



Universitat  
Autònoma  
de Barcelona



**ENABLING DCACHE TO USE PREEXISTENT  
MASS STORAGE SYSTEMS**

Memòria del Projecte Fi de Carrera  
d'Enginyeria en Informàtica  
realitzat per  
Daniel Álvarez Gómez  
i dirigit per  
Miquel Àngel Senar Rosell  
Bellaterra, 15 de Juny de 2007



Universitat  
Autònoma  
de Barcelona



El sotasignat, Miquel Àngel Senar Rosell

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

**CERTIFICA:**

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en Daniel Álvarez Gómez

I per tal que consti firma la present.

Signat: .....

Bellaterra, 15 de Juny de 2007



El sotasignat, Manuel Delfino Reznicek,  
Director del Port d'Informació Científica (PIC)

**CERTIFICA:**

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en Daniel Álvarez Gómez

I per tal que consti firma la present.

Signat: .....

Bellaterra, 15 de Juny de 2007



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Introduction to the project . . . . .	1
1.2	PIC (Port d'Informació Científica) . . . . .	2
1.3	Magic . . . . .	3
1.4	Objectives . . . . .	3
1.5	Project planning . . . . .	4
1.6	Feasibility analysis . . . . .	5
1.6.1	Technical feasibility study . . . . .	5
1.6.2	Economic feasibility study . . . . .	6
1.6.3	Temporal planning . . . . .	7
1.6.4	State of the art . . . . .	8
1.6.5	Project feasibility . . . . .	9
1.7	Development model . . . . .	9
1.8	Report structure . . . . .	10
<b>2</b>	<b>Storage systems and associated technology</b>	<b>11</b>
2.1	Tertiary Storage Systems . . . . .	11
2.1.1	Tape libraries . . . . .	12
2.2	Hierarchical Storage Management . . . . .	12
2.3	The Grid . . . . .	13
2.4	dCache . . . . .	14
2.4.1	Architecture . . . . .	15
2.4.2	Admin node architecture . . . . .	16
2.4.3	Pool node architecture . . . . .	19
2.4.4	Interface . . . . .	20
2.4.5	Storage management . . . . .	20
2.4.6	HSM Interface . . . . .	21
2.5	Castor . . . . .	23
2.6	ViVo . . . . .	23
2.7	Ruby . . . . .	24
2.8	SQLite3 . . . . .	25

2.9	Ext2 Filesystem . . . . .	25
<b>3</b>	<b>Design and Implementation</b>	<b>27</b>
3.1	Design . . . . .	27
3.1.1	Functional requirements . . . . .	27
3.1.2	Use Cases . . . . .	28
3.1.3	Flux diagrams . . . . .	31
3.1.4	Sequence diagrams . . . . .	34
3.1.5	Database design . . . . .	36
3.2	Implementation . . . . .	37
3.2.1	dCache configuration . . . . .	37
3.2.2	dCache to HSM script . . . . .	39
3.2.3	Additional scripts . . . . .	43
<b>4</b>	<b>Production tests</b>	<b>47</b>
4.1	Introduction . . . . .	47
4.2	Setting up a production server . . . . .	47
4.3	Performance . . . . .	50
4.3.1	Disk drive pool . . . . .	50
4.3.2	Tape storage without filesystem generation . . . . .	51
4.3.3	Tape storage with filesystem generation . . . . .	52
4.3.4	Test analysis . . . . .	54
4.3.5	Magic performance . . . . .	57
<b>5</b>	<b>Conclusion</b>	<b>59</b>
5.1	Future work . . . . .	60
	<b>Bibliography</b>	<b>61</b>

# List of Tables

2.1	StorageInfo options . . . . .	22
3.1	Copy file Use Case . . . . .	29
3.2	Retrieve file Use Case . . . . .	30





# List of Figures

1.1	Temporal planning . . . . .	7
1.2	Gantt Diagram . . . . .	8
2.1	Simplified gLite architecture . . . . .	14
2.2	Simplified dCache Architecture . . . . .	16
2.3	dCache Admin node architecture . . . . .	17
2.4	dCache Pool node architecture . . . . .	19
3.1	dCache to HSM Interface - Use Cases . . . . .	28
3.2	dCache to HSM Interface - Basic design . . . . .	31
3.3	dCache to HSM Interface - Write to HSM . . . . .	32
3.4	dCache to HSM Interface - Fetch from HSM . . . . .	33
3.5	Sequence Diagram: Write File . . . . .	34
3.6	Sequence Diagram: Retrieve File . . . . .	35
3.7	Database Schema . . . . .	36
4.1	Performance comparison of different solutions . . . . .	54
4.2	Time elapsed on a file request - Small files . . . . .	55
4.3	Time elapsed on a file request - Large files . . . . .	56



# Chapter 1

## Introduction

In this chapter we briefly introduce the background and objectives of this project, describing the current situation, as well as a brief introduction to the solution to the exposed problems that provides this project.

The purpose of this project is to study the possibility of creating a hybrid between two methods of storage systems, namely hard disk drive pools and tape robots.

### 1.1 Introduction to the project

Large amounts of small files pose a performance issue when stored in Mass Storage Systems. This fact has become patent particularly in the world of physics, where large amounts of data are processed, requiring big amounts of CPU time, but also large storage systems with fair performance to store, retrieve and analyse the results of experiments.

CERN (Organization Européenne pour la Recherche Nucléaire), or European Organization for Nuclear Research, situated in northwest Geneva, comprising Switzerland and France, is actively working in improving data processing and storage capabilities.

One of the most important data processing projects is a Grid based distributed cluster, EGEE (Enabling Grids for E-sciencE), with over 30,000 nodes of computing power that hosts several physics experiments and some non-physics related projects.

PIC is an important part of this cluster, being a Tier-1 center. Tier-1 centers cache data generated at CERN (Tier-0) and several other Tier-1

centers, where Tier-2 centers can download data sets from an appropriate Tier-1 center.

It is then, necessary to find a way to properly manage this flux of data, using Hierarchical Storage Managers to move data from high cost and performance to low cost and performance storage when necessary.

The ViVo project solved some of these problems by generating ISO images containing small files, which were finally stored and retrieved from a MSS.

The retrieval process was handled externally, using ISO servers as a temporary cache, which retrieved the contents of an ISO image on demand, with a certain read-ahead and removal policy.

This project improves ViVo, by adding the functionality to an already in production storage management system, thus removing the need to use an external ISO server. The final implementation also aims at improving the performance of the system, by using read-ahead algorithms which will smartly cache likely to be accessed files in a pool disk.

## 1.2 PIC (Port d'Informació Científica)

This project has been developed at PIC (Port d'Informació Científica), a center funded by the regional and central governments, as well as the UAB and IFAE (Institut de Física d'Altes Energies). PIC supports scientific groups working in projects which require strong computing resources for the analysis of massive sets of distributed data.

The team that made this project possible included Manuel Delfino (director of PIC, and co-director of this project), Esther Acción (ViVo developer), and Adrià Casajús (dCache maintainer at PIC). Meetings took place regularly to specify requirements, check the status of the project, and solve design or development issues.

One of these projects is Magic, a physics experiment that uses multiple file sets of data, accessed in a sequential way. This experiment provided us with a good test set for our project.

IFAE has provided us with the Magic software, data sets of experiments and examples on how to run them. Our contacts at IFAE have been Abelardo Moralejo and Roger Firpo.

## 1.3 Magic

During the development of the project, we have been using the Magic project to perform tests for both the development and production phases. We chose Magic due to its file access characteristics. A Magic job usually uploads several files with different sizes, while the latter jobs access them in sequential order. Therefore, it is a perfect candidate to be used to test the performance of our project.

Magic is an imaging atmospheric Cherenkov telescope located in the Canary Island of La Palma. It observes gamma rays (electromagnetic radiation of high energy).

The cosmos and its evolution are studied using all radiation, in particular electromagnetic waves. The observable spectrum extends from radio waves (at wavelengths of several tens of meters, or energies of some .00001 eV) to ultra-high energy gamma quanta (wavelengths of picometers or energies of 100 TeV).

Magic is a ground-based telescope for the detection of very high energy (VHE) electromagnetic particles, in particular gamma rays. Having no electric charge, VHE gammas are not affected by magnetic fields, and can, therefore, act as messengers of distant cosmic events, allowing straight extrapolation to the source.

Although high-energy gamma quanta get absorbed in the atmosphere, they can be observed indirectly. The absorption process proceeds by creation of a cascade or shower of high-energy secondary particles. The Cherenkov method uses the fact that the charged secondary particles emit radiation at a characteristic angle, the Cherenkov radiation. Cherenkov photons have energies in the visible and UV range, and pass through the atmosphere; thus they can be observed on the surface of the earth by sufficiently sensitive instruments.

## 1.4 Objectives

The main objective of this project is to provide an interface between a mass storage system (based on tapes) and a higher speed pool of hard disk drives, which uses dCache, a disk storage system widely used in the particle physics community.

The objectives can be summarized as follows:

- Analyse whether this project is both technically and temporally feasible.
- Develop an interface between a MSS and a disk storage system.
- Use dCache as a hierarchical storage manager, while keeping the current features that are provided by the current dCache production server.
- Leave the possibility of using different tape manager systems, with minimal changes to the project.
- Avoid modifying dCache source code, which will pose upgrading difficulties.

A connection between dCache and the MSS has been developed, using virtual filesystems containing large amounts of small files. By using the MSS interface present in dCache, we have been able to store these filesystems into a MSS, retrieving them on demand.

Maintaining a catalog of files and filesystems allows us to fetch the precise filesystem from tape containing the requested file. Assuming that files in a same filesystem have higher probability of being, if not accessed sequentially, requested in short periods of time, we read ahead all the files from the retrieved filesystem.

Using dCache, it is possible to force the action of file caching. Thus, by forcing the caching of the rest of the files in the filesystem, we leave the removal policy to dCache, that will manage individually for each file whether it should be kept in cache.

## 1.5 Project planning

When developing a project, and in particular, a computer science project, it is necessary to establish a good project and time planning, due to the unpredictable time needed to define requirements, learn about the background of the project and the issues that we may find in the development.

In this particular project, the time needed to achieve these skills has taken a large amount of time of the project.

In this chapter we describe the temporal planning of the project, the phases in which it has been divided, as well as a feasibility analysis, taking into account the risks that may appear in the developing phase.

In order to achieve the goals of this project, the plan has been divided as follows:

- **Background:** Familiarize with the tools and server applications that will host the piece of software that is this project. Learn theoretical concepts about Hierarchical Storage Systems. Manage to install and properly configure an instance of dCache in a test server pool.
- **Requirements:** Using periodical meetings with the project director and other team members at PIC, determine the functional requirements that the final version of the application should comply.
- **Design:** Establish a design of the application, the modules that will define the application and the structure and language used.
- **Coding**
- **Tests and optimizations:** After reaching a first development version of the application, perform unity tests, checking whether all the use cases described in the requirements phase have been accomplished. Correct errors if necessary and improve features if possible.
- **Report**

## 1.6 Feasibility analysis

The feasibility analysis consists in a preliminary study undertaken to determine and report the viability of a project. Depending on the results of this study, it has to be decided whether to proceed with the project, or cancel it.

### 1.6.1 Technical feasibility study

The hardware resources needed to develop the project were as follows:

- **Workstation**
  - **CPU:** 2GHz Intel Core Duo (MacBook Pro)
  - **Memory:** 1.5GB RAM
  - **Disk:** 80GB

- **Software:** MacOS X 10.4
- **Servers:** Used two standard workstations (provided by PIC)
  - **CPU:** 2GHz Intel Pentium-4
  - **Memory:** 1GB RAM
  - **Disk:** 100GB
  - **Software:** Scientific Linux 4
- **Production Server:** Real world tests
  - **CPU:** 3.20GHz Quad Intel Xeon
  - **Memory:** 4GB RAM
  - **Disk:** 2 x 1.7TB RAID-1 disk arrays
  - **Software:** Scientific Linux 4

The software resources needed to develop the project were as follows:

- Operating System: Scientific Linux 4, MacOS X (10.4)
- dCache 1.6
- Ruby language (with Rubygems)
- SQLite3
- TextMate
- SSH Clients

### 1.6.2 Economic feasibility study

The hardware used was provided either by PIC (servers and production server) or by the project developer, and the software used had either a GPL-type license or a paid license provided by the computer manufacturer.

Additionally, a grant was provided by the Universitat Autònoma de Barcelona, the central and regional governments and the IFAE to develop this project as a degree thesis.

Analysing the cost of the requirements and developer, we concluded the project was feasible in economic terms.



### 1.6.3 Temporal planning

The project has been planned in a half year time. We consider that the project started in November 1st, 2006, and ended in June 10th, 2007, with several pauses due to exam periods and national holidays.

