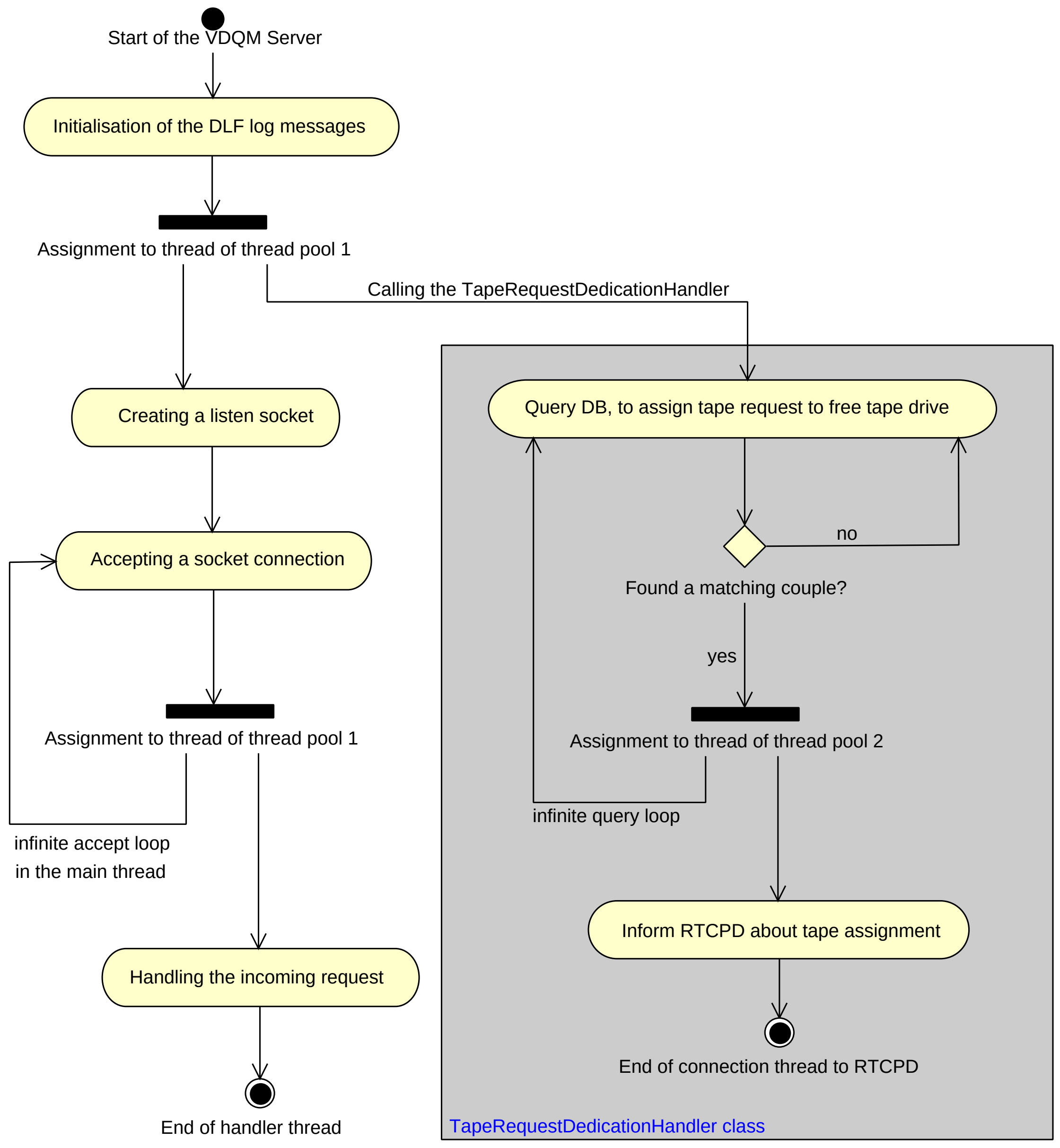
Start of the VDQM Server
Initialisation of the DLF log messages
Assignment to thread of thread pool 1
Calling the TapeRequestDedicationHandler
Creating a listen socket
Accepting a socket connection
Assignment to thread of thread pool 1
infinite accept loop
in the main thread
Handling the incoming request
End of handler thread
Query DB, to assign tape request to free tape drive
no
Found a matching couple?
yes
Assignment to thread of thread pool 2
infinite query loop
Inform RTCPD about tape assignment
End of connection thread to RTCPD
TapeRequestDedicationHandler class

Figure 7.10: Activity diagram of the thread pool handling

Chapter 8

Implementation of the Robotic Tape Queueing System

After the detailed architectural description elaborated in the previous Chapter, the adjacent implementation process will now be described, and relevant code examples will be given. Unfortunately, due to the size of the project, only the most important parts of the code can be presented.

