硕士学位论文

Dissertation for Master's Degree

(工程硕士)

(Master of Engineering)

基于 CastorFS 和 Xrootd 的用户空间文件系统的进一步设计与实现

Further implementation of the user space file system based on CastorFS and Xrootd

Manjun JIAO

哈爾濱工業大學

Université Blaise Pascal

2011 年 9 月

国内图书分类号：TP311 学校代码：10213
国际图书分类号：681 密级：公开

工程硕士学位论文
Dissertation for the Master’s Degree in Engineering
(工程硕士)
(Master of Engineering)

基于CastorFS和Xrootd的用户空间文件系统的进一步设计与实现
Further implementation of the user space file system based on CastorFS and Xrootd

硕士研究生：	焦满峻
导　　师：	Niko NEUFELD
副 导 师：	徐晓飞, Kun-Mean HOU
申请学位：	工程硕士
学　　科：	软件工程
所在单位：	软件学院
答辩日期：	2011 年 9 月
授予学位单位：	哈尔滨工业大学

Classified Index: TP311
U.D.C: 620

Dissertation for the Master's Degree in Engineering

Further implementation of the user space file system based on CastorFS and Xrootd

Candidate:	JIAO Manjun
Supervisor:	Niko NEUFELD
Associate Supervisor:	Prof. Xiaofei XU Prof Kun-mean HOU
Academic Degree Applied for:	Master of Engineering
Speciality:	Software Engineering
Affiliation:	School of Software
Date of Defence:	September, 2011
Degree-Conferring-Institution:	Harbin Institute of Technology

摘　要

在欧洲核物理研究委员会（CERN），科学家们用大型强子对撞（LHC）实验使用海量存储系统来对数以千 T 字节的实验数据进行存储和管理。这个系统就是 CASTOR（CERN 高级存储管理系统）。它为进行科学实验提供了强大和便捷的数据支撑。然而，用户在使用这个数据管理系统的时候只能用其提供的具体命令行去访问和操作其中的数据。这样一个基于 FUSE（用户空间文件系统）和两个 CASTOR 输入输出库（NS 库和 Rfio 库）的文件系统 CastorFS 被开发出用于进一步简化 CASTOR 系统的使用。尽管此文件系统已经成功被部署在 LHCb 的节点上，然后它有很大的缺陷极大地限制了其应用范围。当用户想要获取某个存储大量数据的文件夹时，它使用的两个库极大地增加了 CASTOR 服务器端的负载，而且它们也不提供任何数据保护的机制。除此而外，缓慢的数据传输速度也极大地限制了文件系统的应用。这样，我们需要开发一个缓存来有效地对短时期要用的大量数据进行存储并使用新的数据传输协议 Xrootd 来重新构建 CastorFS 以降低 CASTOR 服务器端的负载，提供更快的数据传输服务，提供安全的数据访问机制。新的系统实现 LRU（Least recently used）算法作为实现缓存管理的机制，运用 FUSE 提供的实现虚拟文件系统的接口以及 Xrootd 所提供的符合 POSIX 标准的功能函数来提供一个有效降低服务器端负载，传输速度更加稳定可靠，带有安全认证功能的用户空间文件系统，当前系统以及在 LHCb 的节点上经过测试并已经被成功部署以服务于科学计算。

关键词：CastorFS，用户空间文件系统，缓存，Xrootd

Abstract

The LHC (Large Hadron Collider) experiments use a mass storage system for recording petabytes of experimental data. This system is called CASTOR[1] (CERN Advanced STORage manager) and it is powerful and convenient for many use cases. However, it is impossible to use standard tools and scripts straight away for dealing with the files in the system since the users can access the data just in the way of using command-line utilities and parsing their output. Thus a complete POSIX filesystem – CastorFS[2] is developed based on the FUSE[3] (File System in Userspace) and two CASTOR I/O libraries-RFIO (Remote File I/O) library and NS (Name Server) library. Although it is applied successfully, it has serious limitation of wide application because the I/O protocols it relies on are very old. Each time the CASTOR side receives the calling for accessing data files from the user side, the load of the CASTOR side system will increase quickly even out of its upper bound and the system would crash. Besides that, those two protocols provide very primitive mechanism for authentication and the data transmitting rate is very low. Furthermore, since the CastorFS is implemented in C language, it can hardly be extended by using the other well functioned libraries. Hence, it needs to be rewritten in C++ with the implementation of caching and the Xrootd[4] (eXtended Root Daemon) which is developed by SLAC/CERN to provide a high performance, scalable fault tolerant access to data. The new system implemented the LRU (Least Recently Used) algorithm as the mechanism to manage the cache. We use FUSE as the tool to provide the interfaces for the virtual filesystem. By implementing the actual filesystem function with Xrootd Posix APIs, we provided a filesystem which can effectively ease the server load, provide a steady data transmitting service and provide a good data protection mechanism.

Keywords: CastorFS, FUSE, Caching, Xrootd

Table of contents

Chapter 1 Introduction

1.1 Background

The CERN Advanced STORage manager (CASTOR) is a hierarchical storage management system developed at CERN for physics data files. Files can be stored, listed, retrieved and remotely accessed using CASTOR command-line tools or user applications developed against the CASTOR API. Multiple access protocols are available such as RFIO (Remote File IO), ROOT, XROOT and GridFTP. CASTOR exposes also a SRM interface. The design is based on a component architecture using a central database to save guard the state changes of CASTOR components. The access to disk pools is controlled by the Stager; the directory structure is kept by the Name Server. The tape access (write and recalls) is controlled by the Tape Infrastructure.

The FUSE system was originally part of A Virtual Filesystem (AVFS), but has since split off into its own project on SourceForge.net. FUSE is available for Linux, FreeBSD, NetBSD (as PUFFS), OpenSolaris, and Mac OS X. It was officially merged into the mainstream Linux kernel tree in kernel version 2.6.14. FUSE is particularly useful for composing virtual file systems. Unlike traditional file systems that essentially save data to and retrieve data from disk, virtual filesystems do not actually store data themselves. They act as a view or translation of an existing file system or storage device. In principle, any resource available to FUSE implementation can be exported as a file system [5]. There are many file systems and applications were developed based on FUSE, like run-time-access, KIO FUSE Gateway, LUFS bridge, mcachefs, Logic File System, GnomeVFS2 FUSE, AFUSE, Mountlo, etc. However, for interaction with the CASTOR system, this is a new application and a practical one[6].

XROOTD aims at giving high performance, scalable fault tolerant access to data repositories of many kinds. The typical usage is to give access to file-based ones. It is based on a scalable architecture, a communication protocol, and a set of plugins and tools based on those[7]. The freedom to configure it and to make it scale (for size and performance) allows the deployment of data access clusters of virtually any size, which can include sophisticated features, like authentication/authorization, integrations with other systems[8], WAN data distribution, etc. Recently, starting from the fact that xrootd is just the name of one constituting block (the data access daemon), the name "Scalla" is being sometimes used to refer to the whole software suite. Its meaning is "Structured Cluster Architecture for Low Latency Access". This Savannah point of access is supposed to grow and constitute a point of aggregation for the various needs of people willing to use the platform for their data access needs in the HEP community[9]. This will include access to updated source code and to the documentation, as well to any other kind of information related to the project. Xrootd is a newly developed tool and it is not be used widely, so it's a new application full of challenge. It is a high performance network storage system widely used in High Energy Physics experiments such as Babar, STAR and LHC. The underline Xroot data transfer protocol provides very high efficient access to the ROOT based data files[10]. Using filesystem which is based on Xrootd to access data files will not take the advantages provided by the Xroot data transfer protocol[11]. For this reason, the preferred environments to use Xrootd are data import, export and data management, not the actual data analysis. Because Xrootd is designed with large data files in mind, it is not efficient to use the filesystem based on Xrootd for large number of small files[12].

1.2 CASTOR, FUSE, CASTORFS and Xrootd

1.2.1 CASTOR

CASTOR is a hierarchical storage management (HSM) system developed at CERN. Files can be stored, listed, retrieved and accessed in CASTOR using command line tools or applications built on top of the RFIO (Remote File IO) or ROOT libraries[13][14].

CASTOR provides a UNIX like directory hierarchy of file names. The directories are always rooted /castor/cern.ch (the cern.ch will be different in other CASTOR sites). The CASTOR name space can viewed and manipulated only through CASTOR client commands and library calls. OS commands like ls or mkdir will not work on CASTOR files. The CASTOR name space holds permanent tape residence of the CASTOR files, while the more volatile disk 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.

CASTOR name space can be viewed using the `nsls` or `rfdir` commands. Both commands use the same mechanism for talking to CASTOR but there are important differences:

- `nsls` can also list the tape residence (`-T` option) and supports a special mode-bit 'm' flagging that the file has been migrated to tape
- `rfdir` is a RFIO command and can therefore also be used to list local or remote files. `nsls` can *only* list CASTOR files

Example:

```
[lxplus] nsls -l /castor/cern.ch/user/l/linda
mrw-r--r--   1 linda aa        29194240 Mar 08  2004 thesis.tar
mrw-r--r--   1 linda aa        16723666 Jan 14  2004 muons.root
mrw-r--r--   1 linda aa        2496 Aug 12 10:06 logfile
drwxr-xr-x 102 linda aa        0 Jul 20 13:45 higgs
[lxplus] rfdir /castor/cern.ch/user/l/linda
-rw-r--r--   1 linda aa        29194240 Mar 08  2004 thesis.tar
-rw-r--r--   1 linda aa        16723666 Jan 14  2004 muons.root
-rw-r--r--   1 linda aa         2496 Aug 12 10:06 logfile
drwxr-xr-x 102 linda aa         0 Jul 20 13:45 higgs
[lxplus] rfdir .
drwxr-xr-x   6 linda root       4096 Oct 09  2003 private
drwxr-xr-x  27 linda root       4096 Aug 19 20:25 public
-rw-r--r--   1 linda root        547 Oct 19  2000 .login
-rw-r--r--   1 linda root       3905 Dec 10  1996 .profile
-rw-r--r--   1 linda root       6228 Sep 28  2004 .tcshrc
-rw-r--r--   1 linda root       2151 Dec 10  1996 .zprofile
-rw-r--r--   1 linda root       3436 Dec 10  1996 .zshenv
-rw-r--r--   1 linda root       4159 Dec 10  1996 .zshrc
[lxplus] nsls -T /castor/cern.ch/user/l/linda/thesis.tar
    -  1       1   P16116         1663   00353758     29194240     0
/castor/cern.ch/user/l/linda/thesis.tar
```

Figure 1-1 is the overview of CASTOR2,

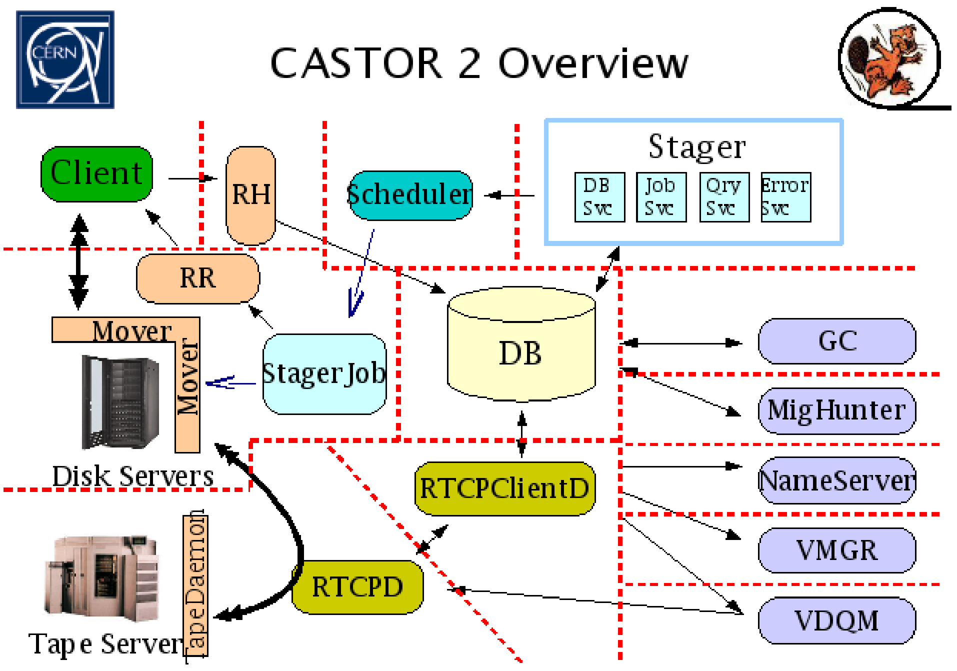


Figure 0-1-1 Castor status and overview

The functionality of CASTOR covers a wide range of the requirements of the data management: SRM conventions for client command set, transactions for input streams, handling of new queries, pluggable policies, request priorities, pluggable protocols and SRM interfaces. Besides that, it also integrated authorization[15], authentication, resiliency against hardware failures and disaster recovery functions. The scalability and the flexibility which are two important features in the domain of high performance computing are also supported by CASTOR.