The final planning is shown on Figure 1.1:





















Activities							
#	Info	Title	Given Work	Flag	# Predec.	Expected Start	Actual Work
0		▼ dCache PFC				11/1/06	182.5 days
1		Viability study	10 days			11/1/06	10 days
2		▼ Theoretical background				11/1/06	56 days
3		ViVo	7 days			11/1/06	7 days
4		dCache	42.5 days		3	11/10/06	42.5 days
5		HSM	6.5 days		4	1/9/07	6.5 days
6		Concept and Design	18 days		1; 2	1/18/07	18 days
7		▼ Development			2	1/18/07	63 days
8		dCache: Installation	4 days			1/18/07	4 days
9		dCache: Configuration	11 days		8	1/24/07	11 days
10		Enabling dCache-HSM inte...	4 days		6; 9	2/13/07	4 days
11		HSM Interface development	32 days		10	2/19/07	32 days
12		Optimization	12 days		11	4/4/07	12 days
13		▼ Tests			7	4/20/07	15.5 days
14		Tests	10.5 days			4/20/07	10.5 days
15		Improvements	5 days		14	5/4/07	5 days
16		Report	20 days		13	5/11/07	20 days

Figure 1.1: Temporal planning

The following figure (1.2) represents a Gantt diagram of the temporal planning:

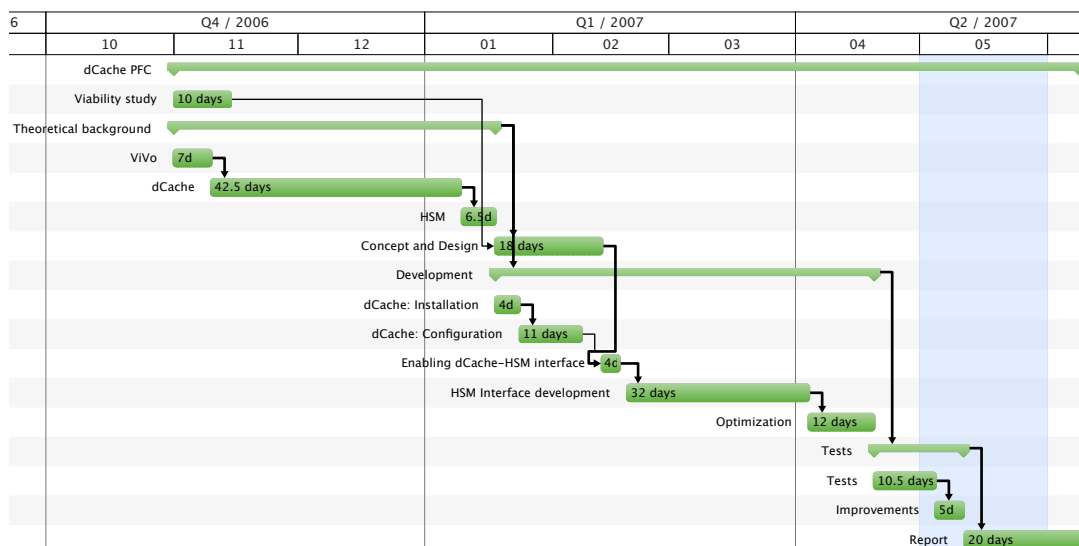


Figure 1.2: Gantt Diagram

### 1.6.4 State of the art

Currently, PIC is using ViVo to manage large amounts of small files. As ViVo is an externally handled piece of software, it does not depend on dCache. However, unlike the solution provided by this project, it needs external dedicated temporary pool servers to download and cache the ISO images. ViVo uses an automounter feature which fetches the images from the tape server when needed.

In production, this has malfunctioned several times, mainly due to the use of the automounter, and the lack of consistent cache removal and read ahead policies.

By integrating these features into dCache, we will be able to manage in a more efficient way the reading and caching of these files.

The goal of this project is to simplify the usage of the tertiary storage, using the existing dCache production server to fetch and cache the files.

### 1.6.5 Project feasibility

Analysing the feasibility of the project it can be observed that the benefits that this project will bring, particularly to the physics community, are important. Not only we will be able to improve the performance of the current solution, but also ensure the stability of the system.

Therefore, this project was found to be feasible both in a technical and economic analysis.

## 1.7 Development model

This project needed to be flexible in terms of functional requirements and planning, which were specified as the project advanced, using team meetings and brainstorming. Each phase started with a design goal and ended with the team consisting of the project director at PIC and other members reviewing and commenting the progress. Thus, the ideal software engineering model is the spiral model, because of the following reasons:

- The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the external or internal users and other aspects of the existing system.
- A preliminary design is created for the new system.
- A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product.
- A second prototype is evolved by a fourfold procedure: evaluating the first prototype in terms of its strengths, weaknesses, and risks; defining the requirements of the second prototype; planning and designing the second prototype; constructing and testing the second prototype.
- The existing prototype is evaluated in the same manner as was the previous prototype, and, if necessary, another prototype is developed from it according to the fourfold procedure outlined above.
- The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired.
- The final system is constructed, based on the refined prototype.

- The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.

## 1.8 Report structure

The report for the project has been divided in 5 main chapters:

In the first chapter, Introduction, we briefly explain the project to the reader, defining the context in which this project was developed. We also analyse whether the project is feasible both in technical and economic terms, and specify the development model and the structure of the report.

The second chapter, Storage systems and associated technology, is used to introduce the technical background needed to fully understand the objectives of this project, particularly emphasizing dCache, the main storage software in which this project is based.

The third chapter, Design and Implementation, explains, using use cases and UML diagrams, how a prototype of the project was designed. Some information about the actual implementation based in the design is also given, using fragments of code to explain how the project was developed.

In the fourth chapter, Production tests, we perform tests to the final application prototype, checking the functional requirements, and providing a quantitative and qualitative analysis of the performance of our solution, comparing it to other available alternatives.

Finally, the last chapter, Conclusion, summarizes the objectives of the project and checks whether all the objectives were successfully accomplished, and provides closure for the project. Additionally, a possible future work line is also introduced.

## Chapter 2

# Storage systems and associated technology

In this chapter, we describe the background needed to understand the contents of the project. Some technical concepts are briefly introduced, and the software used to develop this project is concisely explained.

### 2.1 Tertiary Storage Systems

Tertiary Storage Systems are usually lower cost storage systems than Secondary Storage Systems (hard disk drives), at the expense of having lower read and write performance. Some examples of Tertiary Storage Systems are optical disks or magnetic tape drives, of which the latter is the most common solution.

While it would be ideal to have all data available on high-speed devices all the time, this is prohibitively expensive when handling large amounts of data. Robot-based tape systems provide a much lower cost per byte storage than high-speed storage devices such as hard disk drive arrays.

Tertiary Storage Systems perform better when fetching files in consecutive order, due to the sequentiality of the reading process. It is the responsibility of the storage system to optimize access to the system, storing likely sequentially accessed files together, or implementing read-ahead techniques.

### 2.1.1 Tape libraries

In order to store and organise data among several hundreds or thousands of tapes, a tape library (or tape silo) is used, where tapes are stored in slots, identified with barcodes.

A number of automated robot heads with a barcode scanner travel through the tape silo when a certain data is requested or has to be saved.

These devices can store immense amounts of data, currently ranging from several terabytes up to petabytes of data, or about ten thousand times the capacity of a typical hard drive.

CERN announced that using 45 newly installed StorageTek tape drives, capable of writing to tape at 30 megabytes/s, were able to achieve storage-to-tape rates of 1.1 gigabyte/s for periods of several hours, with peaks of 1.2 gigabyte/s. The average sustained over a three day period was of 920 megabytes/s[6].

Currently, PIC owns two tape libraries, a L5500 StorageTek tape library with capacity for 6000 x 200 GB tapes, roughly 1200TB, or 1PB, and an IBM 3584 L52 tape library, initially configured to hold 727 LTO-3 tapes (400 GB per tape) and 4 tape drives, with the possibility of growing to over 4 times this capacity.

## 2.2 Hierarchical Storage Management

Hierarchical Storage Management (HSM) is a technique used to accomplish a compromise between cost and speed, by using both relatively small capacity high-cost systems and large capacity low-cost storage media.

HSM systems, then, use the low performance systems to store the data, using the high-speed devices as a sort of cache, copying data from the tape server to the array of disks when needed, and removing it when it is no longer essential.

Moreover, most HSM systems may implement read-ahead techniques, predicting, depending on the user's actions, which files are most likely to be accessed in the future. These techniques speed up considerably the overall performance of the HSM system.

## 2.3 The Grid

The Grid, as it is usually identified the EGEE project, takes its inspiration from power grids. Power grids enable quick and easy access to power, where users do not have to take into consideration where the power comes from and how it actually gets to the outlet.

Enabling Grids for E-science (EGEE) is project funded by the European Commission's Sixth Framework Programme through Directorate F: Emerging Technologies and Infrastructures, of the Directorate-General for Information Society and Media. It connects more than 70 institutions in 27 European countries, and more than 100 sites in 31 countries all over the world, to construct a multi-science Grid infrastructure for the European Research Area, sustaining more than 30000 jobs a day (over a million per month)[7].

The EGEE project aims to provide computing resources for analysis of data coming from the forthcoming Large Hadron Collider (LHC) at CERN. This project connects High Energy Physics computing resources from across the globe, and is required to process the predicted 15 petabytes of data the LHC will produce each year. EGEE started from this infrastructure, adding more resources from all parts of the globe and attracting users from a number of other communities to form what has become the largest multi-science Grid infrastructure in the world.[8]

The EGEE project uses the gLite middleware, a software installed in the nodes that compose the Grid, allowing to interconnect all these computing elements distributed through the globe.

The infrastructure of EGEE is large and complex and it is beyond the scope of this report to describe the entire system in detail. We will therefore only briefly describe the architecture and explain the links between The Grid and our project.

As shown in Figure 2.1, when users submit their jobs to The Grid, these are submitted to a so called Resource Broker, where they are queued and later, processed by a Workload Manager and passed through to Condor, where are scheduled and later sent to a Computing Element.

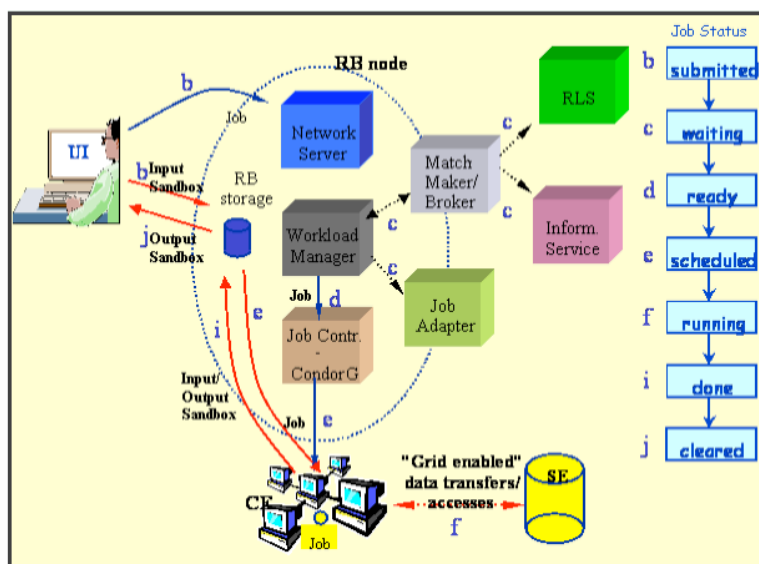


Figure 2.1: Simplified gLite architecture

The Computing Element can be, for instance, PIC, where another scheduler, Torque/PBS, will take the job, and will be later sent to the final Worker Node of the cluster, where the job will be actually processed.

While the actual job would have arrived to its final destination to be processed, the data associated to it must be reached somewhere. An additional software is needed to access to the data associated to the job.

That software could be, for instance, dCache. The job can use therefore dCache's API to access the data stored in one of the dCache pools.

## 2.4 dCache

dCache is a storage system developed by the Deutsches Elektronen-Synchotron (DESY) in Hamburg, and the Fermi National Accelerator Laboratory (Fermilab) in Chicago, widely used by the high energy physics community, with the objective of being useful to manage the data that will be generated when the LHC (Large Hadron Collider) begins operation in early 2008.



It provides a system for storing and retrieving huge amounts of data, which is distributed among several machines, regardless of location, type, or size of the nodes. It provides an interface to manage stored files, using NFS to show a single filesystem tree, to exchange data with the storage system, which provides transparent access to the end user.

The system is able to manage heterogeneous nodes, create and set the behaviour of pools and secure the data, being able to recover from disk or node failures.

Using the interface provided, we are able to connect a tertiary storage system (i.e. tape server), where safe copies of the files will be stored, removing them from cache when are no longer being used, thus freeing the cache, allowing more files to be stored.

### 2.4.1 Architecture

dCache is based in a admin node - pool node structure. The admin node has track of the pool nodes associated to its domain, and provides an interface to access the filesystem.

Pool nodes contain disk drive arrays, where data will be stored. The admin node pushes the data sent by a user from the interface into one of the pool nodes, using heuristics which will analyse user requirements and pool statistics, such as the amount of free space available.

These pool nodes may be heterogeneous, not only in storage size, but in architecture, or services associated to it. For instance, a certain pool node may have an interface to an external tertiary storage system. The admin node, can be then, configured, to send certain files to this pool, and force them to be stored in a tape server.