The description of the development process will assume that the designed database base structure is already deployed with the code generator of CASTOR2.

8.1 Development Process

The development was started with the basic program structure, that opens a listen socket and assigns accepted connections to a thread of the main thread pool. This is used by the Request Handler component to accept incoming client requests. Hence, also the complex thread pool architecture was already implemented.

The first milestone was to implement the basic structure of the request accept protocol (see Section 5.2.1). Therefore, the transactions through the socket had to be realised by means of old CASTOR C functions. They assure a proper marshalling and unmarshalling of the message between the network byte order and the order of the used host system. On this basis, the verification of the protocol version and the determination of the request message type could be realised.

First tests of the so far implemented code could then be achieved by sending requests with the aid of a test tool. That was originally been written to test the old VDQM implementation. Up to this point, the code is ran through by every incoming request, as described in Section 7.2.

The implementation of the different request types in the handler facade (see Section 7.3) was started subsequently. This work also included the realisation of the database facility, to allow immediate testing of each implemented request.

The focus was initially on the tape request handling that was much easier to code than the tape drive handling. The aim was to be able to store as soon as possible a request in the database. The use of the existing CASTOR2 architecture to deal with database tasks needed some expert knowledge. The collected experiences then helped to develop quicker the other request types.

The last step was to implement the logic to dedicate a tape request to a compatible tape drive. Again the existing thread pool architecture was used to handle the tape assign protocol (see Section 5.2.2) between the Remote Tape Copy Daemon and VDQM.

Only now, the reimplemented Volume and Drive Queue Manager could be tested as one acting unit.

The VDQM server can be started in the forground or as backround process [daemon]. It is also possible to specify the amount of available threads in each thread pool. Per default are twenty threads in each pool available.

8.2 Using the Distributed Logging Facility

To use the Distributed Logging Facility [DLF] (see also Section 4.3)together with the DLF server, all logging messages have there first to be registered. Hence, each message possess about an explicit number. This design decision was taken, to be able to query the log database for a specific message number. Listing 8.1 shows how the initialisation in the *VdqmServer* class is done.

```
castor::vdqm::VdqmServer::VdqmServer():
  m_foreground(false),
  m_threadPoolId(-1),
  m_threadNumber(DEFAULT_THREAD_NUMBER),
  m_dedicationThreadNumber(DEFAULT_THREAD_NUMBER),
  m_serverName("VdqmServer") {

  // Initialises the DLF logging. This includes
09   // registering of the predefined messages
10   castor::dlf::Message messages[] =
11    {{ 0, " - "},
12     { 1, "New Request Arrival"},
13     { 2, "Couldn't get Conversion Service for Database"},
14     { 3, "Couldn't get Conversion Service for Streaming"},
15     { 4, "Exception caught : server is stopping"},
16
17        // etc.
18        ...
19
20     {-1, ""}};
21   castor::dlf::dlf_init("Vdqm", messages);
22 }
```

Listing 8.1: DLF initialisation

The static function *dlf_init()* needs two parameters to execute the initialisation (see Line 21). The first parameter determines the name of the DLF facility to use. In the configuration file of CASTOR, the name has to be mapped against a local file and/or a host, which runs a DLF server. The standard entry, for instance, to direct all log message to the DLF server on host *lxs5010.cern.ch* is:

```
Vdqm LOGALL x-dlf://lxs5010.cern.ch/
```

The second parameter of *dlf_init()* is the list of log messages, which is used in the component. The list has to be defined as shown between Line 10-20.

The example in listing 8.2 shows how to create an error log message:

```
01 //Initialisation of the parameter list
02 castor::dlf::Param params[] =
03  {castor::dlf::Param("Standard Message",
05                      sstrerror(e.code())),
06   castor::dlf::Param("Precise Message",
07                      e.getMessage().str())};
08
09 // Creation of an error log message for message number 4
10 castor::dlf::dlf_writep(cuuid,
11                         DLF_LVL_ERROR, 4, 2, params);
```

Listing 8.2: Example: DLF error message with two parameters

The log message is created with the static function *dlf_writep()* (see Line 10). The first parameter specifies a unique id, which is used for all log messages of one event. This enables to filter out all logs of one request in the DLF web interface.

The second parameter determines the severity of the log message, and the third parameter the message number. With the last two parameters we can optionally specify a list of log information.

The log messages of the following examples in this Chapter will not further be explained.

8.3 Protocol Determination

The *ProtocolFacade* class is the place, where the decision for the right protocol handling is taken. At the moment, only the old protocol has to be identified. However, in the future the clients are going to be reimplemented in C++ and then the protocol will no longer be based on C structs. At this point it will be necessary to implement on VDQM side a new protocol interpretation.