CASTOR provides a UNIX like directory hierarchy of file names. This directory structure can be accessed using rfio (Remote File Input/Output) protocols either at the command level or, for C programs, via function calls. The service at RAL has an SRM interface which makes it GRID accessible[16][17].

There are two families of commands that can be used to access CASTOR locally at RAL:-

rf*: The rfio commands which can access both local and remote files. See setting the environment for a warning about getting the right version of rf*.

ns*: The ns* CASTOR name server commands have additional functionality but can only be used on local CASTOR files.

These commands don't use GRID certificates, it's all down to UNIX permissions; when files are created they will be owned by the user running on the

UI. If subsequently accessed via some GRID service the username will normally be different but so long as it belongs to the same group i.e. 'minos' then group attributes can be used to control access.

1.2.2 FUSE

A filesystem is a method for storing and organizing computer files and directories and the data they contain, making it easy to find and access them. If somebody is using a computer, he/she is most likely using more than one kind of filesystem. A filesystem can provided extended capabilities. It can be written as a wrapper over an underlying filesystem to manage its data and provide an enhanced, feature-rich filesystem (such as cvsfs-fuse, which provides a filesystem interface for CVS, or a Wayback filesystem, which provides a backup mechanism to keep old copies of data)[18].

Before the advent of user space filesystems, filesystem development was the job of the kernel developer. Creating filesystems required knowledge of kernel programming and the kernel technologies (like vfs). And debugging required C and C++ expertise[19]. But other developers needed to manipulate a filesystem -- to add personalized features (such as adding history or forward-caching) and enhancements.

Now, in a userspace program, we can implement a fully functional filesytem by using FUSE. It provides features like simple library API, simple installation, secure implementation, and efficient kernel interface in userspace. And, to top it all off, FUSE has a proven track record of stability[20]. To create a filesystem in FUSE, we need to install a FUSE kernel module and then use the FUSE library and API set to create our filesystem[21][22]. The supper block, inode, dengry etc. are all virtual. The request of the real information of a file will be passed from layer to layer through drivers and interfaces until request handling program written by user in the user space.

Figure 1-2 shows the data flow used by FUSE to access remote data. FUSE contains three modules: FUSE kernel module, LibFUSE module, User program module[23]. In user space, users should implement the filesystem which is encapsulated by the Libfuse library. Libfuse provide the support to the main filesystem framework, encapsulation for “user implemented filesytem” code, handling "mount", communication with operating system module through character device /dev/fuse.

The kernel module of FUSE has implemented the VFS interface which is used for FUSE file driver module registration, the virtual device driver of FUSE, providing maintenance of supper block, dentry, inode. FUSE kernel will receive the VFS's requests and pass them to LibFUSE. LibFUSE will pass them to our user program interface to actually do the job (Figure 1-2).

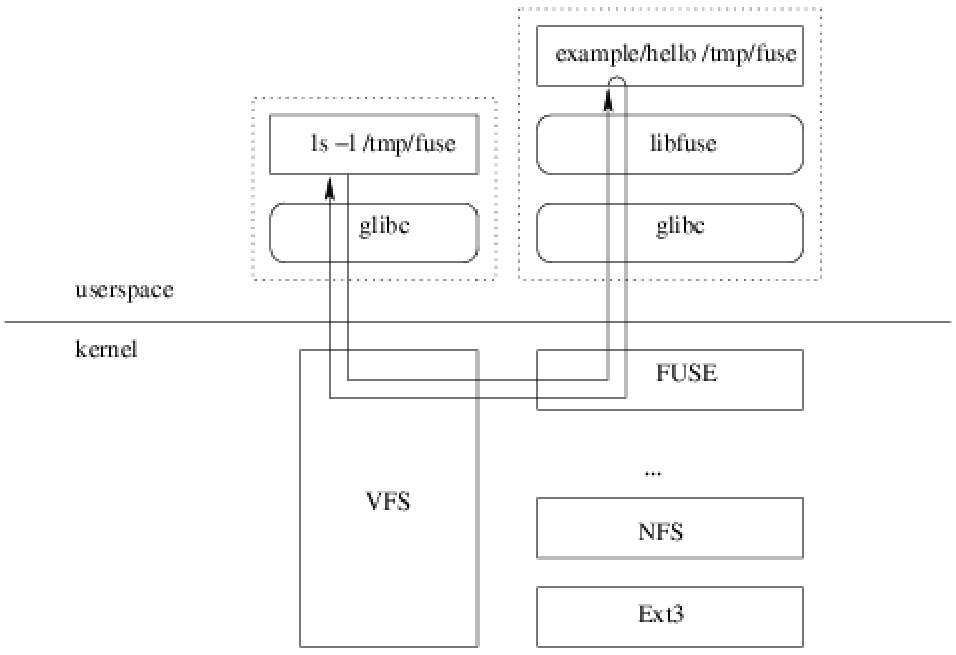


Figure 1-0-2 How fuse works

When our user mode program calls fuse_main() (lib/helper.c), fuse_main() parses the arguments passed to our user mode program, then calls fuse_mount() (lib/mount.c).

fuse_mount() creates a UNIX domain socket pair, then forks and execs fusermount (util/fusermount.c) passing it one end of the socket in the FUSE_COMMFD_ENV environment variable.

fusermount (util/fusermount.c) makes sure that the fuse module is loaded. fusermount then open /dev/fuse and send the file handle over a UNIX domain socket back to fuse_mount().

fuse_mount() returns the filehandle for /dev/fuse to fuse_main().

fuse_main() calls fuse_new() (lib/fuse.c) which allocates the struct fuse datastructure that stores and maintains a cached image of the filesystem data. Lastly, fuse_main() calls either fuse_loop() (lib/fuse.c) or fuse_loop_mt() (lib/fuse_mt.c) which both start to read the filesystem system calls from the /dev/fuse, call the usermode functionsstored in struct fuse_operations datastructure before calling fuse_main(). The results of those calls are then written back to the /dev/fuse file where they can be forwarded back to the system calls.