A simplified dCache architecture may be summarized in Figure 2.2:

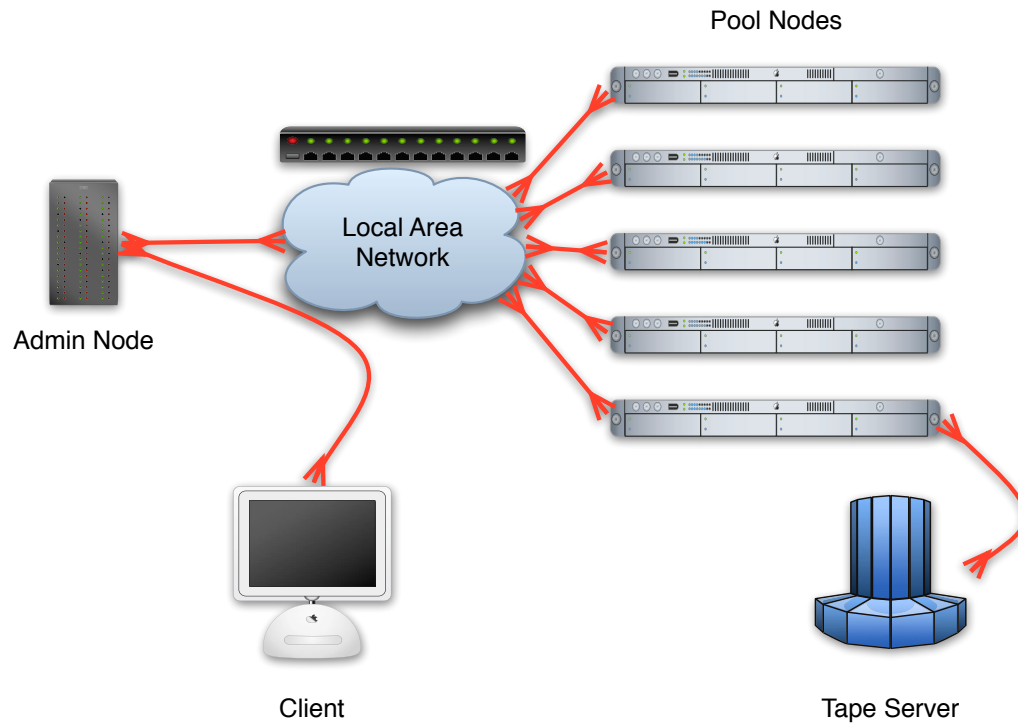


Figure 2.2: Simplified dCache Architecture

The dCache Admin node provides the client with a certain interface, by exporting a POSIX-like virtual filesystem where a user (provided with certain tools) can interact with stored files.

### 2.4.2 Admin node architecture

The admin node consists of several modules, each providing different features. The modules comprised in dCache are shown in Figure 2.3.

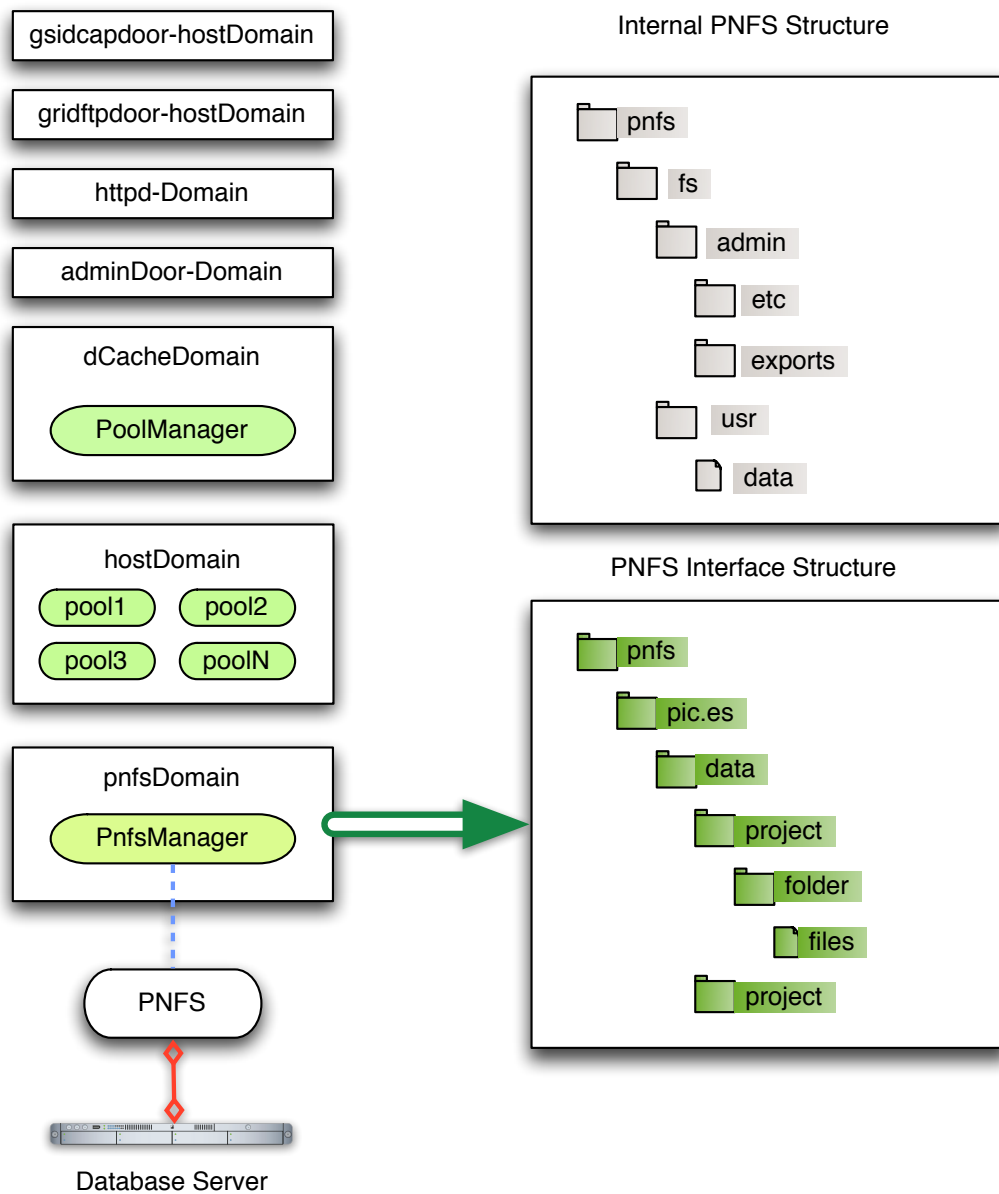


Figure 2.3: dCache Admin node architecture

The main module is the Domain module, which handles the configured pools, providing access to these pools via the Local Area Network.

The door domains provide access by other means to the dCache Admin Server. dCache provides implementations for both dcap and GridFTP protocols. These protocols are used by external machines to access the files contained in the dCache server.

While `httpdDomain` provides a small web server that can be used to check server statistics, `adminDoorDomain` provides a SSH version 1 compliant admin terminal. From the terminal, one can modify server settings or force a certain task to be performed.

The `adminDoor` is virtually the only way to interact with the dCache server. Several Graphical User Interface (GUI) tools have been developed to ease the access to this admin tool.

One of the most important modules is the `pnfsDomain` module, which provides a NFS-based interface allowing read/write access to the filesystem.

The only exported to PNFS directory is the one that contains the data, although internally, the PNFS filesystem also contains a directory `admin/` where several server configuration files are kept. This hidden directory also contains a subdirectory named `exports/`, where, like a standard NFS server, there is an entry for each IP range allowed to export the PNFS filesystem.

Besides the PNFS interface, the dCache server needs a PostgreSQL database where information about files and pools will be stored.

### 2.4.3 Pool node architecture

A typical pool node architecture is shown in Figure 2.4:

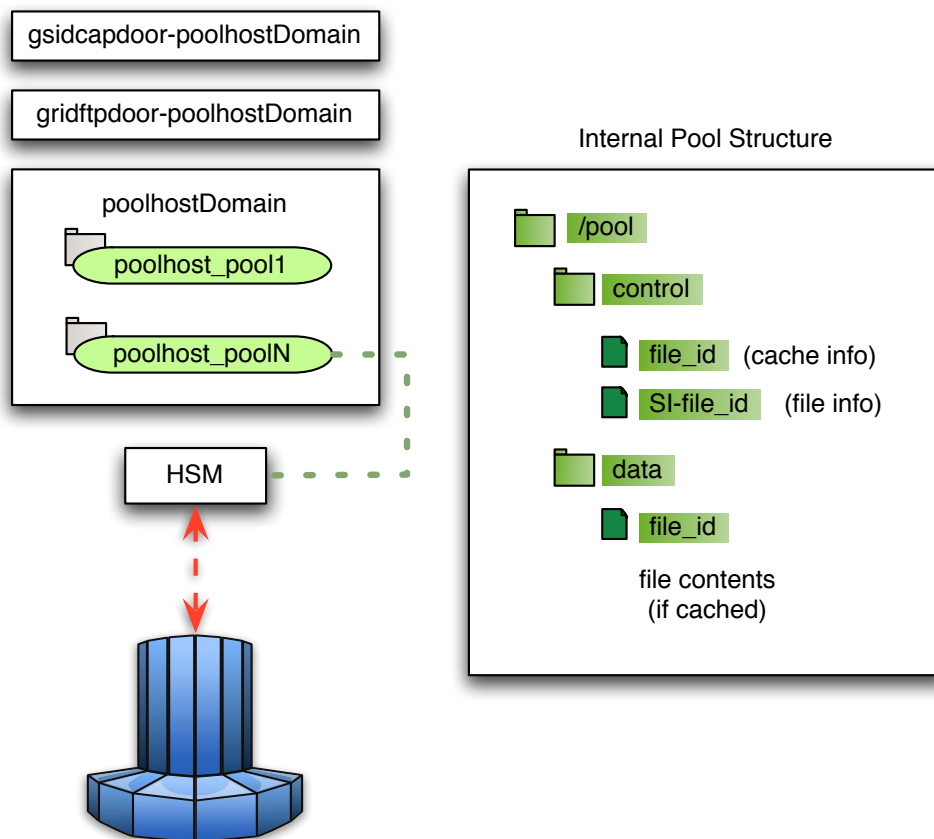


Figure 2.4: dCache Pool node architecture

dCap and GridFTP protocols are provided for node access as well. The poolhostDomain contains the enabled pools in the node. Several pools may be created on a same node to accommodate several pool configurations. Pools can be linked to storage system external interfaces to allow, for instance, connection to a tape storage server.

In the diagram, two pools are shown. The first one (poolhost\_pool1) is a simple pool, where the files are stored at local disks. The second pool (poolhost\_poolN) has an interface to an external storage system. In order to allow a pool to store its files into an external MSS, HSM must be enabled in

the dCache Admin node, and the pool must be configured as a HSM-enabled pool.

#### 2.4.4 Interface

The main user-end interface to dCache is the PNFS mount point, where the read/write commands will be issued. While the PNFS structure is pretty much POSIX-compliant (i.e. listing/creating directories, reading file attributes or User-Group-Other/Read-Write-Execute based permissions), it needs specific file access tools, for both reading and writing, such as dCap.

PNFS is a filesystem not designed for storage of actual files. Instead, PNFS manages the filesystem hierarchy and standard meta-data of a UNIX filesystem, keeping the complete information in a database.

An NFS server is implemented in PNFS. All the meta-data can be accessed with a standard NFS client. While normal filesystem operations work fine, IO operations on the actual files in the pnfs will normally result in an error.

The pnfs filesystem needs to be mounted only by the server running the dCache core services. In addition, it may also be mounted by clients. The command to mount the filesystem may be as follows:

```
# mount -o intr,hard,rw dcdb01.pic.es:/pnfs /pnfs/pic.es
```

Assuming the host mounting the file system is granted access in the exports/ directory, users may then access the meta-data with regular filesystem operations, and the files themselves may then be accessed with the dCap protocol.

#### 2.4.5 Storage management

dCache saves the files internally as a 12 byte long unique ID, called pnfsID. A pnfsID used for a file will never be reused, even if the file is deleted. There are tools provided by dCache to know the path and filename of a file from a pnfsID, and the pnfsID from a file.

For instance, if we wanted to know the name of a certain pnfsID, located in /pnfs/pic.es/MagicTests/2007\_04\_25/CCDAQCheck

```
$ cd /pnfs/pic.es/MagicTests/2007_04_25/CCDAQCheck/  
$ cat `.(nameof)(0001000000000000000019280)`
```

```
MuxDataCheck_2007_04_25.list.errlog
```

Alternatively, the user-friendlier pathfinder tool can be used to retrieve the filename and full path of a pnfsID:

```
$ /opt/pnfs/tools/pathfinder 000100000000000000019280
000100000000000000019280 MuxDataCheck_2007_04_25.list
0001000000000000000183A0 CCDAQCheck
000100000000000000018370 2007_04_25
000100000000000000015150 MagicTests
00010000000000000001060 data
00000000000000000001080 usr
00000000000000000001040 fs
00000000000000000001020 root
00000000000000000001000 -
0000000000000000000100 -
00000000000000000000000 -
/root/fs/usr/data/MagicTests/2007_04_25/CCDAQCheck/MuxDataCheck_2007_04_25.list
```

dCache has several stage levels for the uploaded files. When a file first arrives to dCache, it has the state precious, which means the file is not yet in safe storage and can not be removed from dCache, even if disk space is needed. The next obvious stage will be cached, set when a file has been copied to secure storage, and can be removed from cache if required.