The *handleProtocolVersion()* function, which is listed in Listing 8.3, has then to be extended, so that the new protocol can be recognised. Therefore it is a prerequisite, that the magic number is still sent in the first four bytes of the message.

```
void castor::vdqm::ProtocolFacade::handleProtocolVersion()
  throw (castor::exception::Exception) {

  //The magic Number of the message on the socket
  unsigned int magicNumber;

  // Read the incoming request
  try {
    //First check of the Protocol
    magicNumber = ptr_serverSocket->readMagicNumber();
  } catch (castor::exception::Exception e) {
    // "Unable to read Request from socket" message
    castor::dlf::Param params[] =
      {castor::dlf::Param("Standard Message",
                          sstrerror(e.code())),
       castor::dlf::Param("Precise Message",
                          e.getMessage().str())};
       castor::dlf::dlf_writep(*m_cuuid, DLF_LVL_ERROR,
                               7, 2, params);
  }

  switch (magicNumber) {
    case VDQM_MAGIC:
      // "Request has MagicNumber from old VDQM Protocol"
      castor::dlf::dlf_writep(*m_cuuid,DLF_LVL_SYSTEM,6);

      try {
        // Call the function, which handles the
        // old request accept protocol
        handleOldVdqmRequest(magicNumber);
      } catch (castor::exception::Exception e) {
        // Most of the exceptions are
        // handled inside the function

35           // "Exception caught" message
36           castor::dlf::Param params[] =
37             {castor::dlf::Param("Message",
38                             e.getMessage().str().c_str()),
39              castor::dlf::Param("errorCode", e.code())};
40           castor::dlf::dlf_writep(*m_cuuid, DLF_LVL_ERROR,
41                                   9, 2, params);
42         }
43         break;
44
45       // Please, insert here cases for new Protocols!
46
47       default:
48         // "Wrong Magic number" message
49         castor::dlf::Param params[] =
50           {castor::dlf::Param("Magic Number", magicNumber),
51            castor::dlf::Param("VDQM_MAGIC", VDQM_MAGIC)};
52            castor::dlf::dlf_writep(*m_cuuid, DLF_LVL_ERROR,
53                                    13, 2, params);
54   }
55 }
```

Listing 8.3: Protocol determination

Explanation of Listing 8.3:
The *ProtocolFacade* object receives at its instantiation a pointer to a *VdqmServerSocket* object, which includes the accepted socket connection of the new request. The *VdqmServerSocket* class provides a function to read out the first four bytes (see Line 10), which represents the magic number of the used protocol. It is compared with the static magic number constants (see lines 22-54), to determine how VDQM has to treat the rest of the message.

8.4 Container Class Handling

This Section will discuss the code example in Listing 8.4, which shows the implementation of the *handleRequest()* function from the *BaseRequestHandler* class. As explained in Section 7.3.1 and 7.3.2, all handler classes are derived from the *BaseRequestHandler* class, which provides functions to handle the storing, updating and deleting of information in the database. The *handleRequest()* function is used to store a new request into a table of the database, depending on the forwarded container class. The other helper functions of that class are designed in an equivalent way.

```
01 void
02   castor::vdqm::handler::BaseRequestHandler::handleRequest
```

```
(castor::IObject* request)
throw (castor::exception::Exception) {

castor::vdqm::TapeRequest *tapeRequest = 0;
castor::vdqm::TapeDrive *tapeDrive = 0;

// Stores it into the data base
castor::BaseAddress baseAddr;
baseAddr.setCnvSvcName("DbCnvSvc");
baseAddr.setCnvSvcType(castor::SVC_DBCNV);

try {
  // Creates a new entry in the table
  svcs()->createRep(&baseAddr, request, false);

  // Stores the TapeRequest associations
  tapeRequest =
    dynamic_cast<castor::vdqm::TapeRequest*>(request);
  if (0 != tapeRequest) {
    svcs()->createRep(&baseAddr,
                      (IObject *)tapeRequest->client(),
                      false);

    svcs()->fillRep(&baseAddr, request,
                    OBJ_ClientIdentification, false);
    svcs()->fillRep(&baseAddr, request,
                    OBJ_Tape, false);
    svcs()->fillRep(&baseAddr, request,
                    OBJ_DeviceGroupName, false);
    svcs()->fillRep(&baseAddr, request,
                    OBJ_TapeAccessSpecification, false);
    svcs()->fillRep(&baseAddr, request,
                    OBJ_TapeDrive, false);
    svcs()->fillRep(&baseAddr, request,
                    OBJ_TapeServer, false);
  }

  // Stores the TapeDrive associations
  tapeDrive =
    dynamic_cast<castor::vdqm::TapeDrive*>(request);
  if (0 != tapeDrive) {
    // similar handling as for the TapeRequest
    ...
  }

  // Handling of the associations of
  // other container classes
  ...
} catch (castor::exception::Exception e) {
  // Error occurred! Rollback of the transaction
  svcs()->rollback(&baseAddr);

  castor::vdqm::TapeRequest *tapeRequest =
    dynamic_cast<castor::vdqm::TapeRequest*>(request);
  if ( 0 != tapeRequest ) {
    // EVQNOVOL: error message number for RTCPClientD
59        castor::exception::Exception ex(EVQNOVOL);
60        ex.getMessage() << e.getMessage().str();
61        tapeRequest = 0;
62        throw ex;
63      }
64
65      // Error handling for the other container classes
66      ...
67    }
68  }
```