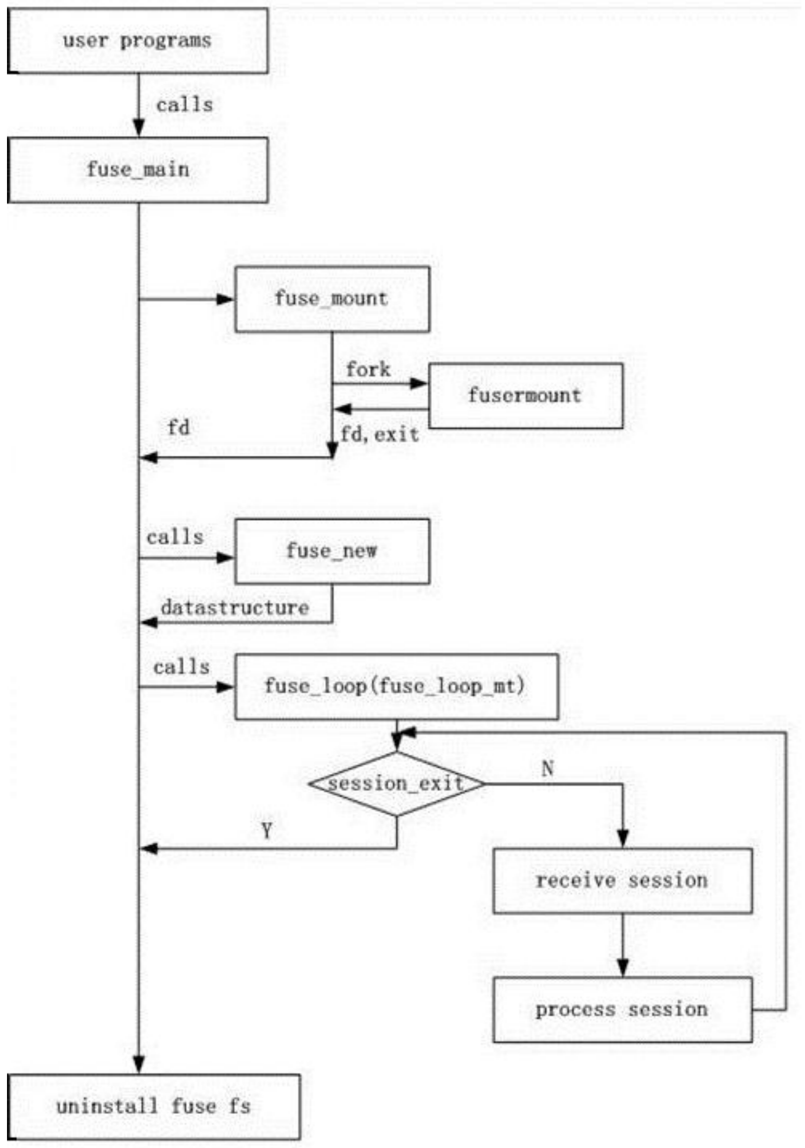


Figure 1-0-3 Fuse working procedure

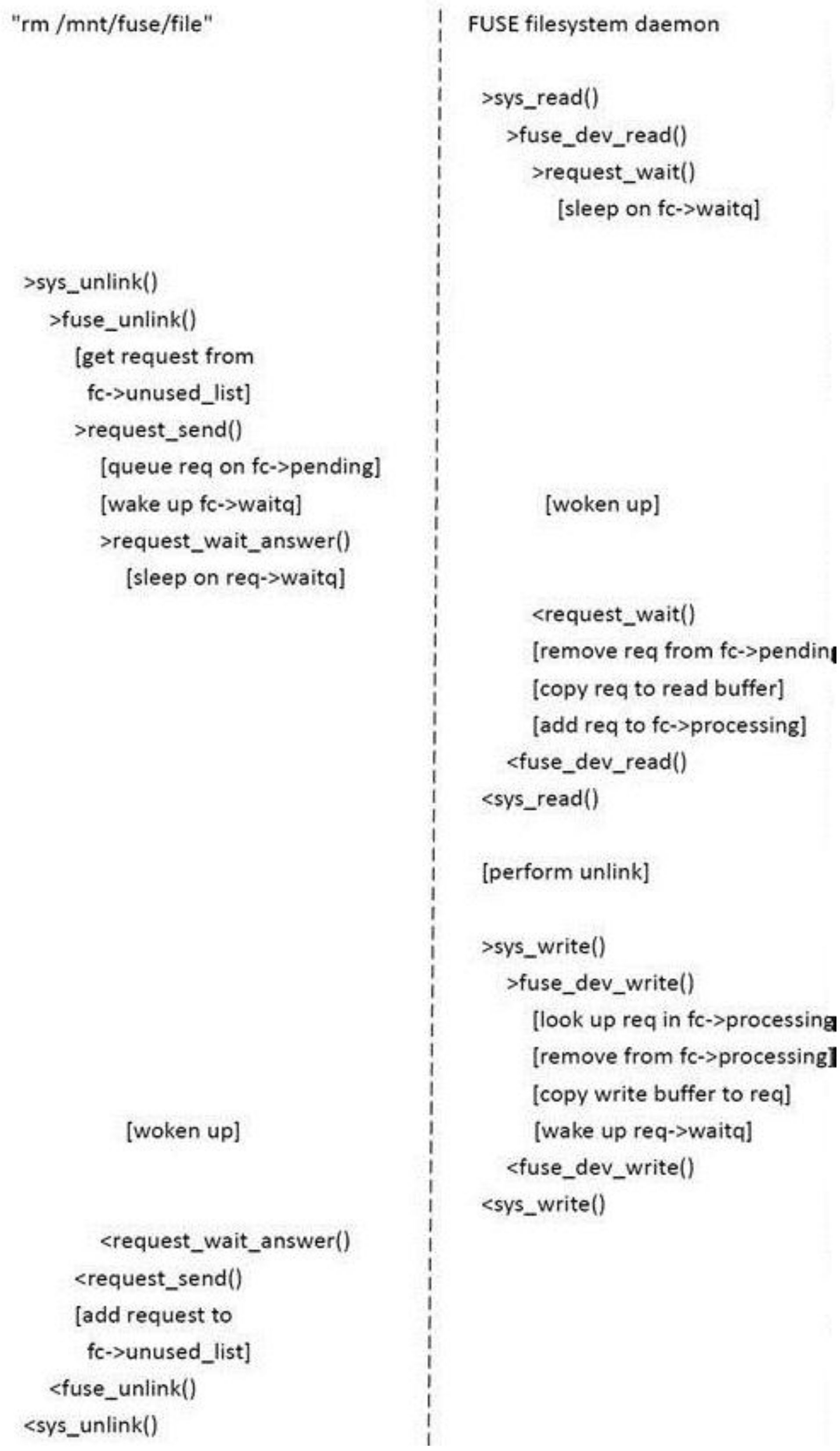


Figure 1-0-4 rm function call procedure in FUSE

To create a filesystem with FUSE, we need to declare a structure variable of type `fuse_operations` and pass it on to the `fuse_main` function. The `fuse_operations` structure carries a pointer to functions that will be called when the appropriate action is required[24]. None of those operations are absolutely essential, but many are needed for a filesystem to work properly. We can implement a full-featured filesystem with the special-purpose methods `.flush`, `.release`, or `.fsync`. Some functions are explained as follows:

- getattr: int (*getattr) (const char *, struct stat *);
  This is similar to stat(). The st_dev and st_blksize fields are ignored. The st_ino field is ignored unless the use_ino mount option is given.
- readlink: int (*readlink) (const char *, char *, size_t);
  This reads the target of a symbolic link. The buffer should be filled with a null-terminated string. The buffer size argument includes the space for the terminating null character. If the linkname is too long to fit in the buffer, it should be truncated. The return value should be "0" for success.
- `getdir: int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);`
  This reads the contents of a directory. This operation is the `opendir()`, `readdir()`, ..., `closedir()` sequence in one call. For each directory entry, the `filldir()` function should be called.
- `mknod: int (*mknod) (const char *, mode_t, dev_t);`
  This creates a file node. There is no `create()` operation; `mknod()` will be called for creation of all non-directory, non-symlink nodes.
- `mkdir: int (*mkdir) (const char *, mode_t);`
  `rmdir: int (*rmdir) (const char *);`
  These create and remove a directory, respectively.
- `unlink: int (*unlink) (const char *);`
  `rename: int (*rename) (const char *, const char *);`
  These remove and rename a file, respectively.
- `symlink: int (*symlink) (const char *, const char *);`
  This creates a symbolic link.
- `link: int (*link) (const char *, const char *);`
  This creates a hard link to a file.

- `chmod: int (*chmod) (const char *, mode_t);`
  `chown: int (*chown) (const char *, uid_t, gid_t);`
  `truncate: int (*truncate) (const char *, off_t);`
  `utime: int (*utime) (const char *, struct utimbuf *);`
  These change the permission bits, owner and group, size, and access/modification times of a file, respectively.
- `open: int (*open) (const char *, struct fuse_file_info *);`
  This is the file open operation. No creation or truncation flags (`O_CREAT`, `O_EXCL`, `O_TRUNC`) will be passed to `open()`. This should check if the operation is permitted for the given flags. Optionally, `open()` may also return an arbitrary filehandle in the `fuse_file_info` structure, which will be passed to all file operations.
- `read: int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);`
  This reads data from an open file. `read()` should return exactly the number of bytes requested, except on EOF or error; otherwise, the rest of the data will be substituted with zeroes. An exception to this is when the `direct_io` mount option is specified, in which case the return value of the `read()` system call will reflect the return value of this operation.
- `write: int (*write) (const char *, const char *, size_t, off_t, struct fuse_file_info *);`
  This writes data to an open file. `write()` should return exactly the number of bytes requested except on error. An exception to this is when the `direct_io` mount option is specified (as in the `read()` operation).
- `statfs: int (*statfs) (const char *, struct statfs *);`
  This gets filesystem statistics. The `f_type` and `f_fsid` fields are ignored.
- `flush: int (*flush) (const char *, struct fuse_file_info *);`
  This represents flush-cached data. It is not equivalent to fsync() -- it's not a request to sync dirty data. flush() is called on each close() of a file descriptor, so if a filesystem wants to return write errors in close() and the file has cached dirty data, this is a good place to write back data and return any errors. Since many applications ignore close() errors, this is not always useful.

1.2.3 CastorFS

CASTOR logically presents files in a UNIX (POSIX) like directory hierarchy of file names[25][26]. This suggests implementing a new filesystem capable of operating on files stored on CASTOR using standard UNIX operation system calls and commands like open, read, cp, rm, mkdir, ls, cat and find.

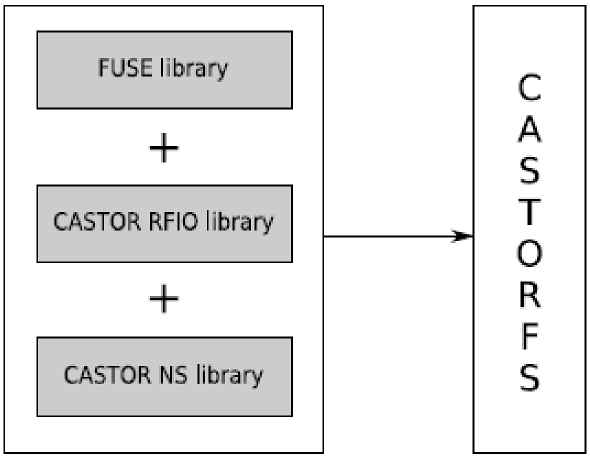


Figure 1-0-5 CastorFS

CastorFS have a performance problem for writing and reading files to/from CASTOR compared to native RFIO about 14 times slower for writing and 3 times slower for reading1 (see Table 1-1). This project will make a further implementation for CastorFS by adding new module for the filesystem and using new data transmitting policy to make a big improvement on CastorFS[27].

Table 1-1 CastorFS performance

Tool	Read (Mb/s)	Write (Mb/s)
POSIX *cp* command on CastorFS	31	5
rfcp command (based on CASTOR RFIO library)	100	70

1.2.4 Xrootd

Scalla means Structured Cluster Architecture for Low Latency Access[28]. This is the relatively new name given to the whole suite of tools which are part of the (formerly called) XRootD distribution. The Scalla software suite provides two fundamental building blocks: an xrootd server for low latency highbandwidth data access and an olbd server for building scalable xrootd clusters. Scalla offers a readily deployable framework in which to construct large fault-tolerant high performance data access configurations using commodity hardware with a minimum amount of administrative overhead[29].

The xrootd server is designed to provide POSIX-like access to files and their enclosing directory namespace[30]. The architecture is extensible in that it relies heavily on a run-time plug-in mechanism so that new features can be added with a minimum of disruption[31]. The plug-in components are shown in Figure 1-6. Seven plug-in components are shown. The components mate (i.e., plug in) at different architectural junctions.

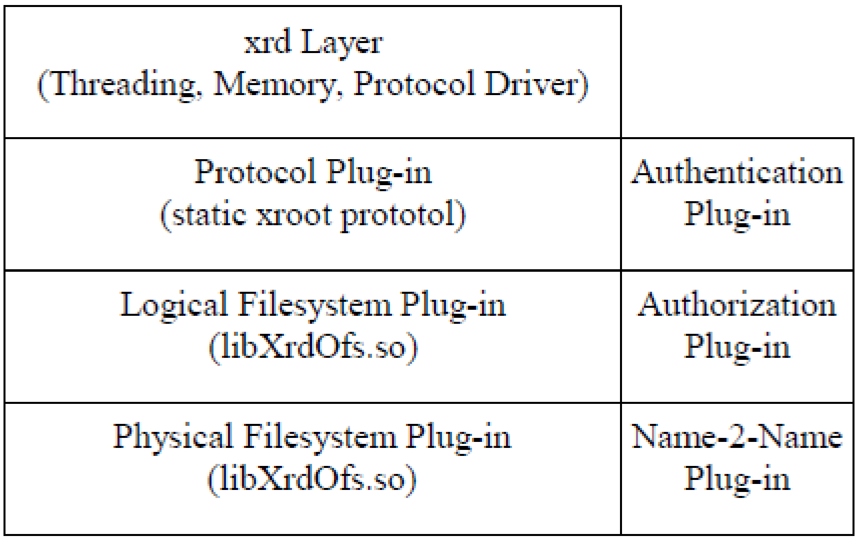


Figure 1-0-6: Xrootd Server Architecture

The core component is the "xrd". This component is responsible for network, thread, data buffer, and protocol management. Because the "xrd" is responsible for a compact set of functions[32], it was easily optimized to do them exceedingly well. For instance, network management was engineered to use the most efficient mechanism available for each type of host operating system. Data buffer management is optimized to provide fast allocation and de-allocation of I/O buffers on page boundaries. Protocol management is designed to allow any number of protocols to be used at the same time. The protocol is selected at the time an initial connection is made to the server. By default, the component that provides the xroot protocol is statically linked with the "xrd". As mentioned before, additional protocols may be specified, and the "xrd" loads these at run-time from appropriate shared libraries. For instance, the PROOF system runs both the xroot protocol as well a special protocol that provides parallel access to multiple data analysis servers within the Root Framework.

The authentication component, XrdSec, plugs into the xroot protocol component. Multiple authentication protocols can be used as the xroot protocol is merely used to encapsulate the client/server interactions required by the protocol. Currently, GSI, Kerberos IV and V, as well as simple password authentication are supported. Additional authentication protocols may be implemented and placed in shared libraries[33]. These protocols are dynamically loaded and used whenever the client supports the particular protocol. Authentication models may also be restricted on a host name and domain basis[34][35].