### 2.4.6 HSM Interface

The dCache Admin server must be configured to allow data to be exchanged with an HSM, and has to define the external method to be called when a file has to be fetched or stored from an HSM.

Additionally, a specific HSM-enabled pool must be defined, and groups associated to the pool must be created. In dCache, creating a group is equivalent to create a directory in the parent dCache path, and associate it to a database group using the Admin interface.

The stager (i.e: the dCache-HSM connection) must be enabled in the PoolManager domain in the dCache Admin interface, using the following command:

```
rc set stage on
```

Additionally, using the dCache Admin interface as well, the HSM external interface has to be defined in the pool that will be staging files:

```
hsm set hsm -command=/opt/d-cache/jobs/hsmcp.sh
```

This external script has to implement put (store data into the HSM) and get (get data from the HSM) methods, and is called as follows:

```
<script> put|get <pnfsid> <localFileName> -si=<storageInfo> [more options]
```

The HSM interface script will receive all the needed information to store and subsequently fetch the file. The information contained in the storageInfo is shown in Table 2.1

Key	Use
size	Size of the file in bytes
new	False if file is already on dCache
stored	True if file is already stored in the HSM
sClass	Group associated to the file. Used by dCache to pin the file to a specific pool.
hsm	Name of the HSM
alloc-size	Size that has to be allocated for the file. May be different than filesize.
path	Original (PNFS) path of the file.

Table 2.1: StorageInfo options

Additionally, user defined options may be passed using the -X option, appending the key to be passed and its value. For instance, if we wanted to pass the key filesystem with a value of 42, we could add the following to the dccp command:

```
-X -filesystem=42
```

These options proved to be very useful in the development of the project, as they are passed to the HSM script. Later on, we were able to know where to fetch the file and where to post it, using the information previously saved.



## 2.5 Castor

CASTOR is the acronym for CERN Advanced STORage manager, a hierarchical storage management (HSM) system developed at CERN used to store physics production files and user files, with currently about 60 million files and 7 petabytes of data. Files can be stored, listed, retrieved and accessed in CASTOR using command line tools or applications built on top of the different data transfer protocols like RFIO (Remote File IO), ROOT libraries, GridFTP and XROOTD[3].

CASTOR provides a UNIX-like directory hierarchy of file names. In our case, the directories are always rooted in /castor/pic.es (with the pic.es domain being different for other centers).

For our project, we will be using Castor1, although the second version of Castor is being developed. While Castor1 is currently the production version used at PIC, it is unlikely that PIC will be using Castor2, and a switch to other storage managers, such as Enstore, is being discussed. This explains the importance that Castor must be easily replaced by any other storage manager with minimal changes.

The CASTOR name space can be viewed and manipulated only through CASTOR client commands and library calls, and will not work with regular OS commands. The CASTOR name space holds permanent tape residence of the CASTOR files, while the more volatile disk residence is only known to the stager, which is the disk cache management component in CASTOR. When accessing or modifying a CASTOR file, one must therefore always use a stager.

We will be using CASTOR as our tape server where files and filesystems will be stored, even though the project should be able to use any other HSM with minimal changes.

## 2.6 ViVo

ViVo (Virtual Volumes) is a project developed by PIC, which uses ISO images to store large amounts of small files. Storing sequentially accessed files in a single ISO image, improves the performance of the tape server, which is heavily undermined when small files are directly requested to the Tertiary Storage System.

ViVo is currently being used by the Magic project. The raw data retrieved

from the telescope is structured in folders named using the date when the files were created. These data sets are stored in ISO9660 volumes created in a daily basis, using cron (the UNIX time-based scheduling service) scripts, and stored in Castor.

Likewise, the so-called star data (output of processed RAW files) is organized in a similar way, using timestamps, with the difference that volumes are generated and stored in Castor monthly.

When a file is requested, using a special virtual path, ViVo fires an auto-mounter, which retrieves the corresponding ISO image from the tape server, implicitly indicated in the path of the file, caching the rest of the files contained in the image, which are likely to be accessed sequentially, in a cache pool. The cached files are automatically removed when disk space in cache is needed or a timeout occurs.

## 2.7 Ruby

Ruby is an interpreted, dynamically typed and object-oriented script programming language written in C. It has an open source license and was created by Yukihiro Matsumoto in 1995, designed for programmer productivity and fun, following the principles of good user interface design.

Its syntax is inspired by Perl with Smalltalk like object-oriented features, and also shares some features with Python and Lisp. It has native exception handling, iterators, regular expressions, an automatic garbage collector, it is highly portable and has a lot of user based support.

It was recently brought to a wider audience with the introduction of Ruby on Rails in 2004, a highly productive database-driven web development framework based in the Model-View-Controller design pattern.

We chose Ruby for its high productivity, and the familiarity and interest of the main developer for this programming language. The fact that this project is based in a series of external small scripts means that the scripts could have been written in any other scripting language, such as Python or Perl.

## 2.8 SQLite3

SQLite is an ACID-compliant (Atomicity-Consistency-Isolation-Durability) relational database management system contained in a relatively small C library, and has bindings for a large number of programming languages, including C, Python and Ruby.

Unlike client-server database management systems, the SQLite engine is not a standalone process with which the program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the program.

The program uses the SQLite functionality through simple function calls. This reduces latency in database access because function calls are more efficient than inter-process communication. The entire database (definitions, tables, indices, and the data itself) is stored as a single cross-platform file.

Several processes or threads may access the same database without problems, with several read accesses being satisfied in parallel. A write access can only be satisfied if no other accesses are currently being serviced.

We decided to choose SQLite for our small file information database because of its simplicity. By keeping the database in a file instead of in memory, we were able to perform tests easily. The fact that the connector in Ruby can be replaced by the one used to connect to PostgreSQL databases with minimal changes if necessary was also an important reason.

## 2.9 Ext2 Filesystem

The ext2 (or extended file system version 2) is a file system designed for the Linux kernel, as a replacement of the old ext filesystem. The space in ext2 is split up in blocks, and organized into block groups, designed to reduce internal fragmentation and minimize the number of disk seeks when reading a large amount of consecutive data.

It has been recently superseded by ext3 as the default filesystem, a journalled filesystem based and compatible with ext2.

Implementations of the ext2 filesystem exist in several other operating systems, such as in some BSD kernels, and third-party support for MacOS and Windows systems.

We have chosen ext2 as the filesystem for our virtual images because of the following reasons:

- It is a mature filesystem widely tested and supported.
- Due to the nature of our virtual filesystems, we do not need a journalled filesystem, because there is no way the filesystem could become corrupted, never being modified, except for the time it is created. A journalled filesystem would pose an unneeded space overhead.
- File system permissions are preserved.
- With a maximum file size of 2TiB, and a maximum filesystem size of 32TiB (using 4KiB blocks), it is more than enough for our purposes.

# Chapter 3

## Design and Implementation

In this chapter, we explain the design and implementation of the dCache to HSM interface, the piece of software that will allow dCache to fetch and store files from an HSM. Moreover, we briefly describe the needed dCache configuration to make the interface to the HSM work.

### 3.1 Design

In this design section, we describe the functional requirements of our project, providing UML diagrams in order to explain the structure and flux of the application.

#### 3.1.1 Functional requirements

The piece of software to develop must comply with the following requirements:

- **Implement write to HSM:** The script will be called when a user uploads a file into a group associated to our HSM pool. It must be able to store the file to an HSM, and save the information needed to retrieve it in the future.
- **Implement fetch from HSM:** The script will be called when a user requests a file from a group associated to our HSM pool, and dCache does not have a cached copy of the file.

- **Improve reading performance:** This project aims to provide mass storage services to dCache without degrading the overall performance, by implementing read-ahead techniques allowing to minimize whenever possible accesses to the tape.
- **Support several HSM's:** Provide a way to use several HSM systems with minimal changes to the script.

### 3.1.2 Use Cases

Using the functional requirements, and the dCache architecture, there are two basic use cases in the application, shown in Figure 3.1.

A detailed description of the use cases Copy file and Retrieve file is shown in Tables 3.1 and 3.2, respectively.

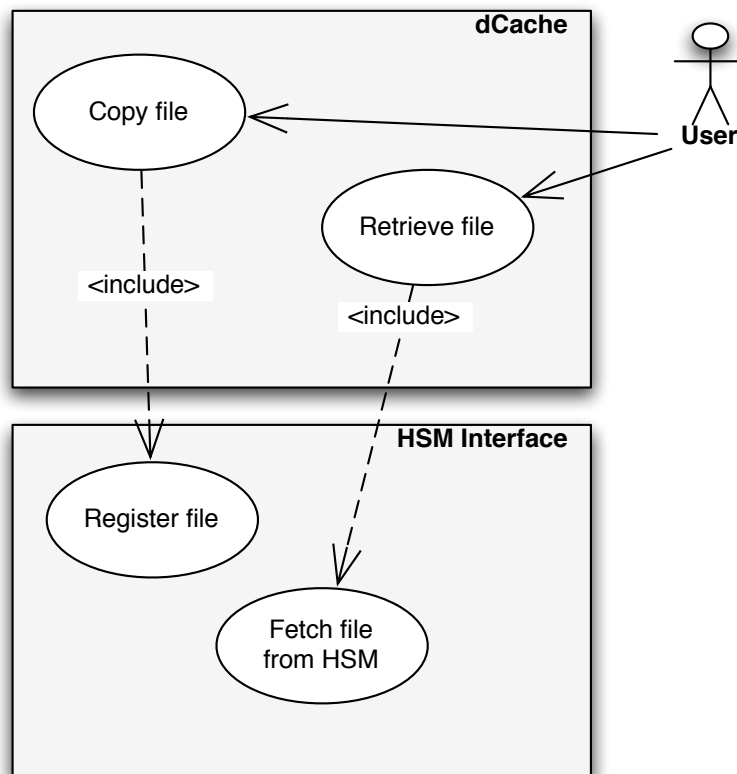


Figure 3.1: dCache to HSM Interface - Use Cases

## Copy File Use Case

Use Case	Copy File
Description	<p>The Copy File Use Case has to be able to copy a file to dCache, using any of the available tools for PNFS, storing it into an HSM-enabled pool.</p> <p>The HSM pool must be able to store temporarily the received files while a certain end of transfer special file is uploaded.</p> <p>Finally, it must store all the files into a virtual filesystem, and upload them to a Tertiary Storage System.</p>
Actors	Project user
Assumptions	<ul style="list-style-type: none"> <li>- All files that are to be stored in the same virtual filesystem must be sent with a specific unique key value (for each filesystem), to be identified in the process and later on. Tools are provided to the user to ease the upload of these files.</li> <li>- It is assumed that in a reasonable time window, the end of transfer special file will be sent. Files will not be sent to secure (tape) storage until the user explicitly requests it by sending this special file.</li> </ul>
Non-Functional	<p><b>Reliability:</b> The HSM Interface must not in any case lose any uploaded files either when waiting for end of transfer or creating the filesystem.</p> <p><b>Atomicity:</b> Each group of files that composes a filesystem to be uploaded to a tape server must be treated as a unique entity.</p> <p><b>Scalability:</b> The HSM Interface must work regardless of how many requests are received concurrently.</p>

Table 3.1: Copy file Use Case

**Retrieve File Use Case**

Use Case	Retrieve File
Description	<p>The Retrieve File Use Case has to be able to retrieve a file from dCache, using any of the available tools for PNFS, checking whether the file is in cache or in the HSM.</p> <p>If the file is stored in an HSM, the application must retrieve the entire filesystem where it is stored, using the filesystems database, and depending on configuration, copy all the files to cache (read-ahead) or just the one that has been requested into the pool filesystem.</p>
Actors	Project user
Assumptions	- Usually, the user will request more files contained in the same filesystem afterwards. By caching ahead those files, we minimize the access to tape, therefore increasing performance.
Non-Functional	<p><b>Performance:</b> The application must be working with performance in mind. While usually the performance improves when caching all files, sometimes just copying the single file to cache results in better performance. Choosing whether to cache all files or just the one requested may vary depending on projects.</p> <p><b>Scalability:</b> The HSM Interface must work regardless of how many requests are received concurrently.</p>

Table 3.2: Retrieve file Use Case



### 3.1.3 Flux diagrams

#### General flux diagram

This flux diagram represents the structure of the dCache to HSM interface (Figure 3.2).