Listing 8.4: Storing of new database entries

Every container class is derived from an *IObject* class. It is represented by the parameter of the function header (*request*). A *BaseAddress* object has to be prepared (see Line 10-12) to recall the needed service to store the data of the container class into the database. The recall of a service has been discussed in Section 4.2.2.

The *BaseRequestHandler* class itself is derived from the *BaseObject* class, which represents the root of every functional object in CASTOR2. It provides basic functionalities for the error handling and for the service calls.

The *svcs()* function of the *BaseObject* class returns in Line 16 an access to the Service Manager (see Figure 4.4). The *createRep()* functions inserts then a new row for the container class into the database. Its third parameter specifies whether the inserted row should immediately be committed, or not. As it is mandatory to do a rollback of the whole request in case of errors, the commit is done not until the whole run through of the protocol process (see 7.2).

Each container object can include other container objects, which represent the association to row of another table. The association have to be committed separately. Therefore, dynamic casts (see Line 20 and 42) are done to determine during runtime the forwarded container class representation. The *fillRep()* function is used to create the association between the two rows. The third parameter determines the object type of the foreign table.

If an error occurs during the transaction, an *Exception* will be thrown. Before it is hand over to the function caller, the transaction is rolled back (see Line 53) and the appropriate error number is determined for the request client.

8.5 Database Query Handling

The recalling of information with a database query has been represented in the sequence diagrams of the previous Chapter via the *IVdqmSvc* interface.

Listing 8.5 gives a concrete example for the *selectTapeRequest* function from the IVdqmSvc. The function is used to recall tape request information from the Oracle database with help of the 32 bit representation of the unique row id. The problem has been discussed in Section 7.3.1.

The other Oracle functions of the *IVdqmSvc* interface have been developed in a similar ways.

```
/// SQL statement for function selectDeviceGroupName
const std::string castor::db::ora::OraVdqmSvc::
  s_selectTapeRequestStatementString =
  "SELECT id FROM TapeRequest WHERE CAST(id AS INT) = :1";


castor::vdqm::TapeRequest*
  castor::db::ora::OraVdqmSvc::selectTapeRequest(
  const int VolReqID)
  throw (castor::exception::Exception) {

  // The 64 bit representation of the tape request id
  u_signed64 id;

  // Checks whether the statements are OK
  if (0 == m_selectTapeRequestStatement) {
    m_selectTapeRequestStatement =
      createStatement(s_selectTapeRequestStatementString);
  }
  // Execute statement and get result
  try {
    // Gives the 32 bit representation of
    // the tape request id as parameter
    m_selectTapeRequestStatement->setInt(1, VolReqID);
    oracle::occi::ResultSet *rset =
     m_selectTapeRequestStatement->executeQuery();

    if (oracle::occi::ResultSet::END_OF_FETCH
        == rset->next()) {
      m_selectTapeRequestStatement->closeResultSet(rset);
      // we found nothing, so let's return NULL
      return NULL;
    }
    // If we reach this point, then we selected
    // successfully a tape and it's id is in rset
    id = (u_signed64)rset->getDouble(1);
    m_selectTapeRequestStatement->closeResultSet(rset);
  } catch (oracle::occi::SQLException e) {
    ...
    throw e;
41   }
42
43   // Now, get the tape from its id
44   try {
45     castor::BaseAddress ad;
46     ad.setTarget(id);
47     ad.setCnvSvcName("DbCnvSvc");
48     ad.setCnvSvcType(castor::SVC_DBCNV);
49     castor::IObject* obj = cnvSvc()->createObj(&ad);
50     castor::vdqm::TapeRequest* tapeRequest =
51     dynamic_cast<castor::vdqm::TapeRequest*> (obj);
52     if (0 == tapeRequest) {
53       castor::exception::Internal e;
54       e.getMessage()
55         << "createObj return unexpected type "
56         << obj->type() << " for id " << id;
57       delete obj;
58       obj = 0;
59
60       throw e;
61     }
62
63     // Get the foreign related object
64     cnvSvc()->fillObj(&ad, obj,
65                       castor::OBJ_ClientIdentification);
66     cnvSvc()->fillObj(&ad, obj, castor::OBJ_TapeDrive);
67     obj = 0;
68
69     return tapeRequest;
70   } catch (oracle::occi::SQLException e) {
71     ...
72     throw e;
73   }
74   // We should never reach this point
75 }
```