1.3 The purpose of project

The purpose of project is to implement a user space file system according to the requirements to provide a better virtual filesytem for the users to facilitate their work and ease the server load by applying interesting mechanisms to the filesystem.

1.4 The status of related application

FUSE is used by many organizations to develop many commercial and nonecomercial applications and products. Here are some introductions of the application FUSE.

1.4.1 Application No.I

Wuala: A multi-platform, Java based Fully OS integrated distributed file system. Using FUSE, MacFUSE and Callback File System respectively for file system integration, in addition to a Java based app accessible from any Java enabled web-browser. It is a secure online storage, file synchronization, versioning and backup, service, originally developed and run by Caleido Inc., which is now part of LaCie. Service is a combination of data centres that are provided by Wuala in multiple European countries (France, Germany, and Switzerland) and the Wuala cloud — distributed data storage that is provided by users who trade storage.

1.4.2 Application No. II

RTA is a library that we can attach to our program to expose our program's internal arrays and data structures as if they were tables in a database. The database interface uses a subset of the Postgres protocol and is compatible with the Postgres bindings for C, PHP, and the Postgres command line tool, psql. One of the problems facing Linux is the lack of run time access to status, statistics, and configuration of a service once the service has started. We assume that to configure an application we will be able to ssh into the box, vi the /etc configuration file, and do a 'kill -1' on the process. Real time status and statistics are things Linux programmers don't even think to ask for. The need for run time access is particularly pronounced for network appliances where ssh is not available or might not be allowed. Another problem for appliance designers is that more than one type of user interface may be required. Sometimes a customer requires that no configuration information be sent over an Ethernet line which transports unsecured user data. In such a case the customer may turn off the web interface and require that configuration, status, and statistics be sent over an RS-232 serial line. Other popular interfaces include the VGA console, SNMP MIBs, and LDAP. The RTA package helps solve both of these problems by giving run time access to the data structures and arrays inside our running program. With minimal effort, we make our program's data structures appear as tables in a Postgres database.

1.5 Main content and organization of the thesis

The main content of this topic is to make an introduction of the further implementation of the CastorFS. The rest of the theis will be requirement analysis, system design, implementation, testing and conclusion. In chapter2, we will introduce the system requirement which includes functional requirement and non-functional requriremnt. In chaper3, we will firstly introduce the overall desingn of the system, and then the design for each importand part of the system will be given. In the end of this chapter, we will see the key technologies which are applied in this project. The last chapter will introduce the system implementation and testing. The implementation and testing environment will firstly be given and then the key interfaces, testing and evaluation will be introduced.

Chapter 2 System Requirement Analysis

2.1 The goal of the system

From the software engineering point of view, we need to do the requirement analysis to define the problem as clearly as possible. After that, we will design and implement the system with the current tools, the libraries that we can use. And then, we will make the test to make sure the quality of software will be guaranteed. At last, an evaluation report will be given to help the other people to well recogonize the improvements between the new and old system and the guidelines will be given to explain how the system should be well configured before people want to use it.

We want to provide filesystem to make users acess the romote data as if they acess the data on their local machine.At the same time, by providing a interface which complies to the POSIX standard, our users will be able to use the common Linux commands to operate the remote data. This will help them work more efficiently.

Specifically, during the internship, I need to make a supplement for the current CastorFS by rewriting it in C++, adding the caching and replacing ns, rfio libraries with Xrootd libriries for transmitting data. After that, I also need to investigate how we deploy the sytem in LHCb computing cluster and write a guideline for the users.

2.2 The functional requirements

2.2.1 The requirement of implementing caching

Before a new CastorFS is implemented by reforming the original one, the system needs be wrapped up to be able isolate the Linux system binding with the FUSE functions to prepare for the furhter implementation. All the implementation functions of the FUSE interfaces will be written in C++ and a middle wrapper layer should be given shown as Figure 2-1.

It is the CastorFS user space program that implements normal I/O operations such as getarrr(), open(), close(), read(), write(), opendir(), readdir(), mkdir(), chown(), truncate(), utimens(), release(), gexattr(), listxattr(), removexattr(), create() and unlink() against the CASTOR storage system.

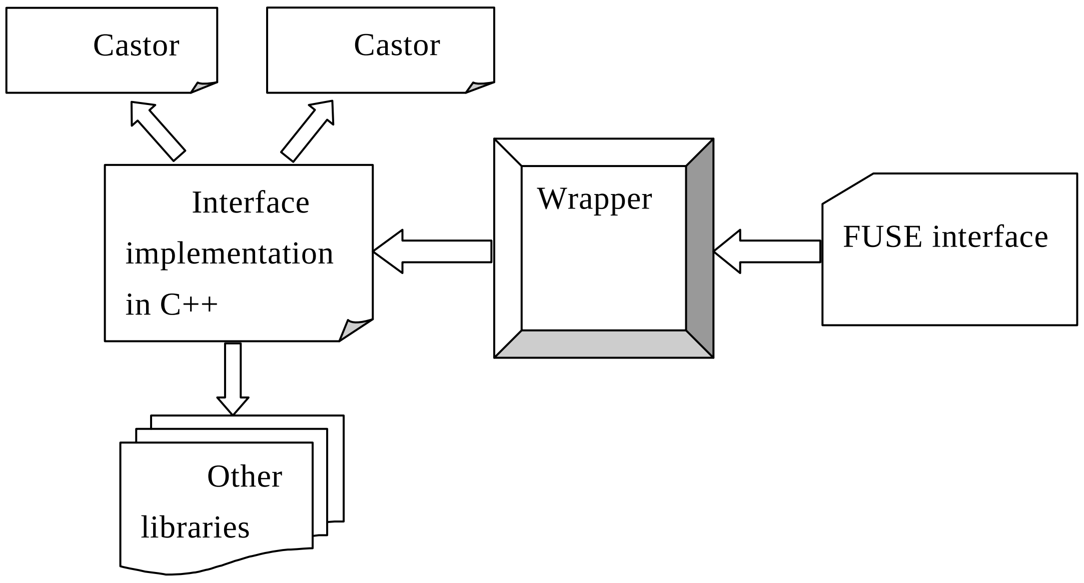


Figure 2-1 Wrap up CastorFS in C++

CASTOR provides a UNIX like directory hierarchy of file names. The directories are always rooted /castor/cern.ch (the cern.ch will be different in other CASTOR sites). The CASTOR name space can viewed and manipulated only through CASTOR client commands and library calls. OS commands like ls or mkdir will not work on CASTOR files. 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. Since the CASTOR needs to deal with many files in the tape, if the user use many Linux file accessing command like “ls”, that will make the load of CASTOR side increasing dramatically. Therefore the frequently used file metadata should be cached during a period of time to reduce the load on the CASTOR server side.

The implementation of a caching mechanism for storing information about frequently requested CASTOR file meta-data is very critical for the wide application of this file system. For the caching, users need to be able to set the caching lasting time, i.e. the efficient lasting time of the cached meta-data because after a period of time there might be some modifications made by the other users on the same file or directory. Another reason for setting the caching efficient time is that CASTOR doesn’t provide any call back function to inform the modification of the files on CASTOR server.

2.2.2 The requirement of applying Xrootd in the system

The xrd is a server that can dynamically support multiple TCP/IP application service layer protocols. The xrd is a generalized daemon and it makes its primary decision on which protocol to support based on the name given to the executable. Currently, the following executable names are fully supported: xrootd for eXtended Root Daemon and related protocols. Records that do not start with a recognized identifier are ignored. This includes blank record and comment lines (i.e., lines starting with a pound sign, #). Other directives are documented in supplemental guide specific to the component they deal with. The location of the configuration file is specified on the xrootd command line. Because each component has a unique prefix, a common configuration file can be used for the whole system. Refer to the manual "Configuration File Syntax" on how to specify and use conditional directives and set variables. These features are indispensable for complex configuration files usually encountered in large installations.

The application of Xrootd is mainly about the using of the new protocols provided by Xrootd and adding the authentication mechanism provided by Xrootd to make the CastorFS faster and more secure.

2.2.3 The requirement of porting the system to Mac and Windows

Since our users use different operation systems, it will be practical if we provide the CastorFS not only on Linux but also on Mac and Windows. The requirement is to implement the CastorFS on Mac which will base on MacFUSE and Windows which will base on the Windows FUSE.

Therefore, after a careful investigation and discussion with experienced software developers in CERN, we found that CASTOR2 client side library didn't support Mac and Windows operating systems. At the same time, Xrootd client library needs a uified authentication support from the CERN server. It would be too complex to implement the functions based on the windows and Mac. So we decided to leave this part to be implemented in the future once Castor and Xrootd provide a full support for the Mac and Windows operating system.

Beside that, we can install the system on the computing cluster, and then we can use the secure shell to exchange data between two networked devices inorder to use the service.

2.3 The un-functional requirements

1. Understandability:

(1) Interface elements should be easy to understand

(2) For a walk up and use system, the purpose of the system should be easily understandable

2. Learnability:

(1) The user documentation and help should be complete

(2) The help should be context sensitive and explain how to achieve common tasks

(3) The system should be easy to learn

3. Operability:

(1) The interface actions and elements should be consistent

(2) Error messages should explain how to recover from the error

(3) The system should be customisable to meet specific user needs

(4) A style guide should be used

4. Attractiveness:

The screen layout and colour should be appealing.

2.4 Summary

Requirements analysis involves frequent communication with system users to determine specific feature expectations, resolution of conflict or ambiguity in requirements as demanded by the various users or groups of users, avoidance of feature creep and documentation of all aspects of the project development process from start to finish. During the requirement analysis, we defined the functional requirement and non-functional requirement for the virtual file system. For the functional requirements, we need to first make some improvements on the old version CastorFS. Later, we need to apply the new xrootd protocol on the CastorFS. During the implementation, the evaluation report also should be delivered to visualize the actual improvement of the system.

Chapter 3 System Design

3.1 The overall design of the system

To design for the new CastorFS will have close interactions with FUSE, caching mechanism and XrdPosix interface. The new CastorFS will implement all the functions that are nessassary for handling the operations in a filesystem by invoking the Xrootd posix functions. Some meta-data of directory entries will be cached for the future use. The overall design is show as Figure 3-1.