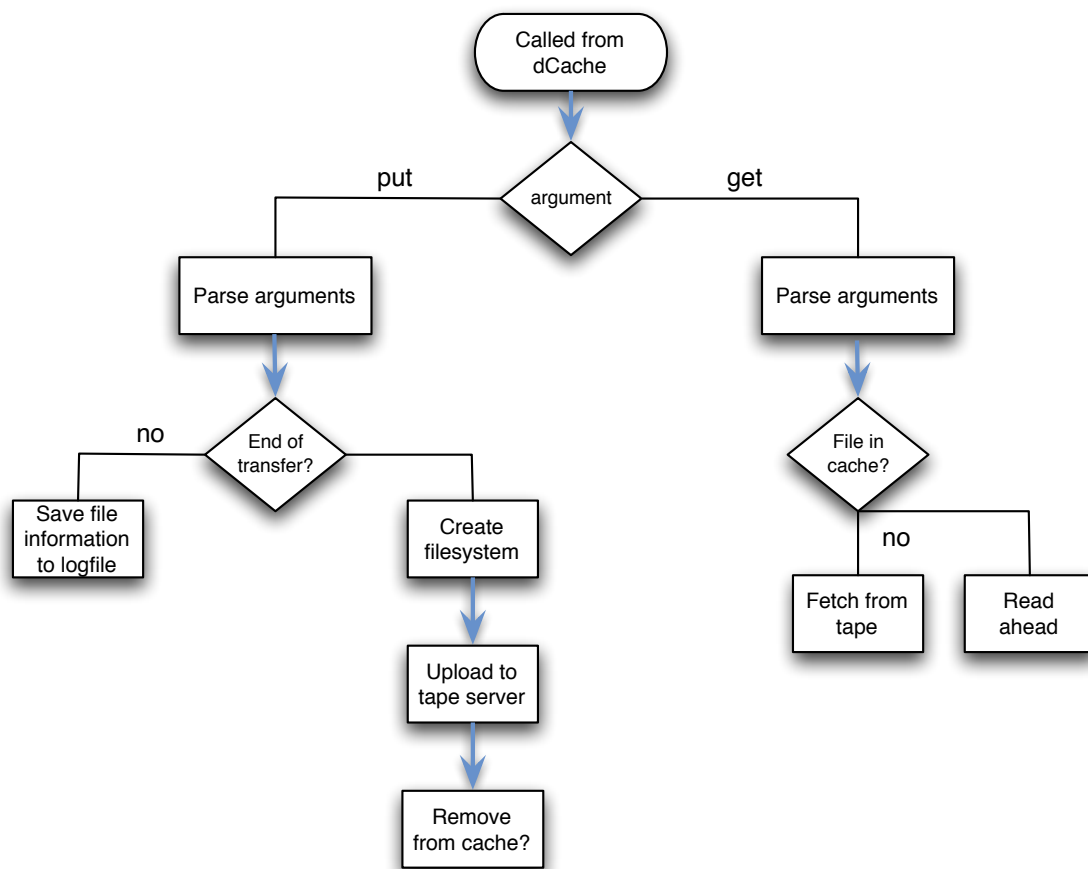


Figure 3.2: dCache to HSM Interface - Basic design

### Write to HSM diagram

This diagram represents the flux of operations performed when a put command is issued from dCache to the script (Figure 3.3).

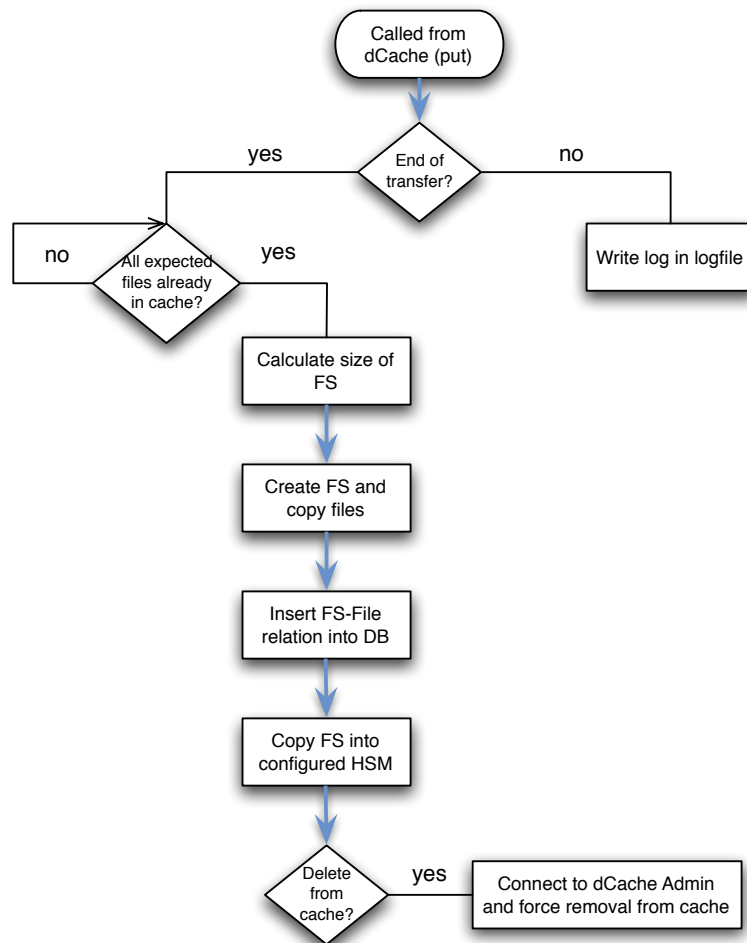


Figure 3.3: dCache to HSM Interface - Write to HSM

### Fetch from HSM diagram

This diagram represents the flux of operations performed when a get command is issued from dCache to the script (Figure 3.4).

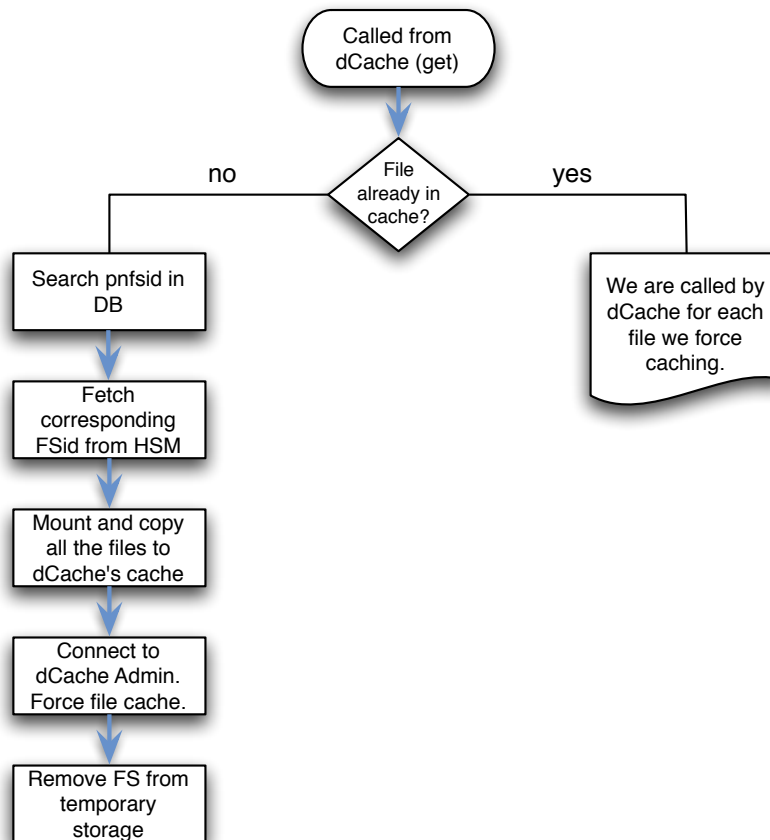


Figure 3.4: dCache to HSM Interface - Fetch from HSM

### 3.1.4 Sequence diagrams

UML sequence diagrams model the flow of logic within a system in a visual manner, enabling both to document and validate the logic, and are commonly used for both analysis and design purposes. Sequence diagrams focus on identifying the behavior within the system.

These sequence diagrams represent usage scenarios, which explain the logic described by the basic course of an action. We have two main usage scenarios in the project, the copy file and the write file use cases.

#### Write to HSM diagram

This diagram represents the sequence of actions performed when a user uploads a file to dCache (Figure 3.5).

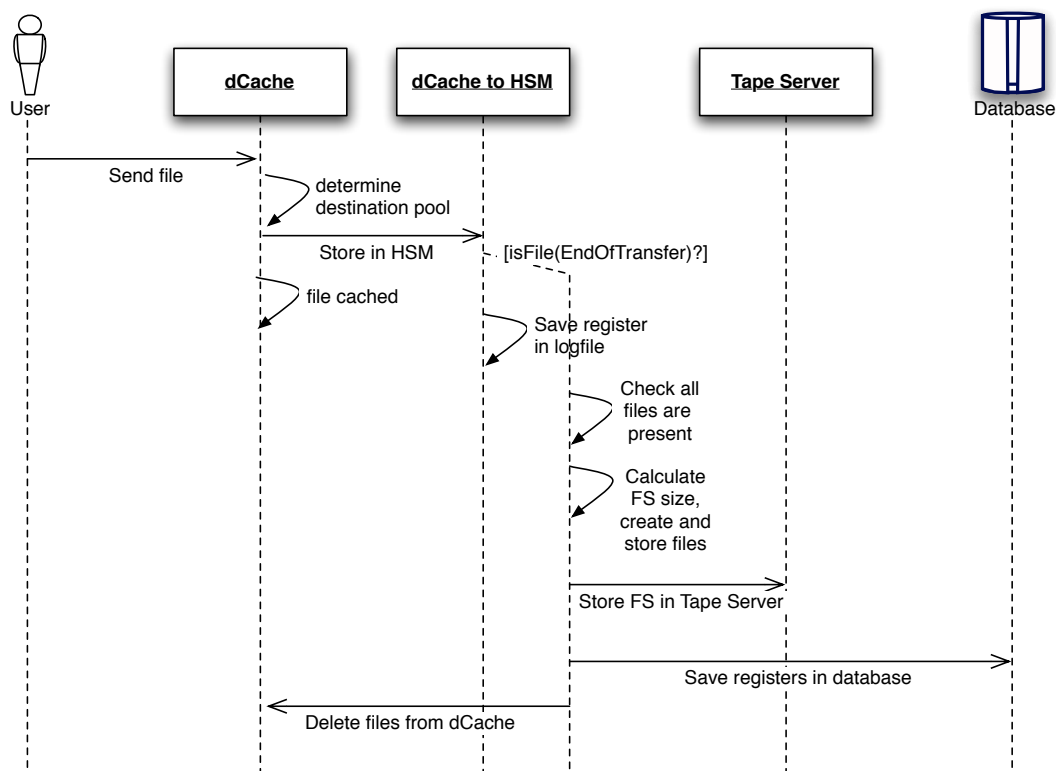


Figure 3.5: Sequence Diagram: Write File

### Retrieve from HSM diagram

This diagram represents the sequence of actions performed when a user requests a file from dCache (Figure 3.6).

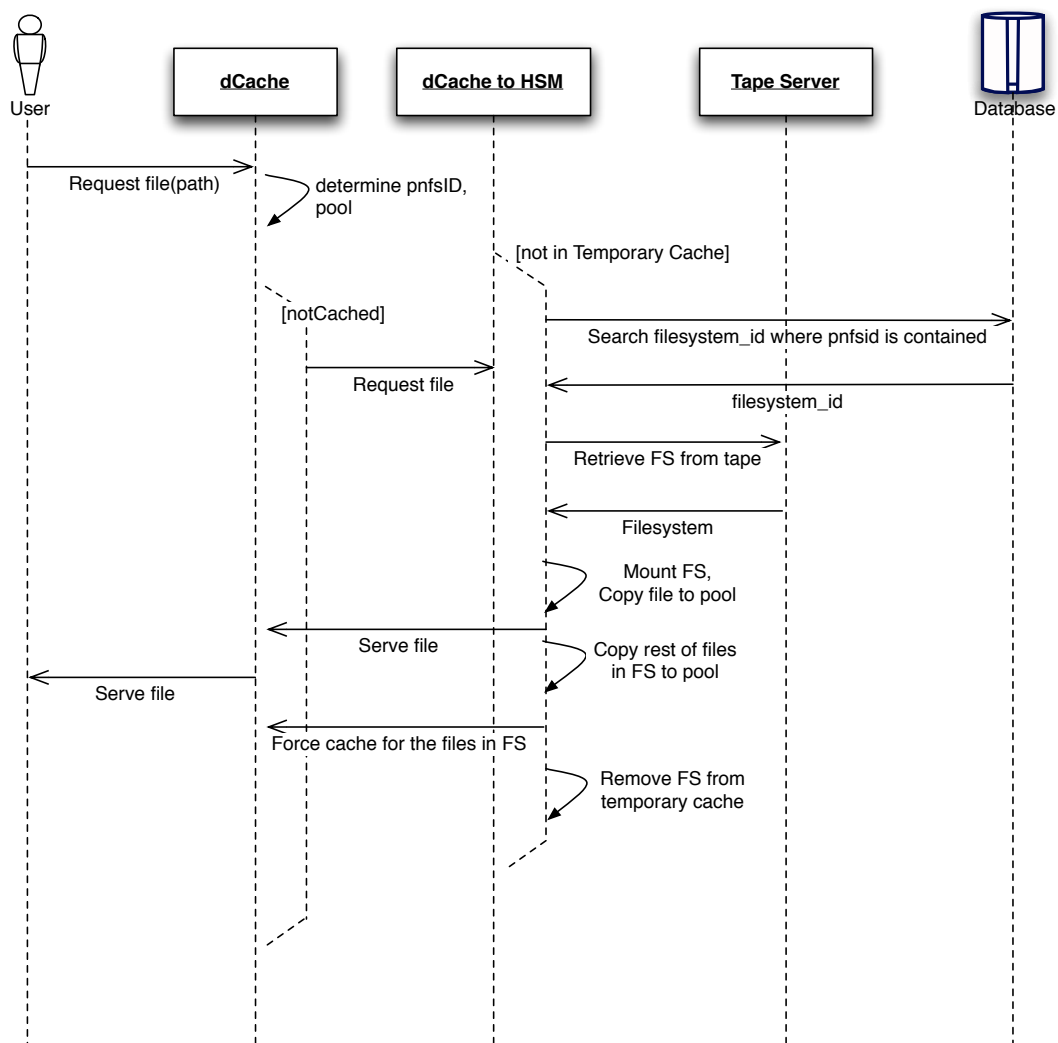


Figure 3.6: Sequence Diagram: Retrieve File

### 3.1.5 Database design

Even though the structure and relations of the database are very simple, we have decided to use a database instead of a plain file for scalability reasons.