Listing 8.5: Recalling of the tape request from the database

As with every service instance, the concrete *IVdqmSvc* instance is managed from the Service Manager. This means, that the object is instantiated only once in the whole lifetime of VDQM.

Each database query is stored in a global *private static* string constant (see Line 02-04). This is required to create once and for all the global *private oracle::occi::Statement* object (see Line 16-19) from the Oracle library for this function in the heap memory. All global objects are declared in the header file of the class.

After the 32 bit id of the tape request is given as first parameter of the statement (see Line 24, 04), it is ready to be executed (see Line 25-26). The result of the database query returns the 64 bit representation of the id number in an *ResultSet* object. This object is a linked list of the found results. In this case it contains only one value.

If the *ResultSet* does not contain any value, the function will return a *NULL* pointer (see Line 28-33). Otherwise the recalled id is stored in an *unsigned* 64 bit variable. In both cases, the connection to the database has to be closed (see Line 30, 37).

With the 64 bit id it is possible to use the Service Manager facility for retrieving the database information of that row. As described in Section 4.2.2, the data are put together in the appropriate container class by the Converter of the table. This information recall is handled with the *createObj()* function of the Service Manager (see Line 49).

If available, the associations of a table row can additionally be recalled with the *fillObj()* function (see Line 64-66). Similar to the storing process (see Section 8.4), the associated container objects have to be specified by their object type constant. The returning *TapeRequest* object contains then all recalled data objects.

Chapter 9

Summary and Conclusion

This Chapter will give a summary of the experiences gained during the implementation and testing phases. Furthermore, we will evaluate whether the technical requirements have been achieved with the reimplementation of the robotic queueing system. Finally, we will discuss reasonable extensions and future prospects.

9.1 Evaluation

The new robotic tape queueing system implementation comprises approximately 20,000 lines of code. About 11,500 lines are autogenerated via the CASTOR2 code generator. This includes all container classes for the database tables and the helper classes for the Service Manager, such as Conversion Services and Factory classes.

The use of an autogeneration saves not only time, but makes an application much more robust. Since it is based on the XMI-file parser of the Open Source Project Umbrello, no licence fees have to be spent for a designing tool. In exchange, the programmers have to live with a defective graphical interface.

The stateless design guarantees a minimal information loss in case of a system crash. Moreover, the use of a database enables us to manage a very large number of tape requests. VDQM can run simultaneously on several machines for load balancing.

The implementation of two thread pools assures that there are always enough resources for the two main tasks: the handling of incoming requests and the sending of tape assign requests to the Remote Tape Copy Daemon. The number of threads in each pool can be determined via start parameters. By default, they are initiated with twenty threads.

Existing test tools, which were developed for the former Volume and Drive Queue manager, were useful to check whether the reimplementation could handle all request types. A stress test with five emulated tape drives and hundreds of tape request ran successful, after some small bug fixes. The new VDQM system was also tested with two HP LTO3 as well as with two IBM 3592 drive devices. The robotic tape library used was an IBM 3584, which was filled with different tape models. In this setup, the application ran without crashing for more than one week, which proved its robustness and error tolerance.

A test run in the production environment arose that the administrative *showqueues* command performed slowly with more than 1000 requests. This can be tuned with a lighter database query for retrieving all tape requests and their associated information from related tables.

Tape drives can be dedicated for specific jobs with entries into the *TapeDriveDedication* table of the database. A Perl Script serves as user interface and assures that the entries in that table are valid. The advantage is that the table can be extended for future needs without having to change the C++ implementation, because only the database procedure to assign tape requests to tape drives makes use of it.

The assignment of tapes to asymmetric tape drives can not yet be proven, because the protocol extension on the tape daemon side is still missing. However, the logic on VDQM is implemented and works fine in symmetric mode. Instead of the drive device name we use the cartridge model name to fill the *TapeDriveCompatibility* table.

The usage of the DLF logging facility has an unaesthetic side-effect, which is related to the mandatory initialisation of the log messages at start time (see Sections 4.3 and 8.2): Since every sent log message has to be referred by its log number, it is not possible to use the implemented classes in foreign CASTOR2 components. This fact reduces a lot the power of the object oriented design.