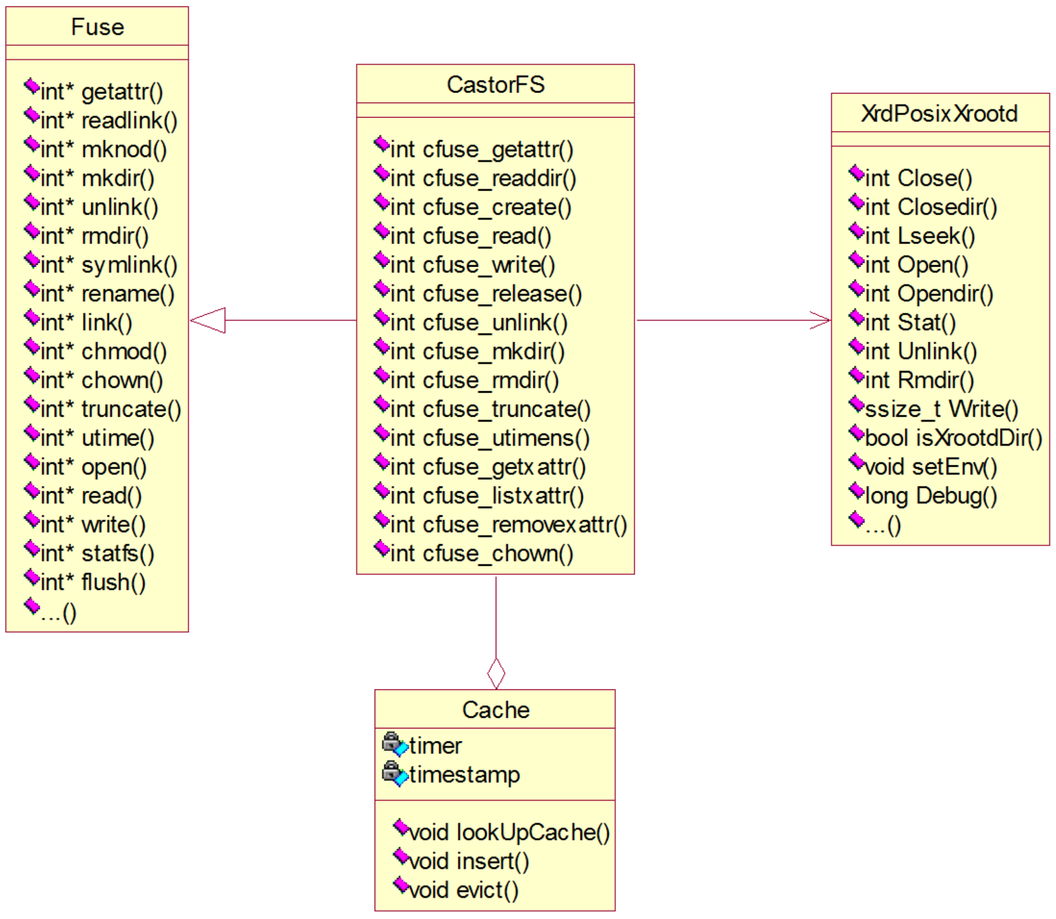
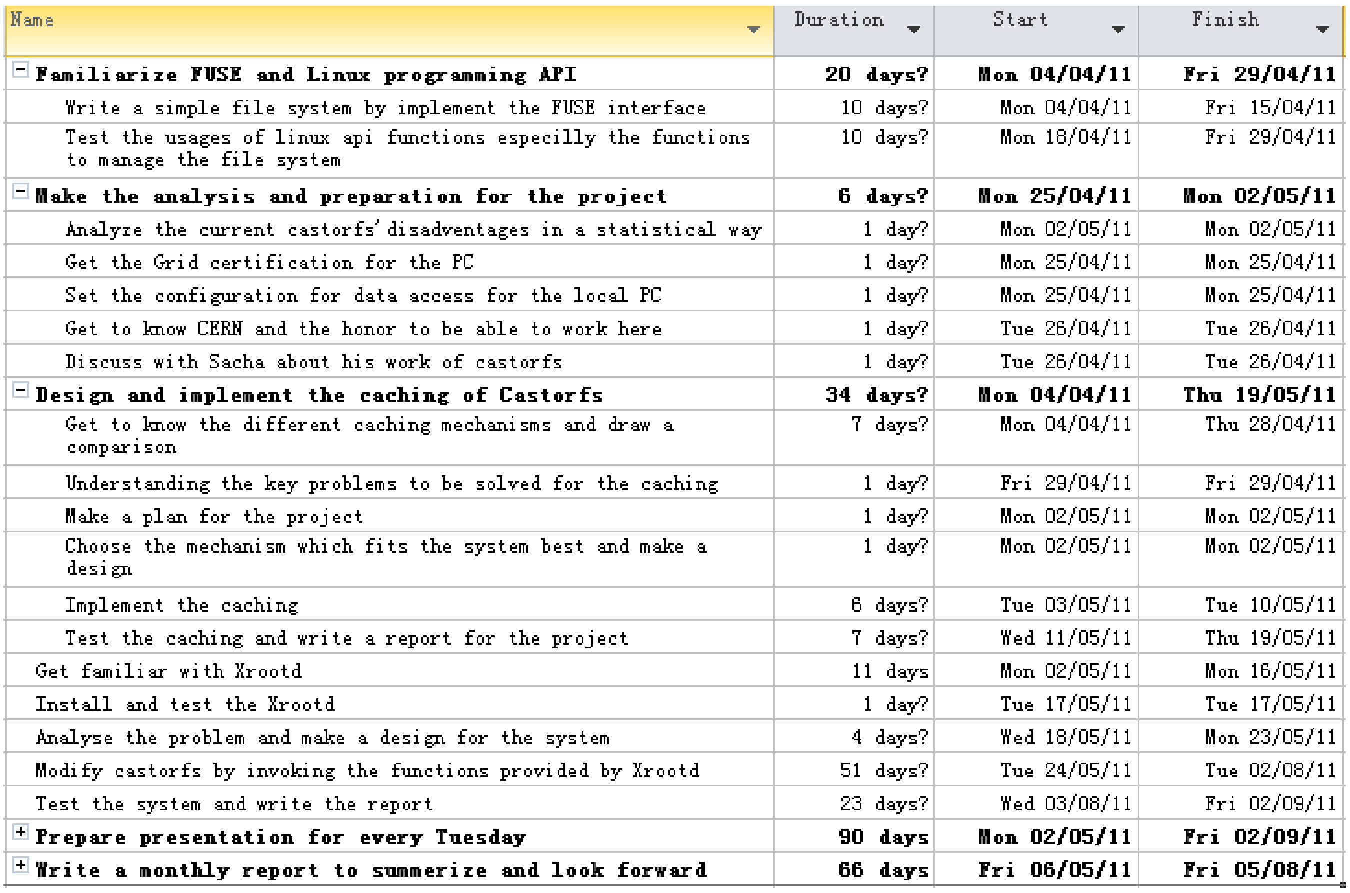


Figure 3-1 System overall design

For the workplan, there are 3 main tasks which are given according to the requirement analysis. Because the work will be done based on some brand new systems and tools, it takes a period of time for me to get familiar with the new systems and tools.

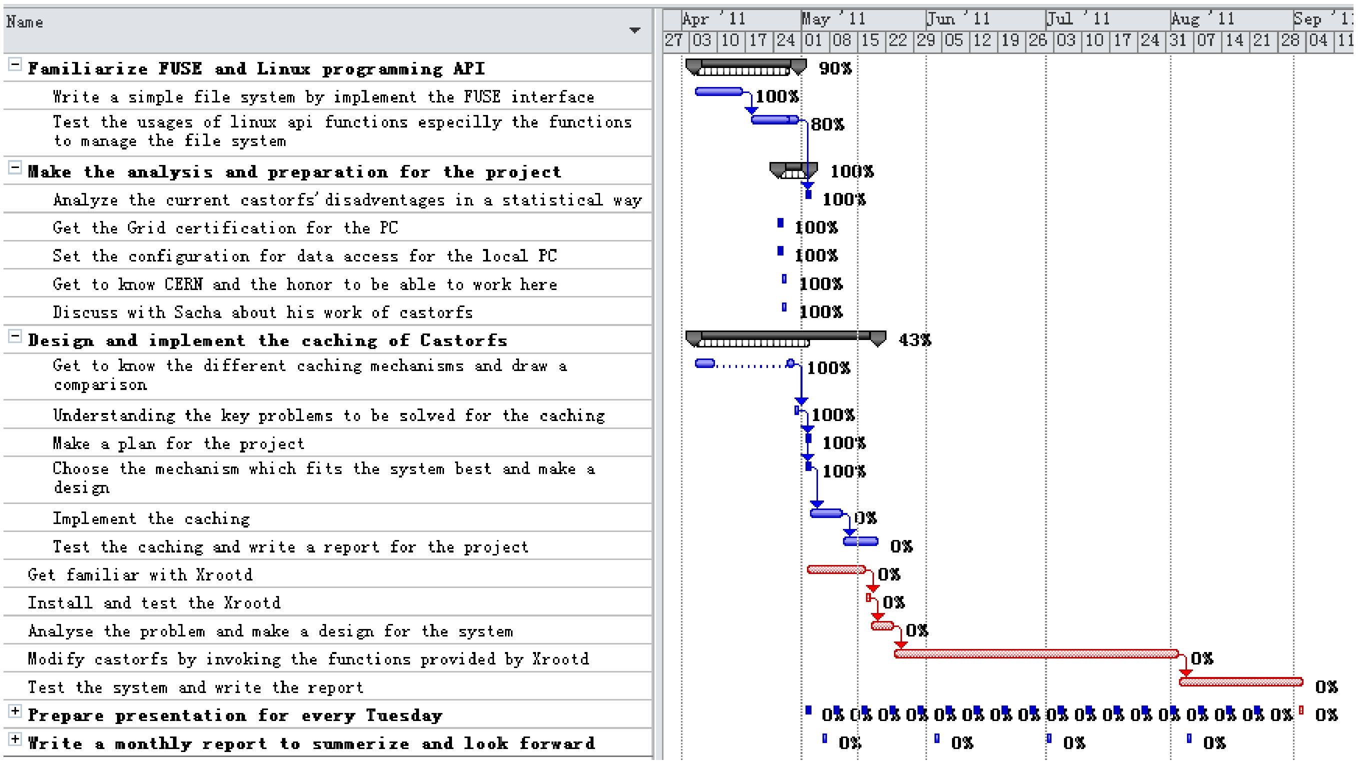
Table 3-1 Workplan



Name	Duration	Start	Finish
⊟ **Familiarize FUSE and Linux programming API**	**20 days?**	**Mon 04/04/11**	**Fri 29/04/11**
Write a simple file system by implement the FUSE interface	10 days?	Mon 04/04/11	Fri 15/04/11
Test the usages of linux api functions especilly the functions to manage the file system	10 days?	Mon 18/04/11	Fri 29/04/11
⊟ **Make the analysis and preparation for the project**	**6 days?**	**Mon 25/04/11**	**Mon 02/05/11**
Analyze the current castorfs' disadventages in a statistical way	1 day?	Mon 02/05/11	Mon 02/05/11
Get the Grid certification for the PC	1 day?	Mon 25/04/11	Mon 25/04/11
Set the configuration for data access for the local PC	1 day?	Mon 25/04/11	Mon 25/04/11
Get to know CERN and the honor to be able to work here	1 day?	Tue 26/04/11	Tue 26/04/11
Discuss with Sacha about his work of castorfs	1 day?	Tue 26/04/11	Tue 26/04/11
⊟ **Design and implement the caching of Castorfs**	**34 days?**	**Mon 04/04/11**	**Thu 19/05/11**
Get to know the different caching mechanisms and draw a comparison	7 days?	Mon 04/04/11	Thu 28/04/11
Understanding the key problems to be solved for the caching	1 day?	Fri 29/04/11	Fri 29/04/11
Make a plan for the project	1 day?	Mon 02/05/11	Mon 02/05/11
Choose the mechanism which fits the system best and make a design	1 day?	Mon 02/05/11	Mon 02/05/11
Implement the caching	6 days?	Tue 03/05/11	Tue 10/05/11
Test the caching and write a report for the project	7 days?	Wed 11/05/11	Thu 19/05/11
Get familiar with Xrootd	11 days	Mon 02/05/11	Mon 16/05/11
Install and test the Xrootd	1 day?	Tue 17/05/11	Tue 17/05/11
Analyse the problem and make a design for the system	4 days?	Wed 18/05/11	Mon 23/05/11
Modify castorfs by invoking the functions provided by Xrootd	51 days?	Tue 24/05/11	Tue 02/08/11
Test the system and write the report	23 days?	Wed 03/08/11	Fri 02/09/11
⊞ **Prepare presentation for every Tuesday**	**90 days**	**Mon 02/05/11**	**Fri 02/09/11**
⊞ **Write a monthly report to summerize and look forward**	**66 days**	**Fri 06/05/11**	**Fri 05/08/11**

The Gantt chart below (in table 3-2) illustrates the start and finish dates of the terminal elements and summary elements of a project. Terminal elements and summary elements comprise the work breakdown structure of the project. This also shows the dependency (i.e., precedence network) relationships between activities. From this chart, we can see clearly the project structure and the overall time assignment.