The database is used to save the correspondences between the files stored and the filesystems where they are contained. The database schema is shown in figure 3.7.

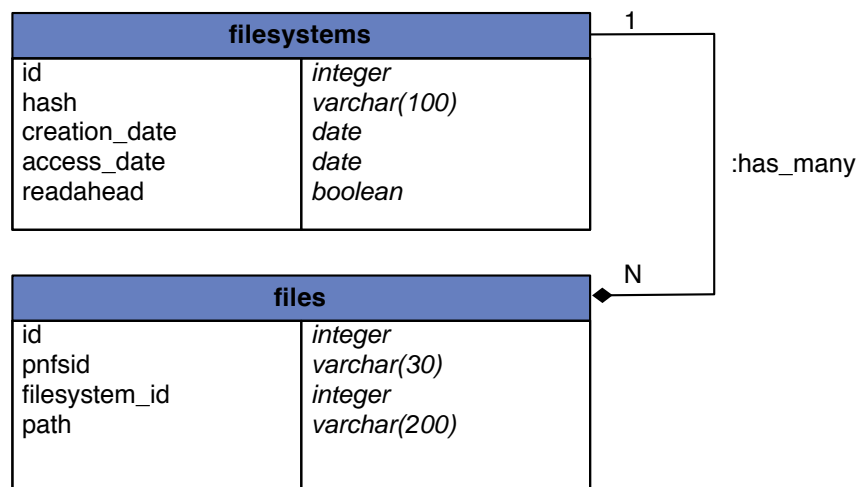


Figure 3.7: Database Schema

## 3.2 Implementation

The main scripts are written in Ruby language, using an SQLite3 database to manage file information. Some needed additional scripts are written in Expect and Python.

### 3.2.1 dCache configuration

dCache had to be properly configured to allow access to an HSM. For this matter, we set up two servers, as described in the Technical Viability chapter, one being the dCache Admin Server, and the other one being the HSM-enabled pool.

#### dCache Admin Server

The dCache Admin needs an instance of PostgreSQL to store its information. In the stable version of dCache (1.6), the following must be added to the `postgresql.conf` configuration file:

```
add_missing_from = on
```

A main database user for dCache needs to be created. We used the default user name, `pnfsserver`. Under this user, the `admin` and `data1` databases will be created. These databases contain the information about the namespace of the `pnfs` filesystem.

```
# createuser -U postgres -no-superuser -no-createrole -createdb  
-pwprompt pnfsserver
```

Another user, being `srmdcache` its default name, needs to be created. Under this user, databases containing information about the pools will be created.

```
# createuser -U postgres -no-superuser -no-createrole -createdb  
-pwprompt srmdcache
```

We need to create manually the companion database, which will store information about the pools where the files are located. The database can be populated using the script provided by dCache:

```
# createdb -U srmdcache companion
```

```
# psql -U srmdcache companion -f /opt/d-cache/etc/psql_install_companion.sql
```

We are not using the `replicas` feature (which will replicate files through

several pools), so we skipped the creation of its database.

The PNFS service was configured as follows, using the `/opt/pnfs/etc/pnfs_config` configuration file:

```
PNFS_INSTALL_DIR = /opt/pnfs
PNFS_ROOT = /pnfs
PNFS_DB = /opt/pnfsdb
PNFS_LOG = /var/log
PNFS_OVERWRITE = yes
```

The PNFS server may now be installed as follows:

```
# /opt/pnfs/install/pnfs-install.sh
```

The main dCache configuration file resides in `/opt/d-cache/config/dCacheSetup`. The main parameter that has to be specified is the service locator host (the dCache Admin Server):

```
serviceLocatorHost=wl-daniel.pic.es serviceLocatorPort=11111
```

The location to the Java binary must be specified. Due to dCache using Serializable objects that are stored into the database, all the dCache instances (server and pools) must be using the same Java version.

```
java="/opt/jdk1.5.0_10/bin/java "
```

The file `/opt/d-cache/etc/node_config` contains information about the node. The important configuration variables for the dCache Admin follow:

```
NODE_TYPE=admin
SERVER_ID=wl-daniel.pic.es
ADMIN_NODE=wl-daniel.pic.es
pnfsManager=yes
```

The dCache Admin can now be installed using the following command:

```
# /opt/d-cache/install/install.sh
```

## dCache Pool

The main dCache pool configuration file in `/opt/d-cache/config/dCacheSetup` has to be configured in a similar way to the dCache Admin.

The node config file resides in `/opt/d-cache/config/node_config`. The



variables that need to be set are the following:

```
NODE_TYPE=pool
SERVER_ID=wl-daniel2.pic.es
ADMIN_NODE=wl-daniel.pic.es
```

The pools assigned to a pool server are specified in `/opt/d-cache/etc/pool_path` using the following syntax: `<poolDataDirectory> <poolSizeInGB> <rein-stallPoolYesNo>`

We used the following configuration for our pool server, creating 2 20GB pools:

```
/home/pool 20 yes
/home/pool2 20 yes
```

### dCache-HSM Interface

Using the dCache Admin interface, we set up the HSM Interface.

First of all, we needed to disable the `lfs precious` setting to allow dCache to use an HSM as the place where precious files will be stored. To do so, we had to remove the `lfs=precious` line from both the dCache Admin and pool servers, located in the following files:

```
/opt/d-cache/config/pool.batch
/opt/d-cache/config/wl-daniel.poollist
/opt/d-cache/config/wl-daniel2.poollist
```

The next step is to enable the HSM stager, issuing the following command under the PoolManager domain:

```
rc set stage on
```

Finally, we must configure the interface in the HSM-enabled pools. To do so, we issued the following command on both pools (`wl-daniel2_1` and `wl-daniel2_2`) under the `wl-daniel2` Domain:

```
hsm set osm -command=/opt/d-cache/jobs/dcache-hsm.rb
```

#### 3.2.2 dCache to HSM script

The HSM interface script must implement both `get` and `put`, being called in the following way:

```
<binary> put|get <pnfsid> <localFileName> -si=<storageInfo> [more options]
```

Additionally, we use the options feature to pass a unique hash that will identify files that belong to a same filesystem. The MD5 hash function used is generated using the seconds and microseconds of the system clock:

```
md5 = Digest::MD5::new
now = Time::now
md5.update(Time.now.to_s)
md5.update(Time.now.usec.to_s)
hash = md5.hexdigest
```

## Put Case

When a file is requested to be stored at the configured HSM, the dCache Admin server calls the script, providing the pnfsid of the file and some other useful information, such as the size and the original path of the file in the PNFS filesystem.

For a regular file, we check whether the filesystem special option exists, parsing the parameters string. If it does not exist, the file must be stored directly to our tape server. We use rfcop as an unprivileged user to copy the file to Castor.

```
system("su -command='rfcp <file_path> /castor/pic.es/<path>/'
dcache")
```

If the filesystem parameter exists, and it is a regular file, file information is saved in a temporary log file, and nothing is copied.

When an end of transfer special file is received, the filesystem is ready to be sent to tape. However, during the development of the application, we found out that dCache does not call the HSM script for each file in a first come, first serve basis. Instead, the files are being flushed to HSM in no particular order.

Due to this situation, we must know how many files are to be contained in the filesystem. First of all, the end of transfer special file contains the number of files that were uploaded to dCache during the copy process. The helper script we provided does this automatically.

The number of expected files in the end of transfer file is compared to the

number of files present in the temporary log file described before. The script, then, it is forced to sleep until all the expected files have been registered by the script.

Using the information in the log file, an estimated filesystem size (precisely, blocks of 1kb) is calculated. Using the loop devices in the Unix filesystem, we can then create the virtual filesystem.

Due to the fact that several instances of our HSM interface may be running at a time, we must search for the next available loop device:

```
while system("losetup /dev/loop#i &>/dev/null")
    i+=1
end
loopdev = "/dev/loop#i"
```

The filesystem we are generating is in fact a regular ext2 FS. Using the `/dev/zero` special device, we create a file of the needed size. Using `losetup`, we can map the file to a virtual `/dev/loop` device where we can create an ext2 FS using the OS `mkfs.ext2` tool.

```
#Create filesystem
system("dd if=/dev/zero of=/home/filesystems/#{fsname} bs=1024 count=#{size}")
system("losetup #{loopdev} /home/filesystems/#{fsname}")
system("mkfs.ext2 #{loopdev}")
system("losetup -d #{loopdev}")
system("mkdir /home/mounts/#{fsname}")
system("mount -o loop /home/filesystems/#{fsname} /home/mounts/#{fsname}")
```

Using the hash generated by the client, we create a new register in the filesystems table, which we will later link to the files.

Again, using the log file, we iterate through the filesystem files, creating an entry in the files table, linking it to the filesystems ID, and we copy them to the mounted filesystem.

Using an external expect-based script, if configured, we connect to the dCache Admin, and we force them to be removed from cache manually.

Once all files are copied, we unmount the filesystem, copy it as an unprivileged user to Castor, and remove it from disk.

## Get Case

When a file is requested to be stored at the configured HSM, the dCache Admin server calls the script, providing the pnfsid of the file. We have first to check whether the file is already on cache. In some cases, as we will explain later, the HSM script is called even though the file has already been copied to cache.

Using the pnfsid provided by dCache, we search the file in our database. If the file is not present in the table, it means it is a single file, not contained in a filesystem, and thus, we can fetch it directly using rfcop.

On the other hand, if a match is found in the database, this means the file is part of a filesystem. The unique filesystem hash is fetched from the database. Using the hash, we can now retrieve the filesystem from Castor.

Once the filesystem is downloaded to a temporary disk partition, it can be mounted. Again, in a similar way to the put case, several scripts can be fetching files from tape, and thus, many filesystems may be mounted at the same time.

```
mount -o loop /home/filesystems/#filesystem /home/mounts/#filesystem
```

When arriving at this stage, the script must know how this filesystem has to be treated. The readahead property in the filesystems table specifies whether this filesystem should be entirely copied to cache or not.

If the read ahead technique is not active for this filesystem, only the requested file is copied to the file pool, the filesystem is unmounted, and deleted from temporary cache.

When the read ahead option has been enabled for the filesystem, information about all the files belonging to this filesystem is queried to the database. The first task is to copy all the files from the filesystem to the dCache pool.

Notwithstanding that the files are copied to the dCache pool, dCache still will not be aware of their presence, so we have to force the cache of the file using the dCache Admin interface.

Due to the Admin interface being based in a SSH version 1 protocol, using blowfish encryption, we were unable to use the easier Net::SSH module for Ruby that implements a SSH version 2 client. Instead, we spawned an ssh client using expect, an external tool used to automate interactive applications, to connect to the dCache Admin, to force the files to be cached.

```
send - "/opt/d-cache/dcap/bin/dctp $path"
```

When this dCache touch program is performed, dCache will call the HSM script again, even though the file is already copied in the pool, since dCache has no record of the files being fetched from the HSM.

The HSM script will now then, easily be able to know whether a file is being requested by a user or by ourselves by checking if the pnfsid requested is indeed already in the disk pool.

```
if !File.exists? "/home/pool/pool/data/#pnfsid"
```

Once all files are copied and cached, the filesystem may be unmounted and removed from the temporary cache.

### 3.2.3 Additional scripts

Several other scripts or applications had to be developed either to assist the main HSM interface script or to ease the upload or download actions to dCache.

#### dctouch

We needed to create a piece of software that forced dCache to read the file, thus requesting it to our HSM script. This was solved modifying the source code of dCap, particularly the dccp code, to perform a touch (i.e. read one byte), using the dc\_read dCache API.

```
buf = malloc(sizeof(char));  
  
n = dc_read(src, buf, 1);  
*size = n;  
  
free(buf);
```

This C program was compiled against the dCap library.

#### dccp\_dir.rb

This small script is example of how files that will go together in a filesystem have to be sent to dCache. It has been used to perform the production tests. Basically, it crawls a specified folder and its subfolders, copying all the files

into another folder in dCache, with the particularity that these files will be stored in a filesystem instead of separately.

The main considerations that such a script must have into account are the following:

- A unique filesystem id must be generated, hence we use a hash function, similar to the one we use in our script server.
- Pass `-X -filesystem=id` to `dccp` so we can identify filesystems later on the HSM script:  

```
system("/opt/d-cache/dcap/bin/dccp -X -filesystem=#hash #path #destination");
```
- The last file must be named `EndOfTransfer`, and must also have the `-X -filesystem=id` hack on `dccp`.
- Files arrive to the HSM script in no order whatsoever. In order to know when we have all the files, and so, we can proceed to generate the filesystem, we need to know the number of files expected. The content of the named file must be only the number of files that we have sent.