We conclude that the main problems of the old implementation, mentioned in Section 5.1, are solved. Furthermore, the code is easier to maintain, because of the object oriented design and the use of clear tape drive states. Above all, the system is extendable for new protocols and its database design allows the inclusion of additional features, such as error logging for the tapes and tape drives.

Because of missing CASTOR2 documentation, the preparation of Chapter 4 took a lot of time. The understanding of the single components has been acquired with the help of the CASTOR2 team members.

9.2 Extensibility and Future Prospects

A revision of the tape daemon still has to be done to support the asymmetric use of the tape drives (see Section 5.6). Moreover, in case of errors during a tape request handling, an informative message could be sent to VDQM. As described in Section 7.1. This message could be used to collect statistical data about the reasons of tapes and drives fail. For instance, it is interesting to know if some tape causes only problems on certain drives. The dedication pattern array of the old protocol, which is not used anymore (see Section 5.5), could be utilised for the transmission. This avoids an adaption of the other components, because the length of the message body would not change.

Since all information about VDQM is stored in the database, it enables us to write administrative web services. For a better overview about the states of the tape drives it is, for instance, suggestive to use a graphical interface.

The deactivation of a tape server for maintenance purposes could also be handled with such a web interface. Therefore, the status of a server has to be set to TAPESERVER_INACTIVE in the database. Its tape drives will then no longer be selected for an tape assignment, because the value is checked in the assign procedure.