Table 3-2 Work plan Gantt chart



Name	
⊟ **Familiarize FUSE and Linux programming API**	90%
Write a simple file system by implement the FUSE interface	100%
Test the usages of linux api functions especilly the functions to manage the file system	80%
⊟ **Make the analysis and preparation for the project**	100%
Analyze the current castorfs' disadvantages in a statistical way	100%
Get the Grid certification for the PC	100%
Set the configuration for data access for the local PC	100%
Get to know CERN and the honor to be able to work here	100%
Discuss with Sacha about his work of castorfs	100%
⊟ **Design and implement the caching of Castorfs**	43%
Get to know the different caching mechanisms and draw a comparison	100%
Understanding the key problems to be solved for the caching	100%
Make a plan for the project	100%
Choose the mechanism which fits the system best and make a design	100%
Implement the caching	0%
Test the caching and write a report for the project	0%
Get familiar with Xrootd	0%
Install and test the Xrootd	0%
Analyse the problem and make a design for the system	0%
Modify castorfs by invoking the functions provided by Xrootd	0%
Test the system and write the report	0%
⊞ **Prepare presentation for every Tuesday**	0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0%
⊞ **Write a monthly report to summerize and look forward**	0% 0% 0% 0%

3.2 The design of the wrapper

Since Xrootd is implemented in C++ and there are some sophisticated libraries we can use in C++, before moving on to the caching part, we need to provide a wrapper layer for the original CastorFS. The original CastorFS was implemented in C.

The first requirement for mixing code is that the C and C++ compilers you are using must be compatible. They must, for example, define basic types such as int, float or pointer in the same way. The Solaris Operating System (Solaris OS) specifies the Application Binary Interface (ABI) of C programs, which includes information about basic types and how functions are called. Any useful compiler for the Solaris OS must follow this ABI.

Sun C and C++ compilers follow the Solaris OS ABI and are compatible. Third-party C compilers for the Solaris OS usually also follow the ABI. Any C compiler that is compatible with the Sun C compiler is also compatible with the Sun C++ compiler.

The C runtime library used by our C compiler must also be compatible with the C++ compiler. C++ includes the standard C runtime library as a subset, with a few differences. If the C++ compiler provides its own versions of of the C headers, the versions of those headers used by the C compiler must be compatible.

Sun C and C++ compilers use compatible headers, and use the same C runtime library. They are fully compatible.

FUSE didn’t provide a C++ version, so we need to provide a wrapper for the FUSE to hook the functions up with the real implementations in C++. If we declare a C++ function to have C linkage, it can be called from a function compiled by the C compiler. A function declared to have C linkage can use all the features of C++, but its parameters and return type must be accessible from C if you want to call it from C code. For example, if a function is declared to take a reference to an IOstream class as a parameter, there is no (portable) way to explain the parameter type to a C compiler. The C language does not have references or templates or classes with C++ features.

For wrapping the old CastorFS, we still need to use the ogrial NS and Rfio libraries. And they are written in C. So at the same time, we need to access the C libraries in C++. The C++ language provides a "linkage specification" with which you declare that a function or object follows the program linkage conventions for a supported language. The default linkage for objects and functions is C++. All C++ compilers also support C linkage, for some compatible C compiler.

When you need to access a function compiled with C linkage (for example, a function compiled by the C compiler), declare the function to have C linkage. Even though most C++ compilers do not have different linkage for C and C++ data objects, you should declare C data objects to have C linkage in C++ code. With the exception of the pointer-to-function type, types do not have C or C++ linkage.

3.3 The design of the caching

There are many caching algorithms we can apply for the project.

There are (1) Belady's Algorithm: The most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Belady's optimal algorithm or the clairvoyant algorithm. Since it is generally impossible to predict how far in the future information will be needed, this is generally not implementable in practice. The practical minimum can be calculated only after experimentation, and one can compare the effectiveness of the actually chosen cache algorithm. (2) Least Recently Used (LRU): discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such implementation, every time a cache-line is used, the age of all other cache-lines changes. LRU is actually a family of caching algorithms with members including: 2Q by Theodore Johnson and Dennis Shasha and LRU/K by Pat O'Neil, Betty O'Neil and Gerhard Weikum. (3) Most Recently Used (MRU): discards, in contrast to LRU, the most recently used items first. According to "When a file is being repeatedly scanned in a [Looping Sequential] reference pattern, MRU is the best replacement algorithm." In the authors also point out that for random access patterns and repeated scans over large datasets (sometimes known as cyclic access patterns) MRU cache algorithms have more hits than LRU due to their tendency to retain older data. MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed. (4) Pseudo-LRU (PLRU): For caches with large associativity (generally >4 ways), the implementation cost of LRU becomes prohibitive. If a scheme that almost always discards one of the least recently used items is sufficient, the PLRU algorithm can be used which only needs one bit per cache item to work.

After a careful studying of all those algorithms, we decided to adopt LRU caching algorithm as our method for the implementation because we are more interested in the recently accessed meta-data of a folder. Those meta-data would be more likely to be accessed in a certain period of time.

The need for caching behaviour sometimes arises during system development. Generally the desire is to preserve some expensive-to-obtain results so they can be reused "for free" without repeating the expensive operation in future. Typically the expense arises because a complex calculation is needed to obtain the result, or because it must be obtained via a time consuming I/O operation. If the total number of such results dealt with over the lifetime of the system does not consume excessive memory, it may suffice to store them in a simple key-value cache (for example, a std::map), with the key being the input to the expensive function and the value being the result. This is often referred to as “memoisation” of a function.

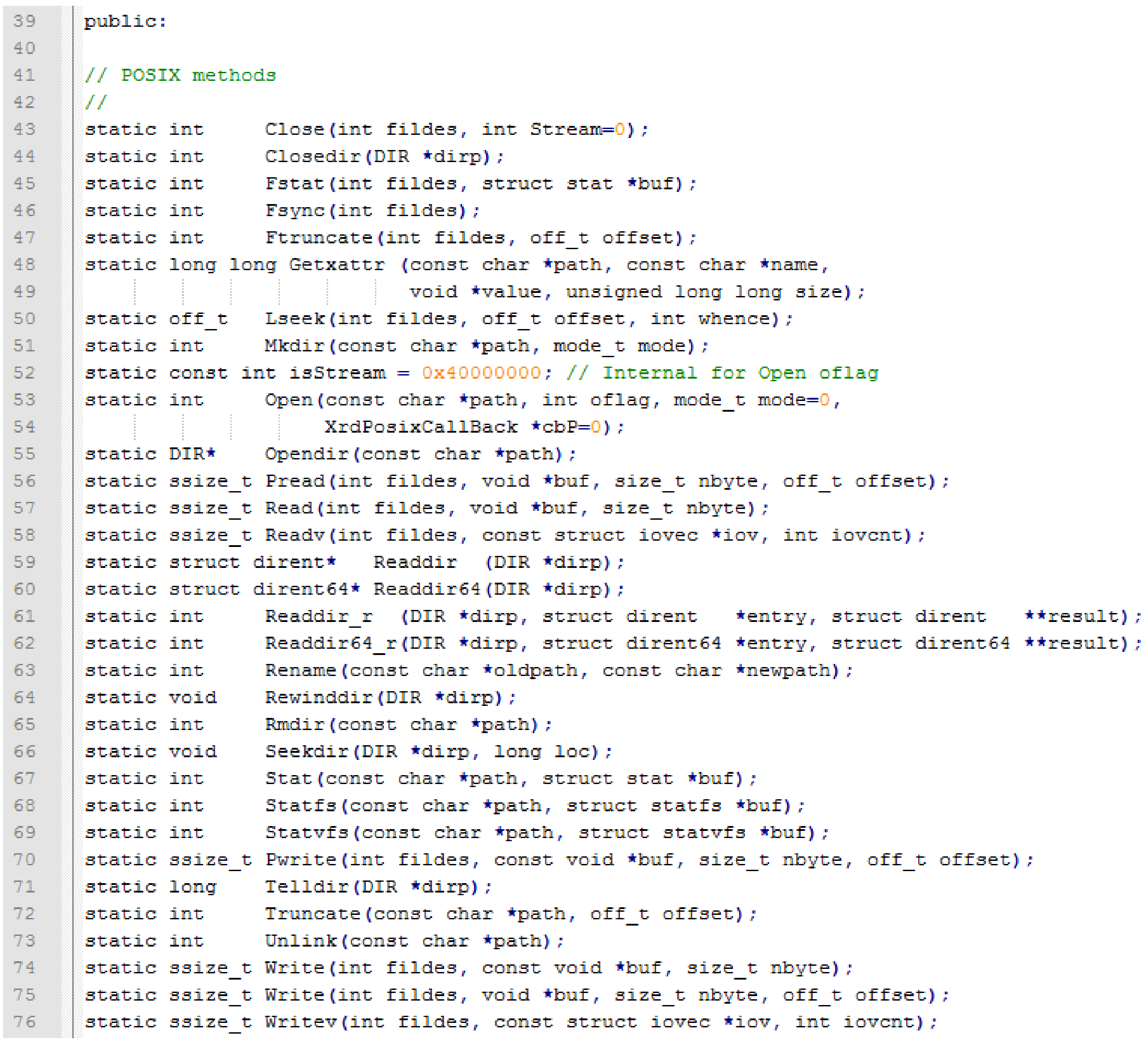
However, for most applications, this approach would quickly consume too much memory to be of practical value. The memory consumption issue can be addressed by limiting the maximum number of items stored in the cache or, if the items have a variable size, limiting the aggregate total stored. Initially the cache is empty and records (key-value pairs) can be stored in it freely. After some further usage, it will fill up. Once full, the question arises of what to do with subsequent additional records which it seems desirable to cache, but for which there is no space (given the limited capacity constraint) without taking action to remove some other records from the store. Assuming the records most recently added to the cache are those most likely to be accessed again (ie assuming some temporal coherence in the access sequence), a good general strategy is to make way for a new record by deleting the record in the cache which was “least recently used”. This is called an LRU replacement strategy.

We will use the C++ standard library and the typical C++ container, iterator as well as algorithm to implement the LRU caching.

3.4 The design of the filesystem using Xrootd

The XrdPosix package allows standard POSIX I/O calls to either vector the I/O to local files or to xrootd served files. In order to use this package we must use the provided POSIX/Xrootd wrapper. We can use the dynamic wrapper or the static wrapper. The dynamic wrapper provides the fastest and easiest way of using xrootd with our application as well as with most Unix commands. The static wrapper provides us with precise control over its deployment and consequently is safer and much faster.