We generate a hash in this script to identify the filesystem, while we are generating another one again in the HSM script.

The reason for this is the user provided filesystem ID can not be trusted, even if we force them to use this particular script. By generating again a FS id in the HSM script, we can assure the id is genuinely unique.

### **read\_test.rb**

This script is a minimal implementation of how files should be retrieved from dCache. It has been used to perform the tests. This script is much simpler than the one used to copy files into dCache, as it does not need the use of filesystem identification. The process is done as if were regular files.

```
Dir.chdir("#{directory}")
files = Dir.glob(File.join("**", "*"))
for file in files
  system("/opt/d-cache/dcap/bin/dccp #{path} #{dest}");
end
```

When provided with a directory name in the PNFS filesystem, it copies all files and directories inside the folder to a specified folder in the local disk.

#### **file\_generator.rb**

This script receives a start string for the filename, the number and size in kb of the files to generate, as well as a pnfs directory where to upload. It generates the required amount of files and uploads them to dCache.

The file is generated as follows:

```
tmpFile = File.open("tmpFile", "w")
size.times do
  tmpFile.puts content
end
tmpFile.close
```

The content variable contains a string with an exact size of 1024 bytes, repeated in 1024-byte blocks. The exact phrase is used frequently in design and computer science for testing purposes, and it is usually called by its first two words, lorem ipsum. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of *de Finibus Bonorum et Malorum* (The Extremes of Good and Evil) by Cicero, written in 45 BC.





# Chapter 4

## Production tests

In this chapter, we describe the tests performed to check both functional and non-functional requirements of the application, and comparing the performance results to the previous situation and alternatives.

### 4.1 Introduction

To be able to properly test our application, we set up a test server, connected to the dCache Admin production node, enabling the HSM interface, submitting the tests to the production node, to check both performance and stability, with real world usage.

Testing the dCache to HSM interface with real world examples is quite difficult, due to the long time the tests require to perform, not only because of the time dCache takes to upload and fetch the files from Castor, but also due to the project processing itself, which accesses the files.

That is the reason why we have included more generic tests, and only have tested one real world example. Generic tests are as valid as real-world tests and take much less time to perform, allowing us to include modifications faster if needed.

### 4.2 Setting up a production server

The installation of a dCache server is explained in a manual in the dCache site, and briefly in the Implementation chapter in this report. However, the

particularity of this dCache installation arose some issues that are described here.

First of all, the `lfs=precious` option must be disabled in the configuration files (located in `config/<pool>.poollist`) of the HSM-enabled pools. This option must also be disabled if present in the general `pool.batch` file in the dCache Admin server. Disabling this option allows the dCache server to consider that files stored in its pools are not precious copies, and may be deleted if necessary if the files have already been copied to the HSM.

In addition, to identify the files that are going to be linked to the HSM-enabled pools, we enabled a new group in the PoolManager domain, using the dCache Admin interface. This group will be identified by a folder rooted under the `data/` directory in the `pnfs` filesystem.

```
psu create vivo-test write-link-group
psu addto vivo-test write-link-group write-link
```

The following command will enable the use of the HSM interface named `osm-vivo` to the group:

```
psu set vivo-test attribute -r write-link-group HSM=osm-vivo
```

The HSM-enabled pools must have an external script defined to flush and fetch data into and from the HSM server. In our case, this was done for both pools, issuing the following command in the pool domains, linking them to our HSM identifier previously created (`osm-vivo`):

```
hsm set osm-vivo -command=/opt/vivo/vivocp
```

Each HSM-enabled pool must have access to the Castor tape server, where precious copies of the files will be stored and retrieved. The Castor client version must be 2.0. Newer versions of Castor will not work in our environment, due to Castor version incompatibilities.

The following RPM packages were installed using the PIC repository:

- `castor-lib`
- `castor-commands`
- `castor-rtcopy-messages`, `castor-rtcopy-client`
- `castor-rfio-client`

These packages provide the NS tools we need (`nsls` and `nsmkdir`, as Castor replacements for `ls` and `mkdir`), and `rfcp`, allowing us to copy files to the `/castor` special path.

The user that runs the expect script that connects to the dCache Admin server must have the ssh key of the server cached. Otherwise, the received data will differ from what is expected and the script will fail.

This is due to the fact that the ssh client has to cache the ssh server fingerprint the first time it connects to a server, showing the following message:

```
[dcache@dc030 vivo]$ ssh -c blowfish -p 22223 admin@dchead
The authenticity of host 'dchead (193.146.197.105)' can't be established.
RSA1 key fingerprint is 45:08:31:78:61:16:37:f8:02:ac:43:99:c1:81:19:0f.
Are you sure you want to continue connecting (yes/no)?
```

As opposed to the subsequent connections, when the server fingerprint has already been cached:

```
[dcache@dc030 vivo]$ ssh -c blowfish -p 22223 admin@dchead
Password:
```

The script must be installed in the path indicated in the hsm setting in the dCache PoolManager domain. Once this is set, the following variables have to be set in the main script accordingly:

- **vivo\_dir** contains the path where the main script and subscripts are located.
- **mounts\_dir** is the path where the directories needed to mount the filesystems fetched from Castor will be created. This will be used while creating new filesystems as well.
- **fs\_dir** is the temporary path where filesystems fetched from Castor are stored while they are being mounted and copied to dCache. It needs to be as big as possible, to allow concurrency.
- **castor\_path** is the Castor directory where our script will save and fetch the files or filesystems.
- **castor\_user** must contain a valid local user, which will be used to retrieve and copy files to Castor.

## 4.3 Performance

We performed several tests to see whether our HSM interface performed as expected, comparing it to a similar situation where the files would not be stored and fetched together, and also comparing it to a non-tape based solution.

The tests performed consisted in storing several file sets to the dCache server, testing three possible situations: not using a tape server (i.e. storing the precious copy of the files in a dCache disk drive array pool), copying files individually to the tape server, and copying files to the tape server in a filesystem container.

The time that the files take to upload appear to be approximately the same in all cases for the user, as the files are first stored in dCache, and afterwards sent to the HSM script.

Therefore, what we call performance, is the measure of the read process. While the writing process will surely vary between the different solutions, it does not affect user performance because the writing is performed in background.

The file sets consisted in a 1,000x1KiB file set, a 1,000x1MiB file set, and a 10x1GiB file set.

### 4.3.1 Disk drive pool

#### 1,000x1KiB files

Generate and write 1000 files of 1x1kb blocks each, using only disk pools, storing them into /pnfs/pic.es/data/test/disk-1000-test:

```
# time ./file_generator.rb lorem 1000 1 disk-1000-test no
real 14m40.866s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/disk-1000-test:

```
# time ./read_test.rb disk-1000-test disk-1000-test
real 10m3.838s
```

### 1,000x1MiB files

Generate and write 1000 files of 1024x1kb blocks each, using only disk pools, storing them into /pnfs/pic.es/data/test/disk-1m-1000-test:

```
# time ./file_generator.rb lorem 1000 1024 disk-1m-1000-test no
real 15m24.314s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/disk-1m-1000-test:

```
# time ./read_test.rb disk-1m-1000-test disk-1m-1000-test
real 11m2.271s
```

### 10x1GiB files

Generate and write 10 files of 1Gb each, using only disk pools, storing them into /pnfs/pic.es/data/test/disk-1g-10-test:

```
# time ./file_generator.rb lorem 10 1048576 disk-1g-10-test no
real 25m3.551s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/disk-1g-10-test:

```
# time ./read_test.rb disk-1g-10-test disk-1g-10-test
real 24m32.442s
```

## 4.3.2 Tape storage without filesystem generation

### 1,000x1KiB files

Generate and write 1000 files of 1x1kb blocks each, using HSM, without filesystem generation, storing them into /pnfs/pic.es/data/test/disk-1000-test:

```
# time ./file_generator.rb lorem 1000 1 hsm-1000-test yes
real 13m24.246s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/hsm-1000-test:

```
# time ./read_test.rb hsm-1000-test hsm-1000-test
```

```
real 45m53.135s
```

### 1,000x1MiB files

Generate and write 1000 files of 1024x1kb blocks each, using only disk pools, storing them into /pnfs/pic.es/data/test/hsm-1m-1000-test:

```
# time ./file_generator.rb lorem 1000 1024 hsm-1m-1000-test yes
real 14m1.451s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/hsm-1m-1000-test:

```
# time ./read_test.rb disk-1m-1000-test hsm-1m-1000-test
real 44m51.341s
```

### 10x1GiB files

Generate and write 10 files of 1Gb each, using only disk pools, storing them into /pnfs/pic.es/data/test/hsm-1g-10-test:

```
# time ./file_generator.rb lorem 10 1048576 hsm-1g-10-test yes
real 26m44.277s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/hsm-1g-10-test:

```
# time ./read_test.rb hsm-1g-1000-test hsm-1g-1000-test
real 51m13.458s
```

## 4.3.3 Tape storage with filesystem generation

### 1,000x1KiB files

Copy 1000 files of 1x1kb blocks each, previously generated, using an HSM with filesystem generation, storing them into /pnfs/pic.es/data/test/fs-1000-test:

```
# time ./dccp_dir.rb /tmp/disk-1000-test /pnfs/pic.es/data/fs-1000-test

real 11m35.656s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/fs-1000-test:

```
# time ./read_test.rb fs-1000-test fs-1000-test
real 19m17.934s
```

### 1,000x1MiB files

Copy 1000 files of 1000x1kb blocks each, previously generated, using an HSM with filesystem generation, storing them into /pnfs/pic.es/data/test/fs-1m-1000-test:

```
# time ./dccp_dir.rb /tmp/disk-1m-1000-test /pnfs/pic.es/data/fs-1m-1000-test

real 14m44.241s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/fs-1m-1000-test:

```
# time ./read_test.rb fs-1m-1000-test fs-1m-1000-test
real 23m48.339s
```

### 10x1GiB files

Generate and write 10 files of 1Gb each, using only disk pools, storing them into /pnfs/pic.es/data/test/fs-1g-10-test:

```
# time ./dccp_dir.rb /tmp/disk-1g-1000-test /pnfs/pic.es/data/fs-1g-1000-test

real 27m18.563s
```

Retrieve the files previously generated from /pnfs/pic.es/data/test/fs-1g-10-test:

```
# time ./read_test.rb fs-1g-10-test fs-1g-10-test
real 58m45.893s
```

#### 4.3.4 Test analysis

First of all, as expected, the disk drive pool solution is the one performing better in all tests. The single file storage (without filesystems) is performing better as file sizes increase. The results can be shown graphically in Figure 4.1.

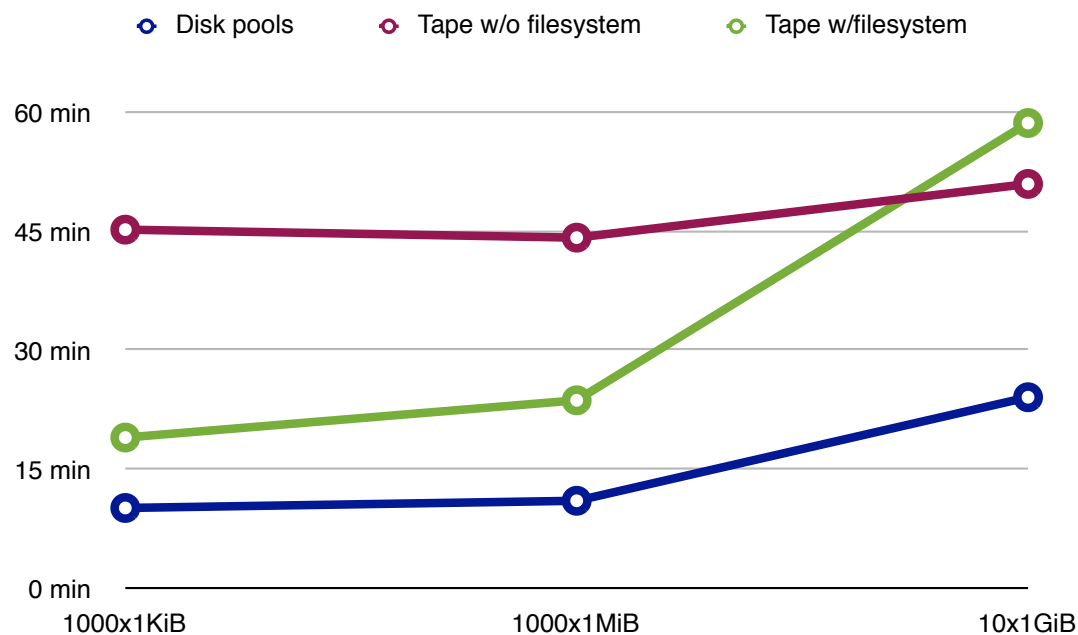


Figure 4.1: Performance comparison of different solutions

The filesystem generation process, then, is found to perform better in a wide range of situations, particularly with small sized files. This proves the already known fact that tape storage systems do not work well with small files, mainly due to the robot positioning, tape fetching and file search overhead.