Appendix A

VDQM Class Diagram

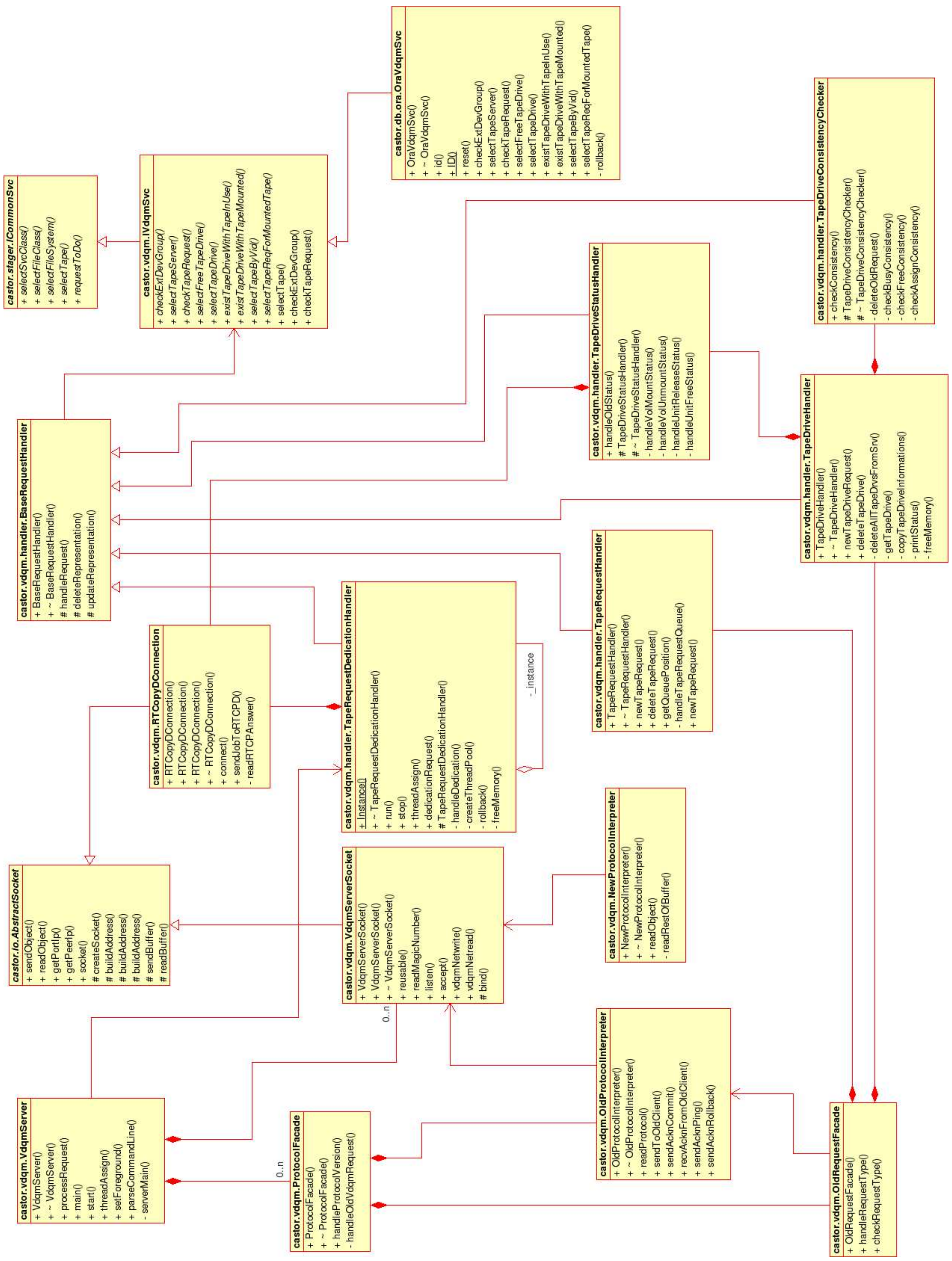

Figure A.1: The complete VDQM class overview

Appendix B

CD Contents

- PDF of this thesis (print version)
- PDF of this thesis (with highlighted hyperlinks)
- Source code of the CASTOR project
- Documentation of the source code in HTML format
- PDF files of the electronically available references
- All images used in this thesis

Appendix C

Glossary

ALICE - A Large Ion Collider Experiment at CERN's Large Hadrons Collider
API - Application Interface
ATLAS - A Toroidal Large Hadrons Collider ApparatuS
CASTOR - **C**ERN **A**dvanced **STOR**age manager
CCM - Configuration Cache Manager
CDB - Configuration Data Base
CERN - European Organization for Nuclear Research
CDR - Central Data Recording service
CLAPI - Client API from HPSS
CMS - The Compact Muon Solenoid
CVS - Concurrent Versions System

DESY - Deutsches Elektronen-Synchrotron
DLF - Distributed Logging Facility

EDG - European DataGrid project
ELFms - Extremely Large Fabric management system

FNAL - Fermi National Accelerator Laboratory

GridFTP - Grid File Transfer Protocol
GSI - Grid Security Infrastructure
GSSAPI - Generic Security Services Application Programming Interface
GssFTP - Generic Security Services File Transfer Protocol

HEP - High Energies Physics HSM - Hierachical Storage Management
HPSS - High Performance Storage System

IBM - Industrial Business Machines

LCG - LHC Computing Grid project (Distributed Production Environment for Physics Data Processing)
LCLI - Lemon Command Line Interface
LEAF - LHC-Era Automated Fabric
LEMON - LHC Era Monitoring
LEP - Large Electron Positron collider
LHC - The Large Hadrons Collider
LHCb - The Large Hadrons Collider study of CP violation in B-meson decays LSF
- Large Scale Facility
LTO - Linear Tape Open

MR - Monitoring Repository
MSA - Monitoring Sensor Agent

NCM - Node Configuration Manger

PKG - Solaris Package Manager
PNFS - The Perfectly Normal File System
POSIX - Portable Operating System Interface

Quattor - System Administration Toolsuite

RFIO - Remote File Input/Output
RPM - RPM Package Manager
RRD - Round Robin Database
RTCPD - CASTOR Remote Tape Copy Daemon
RTCPClientD - CASTOR Remote Tape Copy Client Daemon

SOAP - Service-Oriented Architectural Pattern
SPMA - Software Package Management Agent
SAN - Storage Area Network
STK - StorageTek

TCP - Transmission Control Protocol

UDP - User Datagram Protocol
UML - Unified Modelling Language

VDQM - CASTOR Volume and Drive Queue Manager
VMGR - CASTOR Volume Manager

wassh - Wide Area SSH

XMI - XML Metadata Interchange
XML - Extensible Markup Language

Bibliography

[1] Aerial view of the CERN LHC accelerator,
http://atlas.kek.jp/sub/photos/CERN/CERN-MontBlanc-letter.jpg,
December 2005

[2] German Cancio and Piotr Poznanski."Managing Computer Centre machines with Quattor",
http://quattor.org/documentation/presentations/quattor-c5-12122003.pdf,
December 2003

[3] Miroslav Siket, German Cancio, David Front, Maciej Stepniewsk. "Lemon Monitoring" , Presentation,
http://lemon.web.cern.ch/lemon/doc/presentations/lemon-bologna-2005.ppt,
May 2005

[4] T. Perelmutov, D. Petravick. "Storage Resource Manager", CHEP 04, Contribution 107,
http://indico.cern.ch/materialDisplay.py?contribId=107&sessionId=10&materialId=paper&confId=0, December 2004

[5] S. Ponce. "New stager architecture and deployment", CASTOR external operation meeting, CERN,
http://castor.web.cern.ch/castor/PRESENTATIONS/2005/External-Operation-Worshop-20050614/CASTOR2-overview-20050614.pdf, June 2005

[6] Fred Moore. "Storage Navigator",
http://www.horison.com/horison/books/2005/, 2005