In the file of XrootdPosixXrootd.hh, we can find those POSIX functions which can be directly used in our implementation of new CastorFS:



```
public:

// POSIX methods
//
static int     Close(int fildes, int Stream=0);
static int     Closedir(DIR *dirp);
static int     Fstat(int fildes, struct stat *buf);
static int     Fsync(int fildes);
static int     Ftruncate(int fildes, off_t offset);
static long long Getxattr (const char *path, const char *name,
                           void *value, unsigned long long size);
static off_t   Lseek(int fildes, off_t offset, int whence);
static int     Mkdir(const char *path, mode_t mode);
static const int isStream = 0x40000000; // Internal for Open oflag
static int     Open(const char *path, int oflag, mode_t mode=0,
                    XrdPosixCallBack *cbP=0);
static DIR*    Opendir(const char *path);
static ssize_t Pread(int fildes, void *buf, size_t nbyte, off_t offset);
static ssize_t Read(int fildes, void *buf, size_t nbyte);
static ssize_t Readv(int fildes, const struct iovec *iov, int iovcnt);
static struct dirent*   Readdir  (DIR *dirp);
static struct dirent64* Readdir64(DIR *dirp);
static int     Readdir_r  (DIR *dirp, struct dirent   *entry, struct dirent   **result);
static int     Readdir64_r(DIR *dirp, struct dirent64 *entry, struct dirent64 **result);
static int     Rename(const char *oldpath, const char *newpath);
static void    Rewinddir(DIR *dirp);
static int     Rmdir(const char *path);
static void    Seekdir(DIR *dirp, long loc);
static int     Stat(const char *path, struct stat *buf);
static int     Statfs(const char *path, struct statfs *buf);
static int     Statvfs(const char *path, struct statvfs *buf);
static ssize_t Pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);
static long    Telldir(DIR *dirp);
static int     Truncate(const char *path, off_t offset);
static int     Unlink(const char *path);
static ssize_t Write(int fildes, const void *buf, size_t nbyte);
static ssize_t Write(int fildes, void *buf, size_t nbyte, off_t offset);
static ssize_t Writev(int fildes, const struct iovec *iov, int iovcnt);
```

Figure 3-2 Xrootd POSIX interface

We can use all the functions provided above for the implementation. Mainly we will use the related funcitons in Xrootd to implement the APIs in FUSE in order to make the function call passed smoothly through the file sytem.

3.5 Key techniques

3.5.1 The application of FUSE

In our implementation, FUSE plays a critical role. It provides a way for combining the Linux system calls with the functions handlers which are implemented by us to do the job as a filesytem.

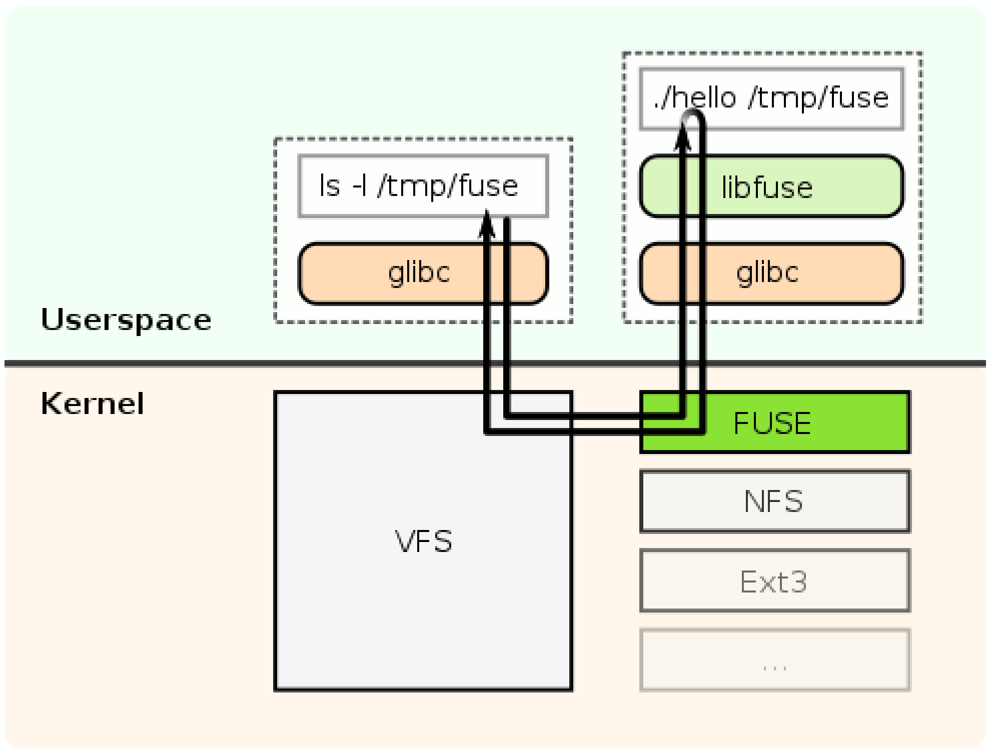


Figure 3-3 FUSE function call

FUSE is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially save data to and retrieve data from disk, virtual filesystems do not actually store data themselves. They act as a view or translation of an existing file system or storage device.

In principle, any resource available to a FUSE implementation can be exported as a file system. A FUSE filesystem is a program that listens on a socket for file operations to perform, and performs them. The FUSE library (libfuse) provides the communication with the socket, and passes the requests on to our code. It accomplishes this by using a "callback" mechanism. The callbacks are a set of functions you write to implement the file operations, and a struct fuse_operations containing pointers to them. In the case of CastorFS, the callback struct is named castorfs_oper. There are a total of 34 file operations defined in castorfsfs.c with pointers in castorfs_oper. The initialization uses a syntax that not everyone is familiar with; looking at a part of the initialization of the struct we see

```
struct fuse_operations castorfs_oper = {
    .getattr = castorfs_getattr,
    .readlink = castorfs_readlink,
    .open = castorfs_open,
    .read = castorfs_read
};
```

(This isn't the complete struct — just for explains how FUSE works) This indicates that castorfs_oper.getattr points to castorfs_getattr(), castorfs_oper.readlink points to castorfs_readlink(),castorfs_oper.open points to castorfs_open(), and castorfs_oper.read points to castorfs_read(). Each of these functions is my re-implementation of the corresponding filesystem function: when a user program calls read(), my castorfs_read() function ends up getting called. In general, what all of my reimplementations do is to log some information about the call, and then call the original system implementation of the operation on the underlyng filesystem.

When the function is called, it is passed two parameters: a file path (which is relative to the root of the mounted file system), and a pointer to a struct fuse_file_info which is used to maintain information about the file.

castorfs_open() starts by translating the relative path it was given to a full path in the underlying filesystem using my castorfs_fullpath() function. It then logs the full path, and the address of the fi pointer. It passes the call on down to the underlying fileystem, and sees if it was successful. If it was, it stores away the file descriptor returned by open() (so I'll be able to use it later), and returns 0. If it failed, it returns -errno. About the return value:

0 should be returned on success. This is the normal behavior for most of the calls in the libraries; exceptions are documented.

A negative return value denotes failure. If I return a value of -i, a -1 will be returned to the caller and errno is set to i. My castorfs_error() function looks up errno as set by the system open() call, logs the error, and returns -errno to this function so I can pass it to the user.

Notice that FUSE performs some translations. The open() system call is documented as returning a file descriptor (behavior I'm depending on), not 0 — so when my return is passed to the original caller, FUSE recognizes that I sent a 0 and returns an appropriate file descriptor (not necessarily the same one I got from my call to open()!). Meanwhile, I've got the underlying file open, and I've got its file descriptor in fi. Future calls to my code will include this pointer, so I'll be able to get the file descriptor and work with it. So... the user program has an open file in the mounted filesystem, and a file descriptor that it is keeping track of. Whenever that program tries to do anything with that file descriptor, the operation is intercepted by the kernel and sent to the castorfsfs program. Within my program, I also have a file open in the underlying directory, and a file descriptor. When the operation is sent to my program, I'll log it and then perform the same operation on my file.

To make this concrete, let's take a look at castorfs_read():

```
int castorfs_read(const char *path, char *buf, size_t size, off_t offset, struct
fuse_file_info *fi)
{
    int retstat = 0;
    log_msg("castorfs_read(path=\"%s\", buf=0x%08x, size=%d, offset=%lld,
fi=0x%08x)\n",
    path,   (int) buf, size,   offset,   (int) fi);
    retstat = pread(fi->fh, buf, size, offset);
    if (retstat < 0)
    retstat = castorfs_error("castorfs_read read");
    return retstat;
}
```

This function allows us to read data from some specified offset from the beginning of a file (so it corresponds more directly to the pread() function than to read()).

The main thing to point out about this function is that I use my file descriptor, which I put in fi when I opened the file, to read it. Also, if I get a non-error return from pread(), I pass this value up to the caller. In this case FUSE doesn't perform any translations, it just returns the value I gave it. To return an error, I use the same technique as in castorfs_open().FUSE provides a mechanism to place entries in a directory structure. The directory structure itself is opaque, so the basic mechanism is to create the data and call a FUSE-supplied function to put it in the structure.

When our readdir() callback is invoked, one of the parameters is a function called filler(). The purpose of this function is to insert directory entries into the directory structure, which is also passed to our callback as buf.

filler()'s prototype looks like this:

```
int fuse_fill_dir_t(void *buf, const char *name,
                    const struct stat *stbuf, off_t off);
```

You insert an entry into buf (the same buffer that is passed to readdir()) by calling filler() with the filename and optionally a pointer to a struct stat containing the file type.

castorfs_readdir() uses filler() in as simple a way as possible to just copy the underlying directory's filenames into the mounted directory. Notice that the offset passed to castorfs_readdir() is ignored, and an offset of 0 is passed to filler(). This tells filler() to manage the offsets into the directory structure for itself. Here's the code:

```
int castorfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t
offset, struct fuse_file_info *fi)
{
    int retstat = 0;
    DIR *dp;
    struct dirent *de;

    log_msg("castorfs_readdir(path=\"%s\", buf=0x%08x, filler=0x%08x,
offset=%lld, fi=0x%08x)\n",
            path, (int) buf, (int) filler,  offset, (int) fi);

    dp = (DIR *) (uintptr_t) fi->fh;
    de = readdir(dp);
    if (de == 0)
        return -errno;
    do {
        log_msg("calling filler with name %s\n", de->d_name);
        if (filler(buf, de->d_name, NULL, 0) != 0)
            return -ENOMEM;
    } while ((de = readdir(dp)) != NULL);
    log_fi(fi);
    return retstat;
}
```

By default, FUSE runs multi-threaded: this means (in brief) that a second request can be handled by the filesystem before an earlier request has completed;

this in turn raises the possibility that different threads can be simultaneously modifying a single data structure, which will cause very difficult-to-debug bugs.

There are a couple of things that can be done about the problem:

If the filesystem is executed with the -s option, it is run single-threaded. this eliminates the problem, at a cost in performance -- frankly, given the nature and intent of many fuse filesystems, it seems to me like the default should be single-threaded and multi-threaded should require an option. But I didn't write it, so it's not my call.

We can analyse our code for critical sections, and insert the normal syncronization primitives (such as semaphores) to ensure no dangerous races occur. Of course, there are several places where FUSE translates a single call into a sequence of calls to our functions; I haven't investigated whether FUSE takes any steps to ensure the atomicity of these calls. If it doesn't (and I suspect that's the case; trying to do so in any meaningful way in the absence of knowledge of the data we're exposing through our filesystem seems somewhere between difficult and impossible to me), then trying to do it seems really, really hard.

Note that even if we do make our filesystem single-threaded, that doesn't guard against access to the underlying data structures through some other means. Taking BBFS as an example:

We can have a single underlying directory mounted through two different mountpoints by using two invocations of bbfs.