Considering the results, also confirmed by several previous tests performed using ViVo, the maximum file size which is optimal to be stored in virtual filesystems is about 1GiB. File sizes larger than 1GiB should not be stored in virtual filesystems, as the performance is not only improving, but slightly decreasing. This is due to the filesystem mount and unmount processes, which introduce an important overhead.

In order to describe easily this fact, we can see in Figure 4.2 how time is spent in a tape read when dealing with small (or heterogeneously sized) files.



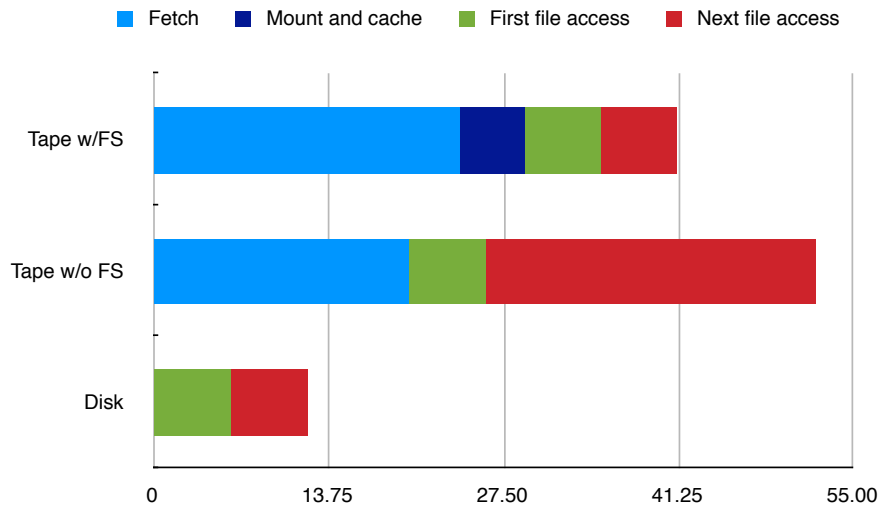


Figure 4.2: Time elapsed on a file request - Small files

While the filesystem generation solution spends some extra time fetching and mounting a bigger filesystem, provided that the overhead in tape reads is, indeed, the positioning of the robot and the tape sequential scan, this overhead is usually neutralized in the subsequent file accesses.

The FS solution has already cached the rest of the filesystem files, therefore next reads will be as fast as a standard disk read, whereas a non-filesystem based solution will have to repeat the retrieval process for each file. The total tape overhead will be more noticeable as smaller files are.

On the other hand, when most of the files are bigger than 1GiB, we found out the performance of our solution decreases (see Figure 4.3).

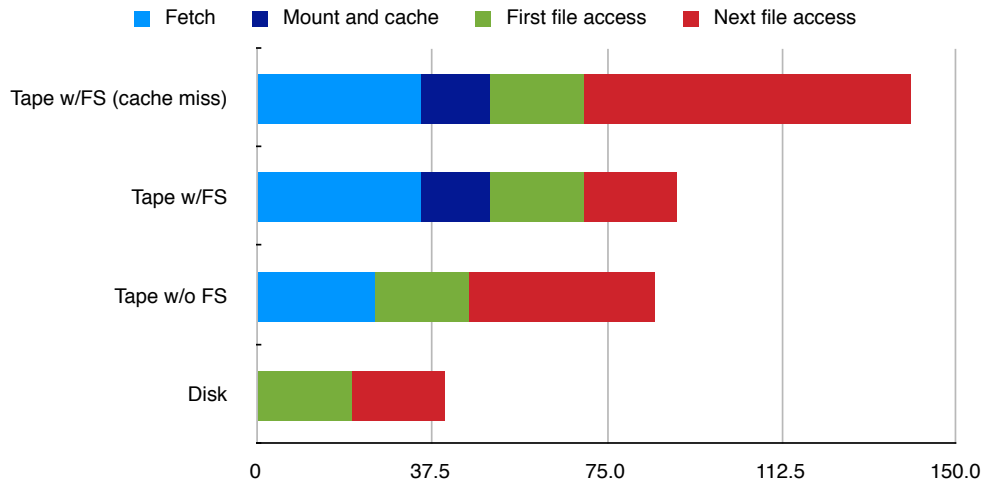


Figure 4.3: Time elapsed on a file request - Large files

The reason of this behaviour can be due to either one or both possible reasons:

- When a project uses a large file, it will usually take some time until the next file is needed. Using a filesystem-based solution will force to retrieve the entire filesystem, likely big enough to reduce the time the tape robot takes to position itself on the tape and scan it compared to the time it takes for the server to download such a big file, mount it and copy all the files to the disk cache (Tape w/FS).
- While the process was reading one of the files, the rest (of some of them) have been deleted from the disk cache by dCache due to lack of available disk space for other requests. This causes the next file access to fetch the entire filesystem again, decreasing the performance substantially, with a behaviour resembling that of a non-filesystem based solution, with the inconvenience of having to download the entire filesystem again (Tape w/FS (cache miss)).

Eventually, our FS-based solution, after several file accesses, even with big files, may perform better than a non-FS based solution, in ideal conditions, i.e, no files are thrown out of cache and almost all files in the filesystem are used.

While the first point is in some cases negligible, depending on how the project accesses the files and how many free disk space dCache has available,

the second point is likely to happen in a production environment. In that case, the system performance will be severely undermined.

### 4.3.5 Magic performance

Magic is based on the ROOT framework, an object-oriented framework written in C++ aimed at solving the data analysis challenges of high-energy physics, created at CERN by René Brun and Fons Rademakers.

It provides platform independent access to a computer's graphics subsystem and Operating System using abstract layers. Parts of the abstract platform are: a graphical user interface and a GUI builder, container classes, reflection, a C++ script and command line interpreter (CINT), object serialization and persistence.<sup>[9]</sup>

A project based in ROOT, like Magic, has to be compiled against the ROOT framework. In our case, some of the files of the project had to be modified, to reflect changes particularly on paths and file access.

Particularly, we had to change the environment variable DATADIR, which was using the following ViVo path:

```
DATADIR = /ViVo/magiclp/RAW
```

setting it to:

```
DATADIR = /pnfs/pic.es/data/test/magiclp/RAW
```

While the Magic project currently used POSIX accesses to the files (i.e. using the standard UNIX tools), to be able to use dCache as a file storage system, the open file functions had to be changed, as we explained earlier on this report.

Thanks to the ROOT framework having a lot of user support, we found that the newest versions of ROOT had a dCache client for PNFS (both local and dCap access). Thus, the lines needed to be changed for Magic to work using dCache are the TFile instances.

```
TFile *f = TFile::Open("root://dcache-admin//pnfs/pic.es  
/data/test/test.file.root")
```

Using the following alternative to access the file via dCap if the dCache admin server is not on the same machine where the script is executed (most likely):

```
TFile *f = TFile::Open("dcap://dcache-admin//pnfs/pic.es  
/data/test/test.file.root")
```

Additionally, the Magic project scripts use Torque/PBS, a resource management and scheduler. When a script is submitted to Torque, it is queued and submitted to an available node when possible. Torque allows the use of external and more powerful schedulers, such as Maui. For the sake of this project, we will use the default pbs\_sched scheduler bundled with Torque.

The default `./configure ; make ; make install` configuration works well, and `make packages` will generate auto-installable packages for both server and clients.

We have been testing and adapting the Magic scripts and found to be working hassle-free. However, due to the large amount of disk space that Magic requires we were not able at the time of writing this report to perform a quantitative performance test, being the cause the dCache production server not entirely working with the project yet.

Regardless, we tested a small reduced set using Magic RAW files with our test server, and the results proved to be similar to those found in the generic tests. Therefore, we can conclude the performance of Magic using our solution will increase remarkably.

# Chapter 5

## Conclusion

In this project we set out to create a link between a high cost pool disk array and a lower cost Tertiary Storage System based on tapes as the basis for creating a system that could supply a better quality of service than the current production service.

We created a dCache to HSM interface, which provides a connection between both systems, while making use of the existing dCache cache removal policy, and thus freeing us of having to implement several policy levels.

Compared to the existing ViVo production service, this project provides a faster and more reliable connection between both layers of storage systems. By relying in dCache to act as a intermediate cache (as ViVo did with a temporary storage server) for files fetched from a tape server, we both remove the need of an additional dedicated server, and improve the removal cache policy.

After gathering information about the projects that may want to use the solution that this project provides in the future, we went on to extend it, allowing whether to use read-ahead policies or not, depending on the project.

Previous to the start of the project, we set up a list of objectives. We believe all the objectives were accomplished with this project. While avoiding modifying the dCache source code, and minimally changing the current dCache configuration, we managed to integrate our solution into the existing dCache production server, enabling it to act as a hierarchical storage manager, keeping the already working groups not using an external HSM.

We managed to develop this interface between dCache and the HSM script, and between the HSM script and Castor, while allowing use of other tape server with minimal changes to the script.

## 5.1 Future work

The following areas are interesting and merit further research:

- Further performance testing

While several tests were performed to check whether this project provided a better performance than the previous solution, there was not enough time to thoroughly test the performance with other projects, mainly due to the large amount of time that such tests take to finish.

We believe further testing would have allowed to tune several aspects of the scripts to improve performance, using different read-ahead policies depending on projects.

- Enstore support

Enstore, a mass storage component which provides access to data on tape or other storage media, was another mass storage system we would have wanted to be supported by our script. However, due to Enstore still not being fully supported by PIC, and the current production status of Castor, inclined us to work with Castor instead.

- Storage mode decision

Currently, the user must decide whether to create virtual filesystems or store files individually. Even though it was not originally stated neither in the requirements nor in the design phase of the project, we lately found out that a feature which automatically decided whether to use virtual filesystems or to send directly the files to the tape server, depending both on the size and the amount of the files will be useful to improve the performance of the system.

# Bibliography

- [1] Mathias de Riese, Patrick Fuhrmann et al. *dCache, the Book: A guide for administrators of dCache systems*. <http://www.dcache.org/manuals/Book/>.
- [2] Dave Thomas, Chad Fowler, Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*, The Pragmatic Bookshelf, Raleigh, North Carolina, second edition, 2006, ISBN 0-974-51405-5.
- [3] Castor - Documentation <http://castor.web.cern.ch/castor/docs.htm>
- [4] RDoc Documentation - Ruby API <http://www.ruby-doc.org/core/>
- [5] SQLite/Ruby Interface API <http://sqlite-ruby.rubyforge.org/>
- [6] CERN breaks gigabyte/s storage-to-tape barrier with StorageTek <http://press.web.cern.ch/press/PressReleases/Releases2003/PR06.03EStoragetek.html>
- [7] World's largest scientific Grid sustains a million jobs per month <http://press.web.cern.ch/press/PressReleases/Releases2006/PR12.06E.pdf>
- [8] LCG - Project Overview <http://lcg.web.cern.ch/lcg/overview.html>
- [9] The ROOT Framework <http://root.cern.ch/>





## Foreword

As modern research relies more and more on computers, data storage is becoming a scarce resource for research projects, as well as a large part of the cost. Some projects try to solve this problem by relying on distributed data storage. It is therefore necessary, for some centers, to provide massive amounts of lower cost storage based in tape libraries. The drawback to this approach is that performance decreases, particularly when dealing with large amounts of small files.

Our goal is to create a hybrid between a high-cost high-performance disk drive pool array, and a lower-cost, not so high performance tape based library. To this end, we will link dCache, a distributed storage system, to Castor, a hierarchical storage management system, while creating virtual filesystems containing large amounts of small files to improve the overall performance of the system.

---

A medida que la investigación depende cada vez más de los computadores, el almacenamiento de datos comienza a ser un recurso escaso para los proyectos y supone una gran parte del coste total. Algunos proyectos intentan solucionar estos problemas usando almacenamiento distribuido, por ello, es necesario que algunos centros proporcionen almacenamiento masivo de bajo coste basado en librerías de cintas. El inconveniente de esta solución reside en que el rendimiento disminuye, particularmente, cuando se trata de grandes cantidades de archivos pequeños.

Nuestro objetivo es crear un híbrido entre un sistema de almacenamiento de alto coste y rendimiento basado en discos, y otro de bajo coste y rendimiento basado en cintas. Para ello, uniremos dCache, un sistema de almacenamiento distribuido, con Castor, un sistema de almacenamiento jerárquico, creando así sistemas de archivos virtuales que contengan grandes cantidades de archivos pequeños para mejorar el rendimiento global del sistema.

---

A mesura que l'investigació depen cada vegada més dels computadores, l'emmagatzematge de dades comença a convertir-se en un recurs escàs per als projectes, i suposa una gran part del cost total. Alguns projectes intenten resoldre aquest problema emprant emmagatzament distribuit. És doncs, necessari, que alguns centres proveeixin de grans quantitats d'emmagatzematge massiu de baix cost basat en cintes magnètiques. L'inconvenient d'aquesta solució és que el rendiment disminueix, particularment, alhora de tractar-se de grans quantitats d'arxius petits.

El nostre objectiu és crear un híbrid entre un sistema d'alt cost i rendiment basat en discs, i un de baix cost i rendiment basat en cintes. Per això, unirem dCache, un sistema d'emmagatzematge distribuit, amb Castor, un sistema d'emmagatzematge jeràrquic, creant sistemes de fitxers virtuals que contindran grans quantitats d'arxius petits per millor el rendiment global del sistema.