[7] StreamLine SL8500 Modular Library System,
http://www.storagetek.com/upload/documents/TC0018B_SL8500_OC.pdf,
December 2005

[8] Linear Tape Open (LTO) Ultrium tape drives,
http://www.savedon.com/upload/documents/TC0021A_LTO_OC.pdf,
December 2005

[9] IBM TotalStorage 3592 Tape Drive Model J1A, *http://www.nctgmbh.de/download/3592TapeDriveModelJ1A.pdf, December 2005*

[10] Gustavo Castets, Yotta Koutsoupias, Chris McLure, Juan Felipe Vazquez. "IBM TotalStorage Enterprise Tape: A Practical Guide", *http://www.redbooks.ibm.com/redbooks/pdfs/sg244632.pdf, June 2004*

[11] Extremely Large Fabric management system (ELFms), *http://cern.ch/elfms, December 2005*

[12] Quattor, *http://quattor.org, December 2005*

[13] LHC Era Monitoring (Lemon), *http://cern.ch/lemon, December 2005*

[14] G. Cancio, T. Kleinwort, W. Tomlin, M. Siket, V. Bahyl, S. Chapeland, J. van Eldik, V. Lefebure, H. Meinhard, P. Poznanski, T. Smith, M. Stepniewski, D. Waldron, CERN, Geneva, Switzerland. D. Front, Weizmann Institute of Science, Rehovot, Israel. "Current status of fabric management at CERN", CHEP 04, Contribution 489, *http://indico.cern.ch/getFile.py/access?contribId=489&sessionId=10&resId=1&materialId=paper&confId=0, December 2004*

[15] Tobias Oetiker. "Round Robin Database tool (RRDtool)", `http://rrdtool.org`, *December 2005*

[16] GridFTP: www-unix.globus.org, *http://www-unix.globus.org/toolkit/docs/3.2/gridftp/key/index.html, December 2005*

[17] W. Allcock. "GridFTP: Protocol Extensions to FTP for the Grid", *http://www.ggf.org/documents/GWD-R/GFD-R.020.pdf, April 2003*

[18] Examples of GridFTP usage, *http://it-dep-fio-ds.web.cern.ch/it-dep-fio-ds/Documentation/gridftp-examples.asp, December 2005*

[19] www.dCache.org, *http://www.dcache.org/, December 2005*

[20] Patrick Fuhrmann. "dCache, the commodity cache", Twelfth NASA Goddard and Twenty First IEEE Conference on Mass Storage Systems and Technologies, Washington DC, *http://www.dcache.org/manuals/ieee2004.paper.pdf, March 2004*

[21] Gene Oleynik, Bonnie Alcorn, Wayne Baisley, Jon Bakken, David Berg, Eileen Berman, Chih-Hao Huang, Terry Jones, Robert D. Kennedy, Alexander Kulyavtsev, Alexander Moibenko, Timur Perelmutov, Don Petravick, Vladimir

Podstavkov, George Szmuksta, Michael Zalokar. "Fermilab's Multi-Petabyte Scalable Mass Storage System", 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005), 0-7695-2318-8/05, *http://storageconference.org/2005/papers/07_oleynikg_fermilab.pdf, April 2005*

[22] J. Bakken, E. Berman, Chi-Hao Huang, A. Moibenko, D. Petravick, M. Zalokar. "The status of the Fermilab Enstore Data Storage System", CHEP 04, Contribution 464, *http://indico.cern.ch/getFile.py/access?contribId=464&sessionId=10&resId=1&materialId=paper&confId=0, December 2004*

[23] Richard W. Watson. "High Performance Storage System Scalability: Architecture, Implementation and Experience", 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005), 0-7695-2318-8/05, *http://storageconference.org/2005/papers/13_watsonr_highperformance.pdf, April 2005*

[24] O. Bärring, B. Couturier, J.-D. Durand, S. Ponce. "CASTOR: Operational issues and new developments", CHEP 04, Contribution 230, *http://indico.cern.ch/materialDisplay.py?contribId=230&sessionId=10&materialId=paper&confId=0, December 2004*

[25] ROOT - An Object-Oriented Data Analysis Framework, *http://root.cern.ch/, December 2005*

[26] The Maui Scheduler, *http://www.nsc.liu.se/systems/cluster/grendel/maui.html, December 2005*

[27] Using Platform LSF License Scheduler, *http://www.ms.washington.edu/Docs/LSF/LSF_6.0_Manual/license_scheduler_6.0/lsf_license_scheduler.html#122068, December 2005*

[28] Umbrello UML Modeller homepage, *http://umbrello.org, December 2005*

[29] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison-Wesley, 1994*

Index