A directory that has a BBFS filesystem mounted on top of it is still accessible to normal filesystem operations.

Either of these facts is sufficient to completely negate any efforts made in our filesystem to guard atomicity.

We should note that the FUSE code itself is careful about locking its own code and data structures. So far as we know, dangerous race conditions won't occur outside of our code.

3.5.2 The application of Xrootd

Xrootd serves a number of local directories to the network in a unified namespace. Files under Xrootd are accessed via a URL like: `root://SERVERNAME//PATH/TO/FILE.root.` There is a library that will allow the standard POSIX to "see" xrootd space. You can use it from the command line by setting two variables: `LD_PRELOAD=$ROOTSYS/lib/libXrdPosixPreload.so` `XROOTD_VMP=daya0001:/xrootd/.` All the functions in the Xrootd library could be called directly in our program.

3.5.3 The application of C++ standard library

The Standard Template Library, or STL, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template.

Like many class libraries, the STL includes container classes: classes whose purpose is to contain other objects. The STL includes the classes vector, list, deque, set, multiset, map, multimap, hash_set, hash_multiset, hash_map, and hash_multimap. Each of these classes is a template, and can be instantiated to contain any type of object.

Iterators are the mechanism that makes it possible to decouple algorithms from containers: algorithms are templates, and are parameterized by the type of iterator, so they are not restricted to a single type of container. Concepts are not a part of the C++ language; there is no way to declare a concept in a program, or to declare that a particular type is a model of a concept. Nevertheless, concepts are an extremely important part of the STL. Using concepts makes it possible to write programs that cleanly separate interface from implementation: the author of find only has to consider the interface specified by the concept Input Iterator, rather than the implementation of every possible type that conforms to that concept. Similarly, if we want to use find, we need only to ensure that the arguments you pass to it are models of Input Iterator. This is the reason why find and reverse can be used with lists, vectors, C arrays, and many other types: programming in terms of concepts, rather than in terms of specific types, makes it possible to reuse software components and to combine components together.

3.5.3 The application of LRU algorithm for caching

STL users needing an LRU-replacement cache generally gravitate towards std::map, because of its good support for keyed value accesses (O(log n) access complexity). The problem then is how to implement the eviction strategy.

The most obvious naive solution is to use a

```
std::map<K,std::pair<timestamp_type,V> >
```

The timestamp_t holds a scalar quantity, the ordering of which indicates when the value was last accessed relative to other values; typically some sort of incrementing serial number is used rather than an actual clock-derived time, to ensure a one-to-one mapping from timestamps to records. Keys then give O(log n) access to values and timestamps, and timestamps can be updated without the need to adjust the map's tree-structure (as this depends entirely on the key values). However, to determine the minimum timestamp in order to evict a record, it is necessary to perform a O(n) search over all the records to determine the oldest one.

As a solution to eviction being expensive, it might be tempting to implement the cache as a std::list<std::pair<K,V> >

Moving any item accessed to the tail of the list (a cheap operation for lists), ensures the least-recently-used item can trivially be obtained (for erasure) at the list head by begin(). However, it is now necessary to resort to a O(n) search simply to look up a key in the cache.

While either naive solution can be got to work (and may well be a simple pragmatic solution to caching a few tens of items) certainly neither can be considered scalable due to the O(n) behaviour associated with either identifying eviction targets or accessing values by key.

It is possible to implement an LRU-replacement cache with O(log n) eviction and access using a pair of STL maps:

```
typedef std::map<timestamp_type,K> timestamp_to_key_type;
typedef std::map<
   K,
   std::pair<V,timestamp_type>
> key_to_value_type;
```

On accessing key_to_value by a key, we obtain access to both the value required and the timestamp, which can be updated in both the accessed record and, by lookup, timestamp_to_key. When an eviction is required, the lowest timestamp in timestamp_to_key provides the key to the record which must be erased.

Pedants will observe that further slight improvement would also have the timestamp_to_key_type map’s value be an iterator into the key_to_value_type but this introduces a circular type definition. It might be tempting to try to break the dependency by using void in place of the iterator, but iterators cannot portably be cast to pointers. In any case, the first iterator optimization mentioned benefits the updating of timestamps needed during cache hits whereas this second iterator optimization benefits the eviction associated with cache misses. Since in a well functioning cached system cache hits should be far more common than misses, this second optimisation is likely of much less value than the first. Another consideration is that whatever expensive operation is required to generate a new result following a cache miss is likely to be hugely more expensive than any O(logn) access to the cache. Therefore this second optimisation is not considered further.

In fact there is one final powerful optimisation possible. The only operations actually done on timestamp_to_key are to access its head (lowest timestamp, least recently used) element, or to move elements to the (most recently used) tail. Therefore it can be replaced by a std::list; this also eliminates the need for any actual instances of timestamp_type (and therefore any concerns about the timestamp possibly overflowing). A list-and-map implementation is almost twice as fast as a version (not shown) using a pair of maps. See Listing 1 for a complete example using typedef std::list<K> key_tracker_type;

```
      typedef std::map<
        K,
      std::pair<V,key_tracker_type::iterator>
    > key_to_value_type;
```

3.5.3 The mechanism of authentication in CERN and Xrootd

CERN Authentication main goal is to provide a Single Sign On (SSO) solution for CERN Web Applications.

The current CERN Authentication SSO solution allows people to authenticate on a Web Site, i.e. EDH, and then re-use the same authentication to use another Web application, i.e. WinServices, without entering again the credentials. The main goal is to make things easier for the user. For years, every CERN application handled its own user database, because no real central solution was provided for authentication mecanisms. This lowered dramatically the user experience when accessing to CERN applications, as different credential pairs had to be remembered: one login on AFS, another on Mail, a third one in AIS, and different passwords everywhere.

The usual workaround was to write the credentials on a small yellow paper and stick it on the screen, or under the keyboard for more security.

With the CERN Authentication solution, users have only one login and password pair to remember. If any security problem occurs on the account, a simple click can disable it, blocking instantly all CERN Applications access.

For the xrootd, it provides flexible security architecture which includes multiple protocols which garantie the easily expandable features and simultaneous heterogeneous protocols which allows multiple administrative domains to be given.

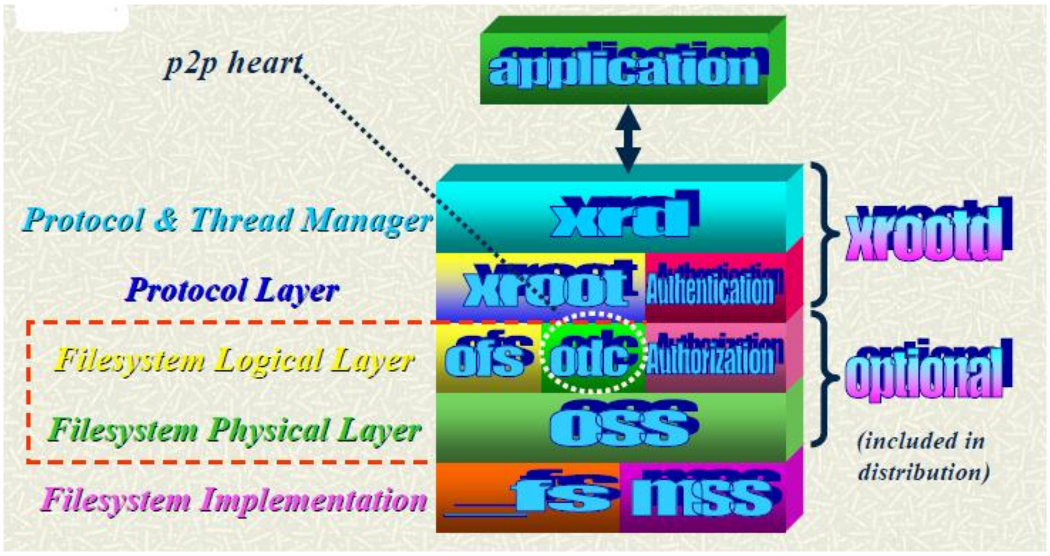


Figure 3-4 Xrootd server architecture

The authentication and authorization are developed as runtime plug-in componets, so they could be easily substituted and trivial to extend. At the same time, client/server architecture plugin will make the other application layer architecture portable.

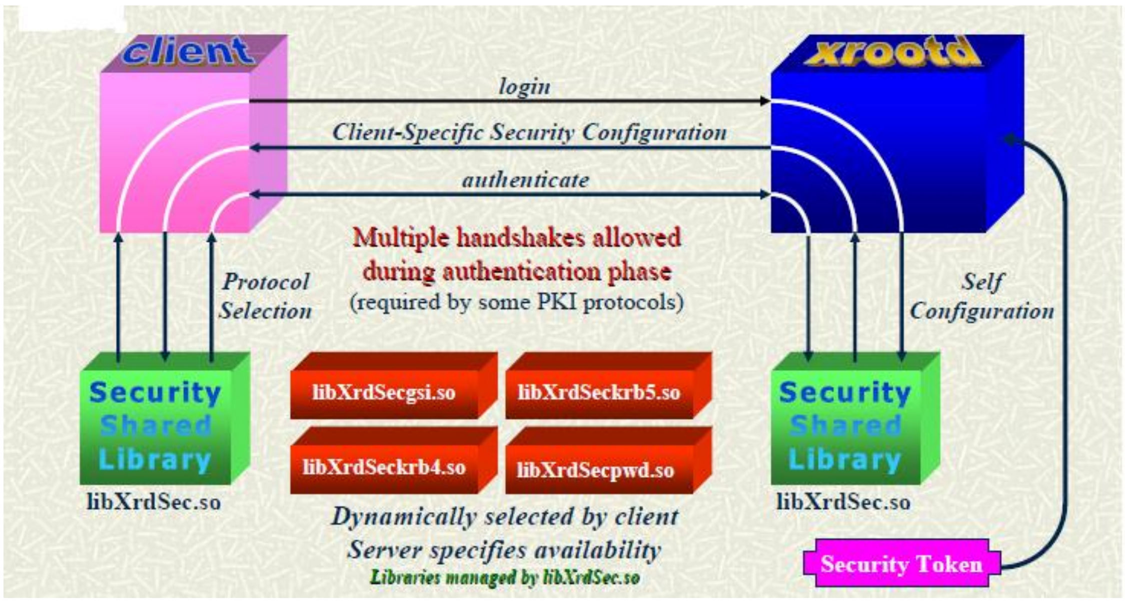

client
xrootd
login
Client-Specific Security Configuration
authenticate
Multiple handshakes allowed
during authentication phase
(required by some PKI protocols)
Protocol
Selection
Self
Configuration
Security
Shared
Library
libXrdSec.so
libXrdSecgsi.so
libXrdSeckrb5.so
libXrdSeckrb4.so
libXrdSecpwd.so
Security
Shared
Library
libXrdSec.so
Dynamically selected by client
Server specifies availability
Libraries managed by libXrdSec.so
Security Token

Figure 3-5 Xrootd security architecture

The xrootd-implementation in dCache includes a pluggable authentication framework. To control which authentication mechanism is used by xrootd, add the xrootdAuthNPlugin option to our dCache configuration and set it to the desired value.

The previously explained methods to restrict access via xrootd can also be used together. The precedence applied in that case is as following: The permission check executed by the authorization plugin (if one is installed) is given the lowest priority, because it can controlled by a remote party. E.g. in the case of token based authorization, access control is determined by the file catalogue (global namespace). The same argument holds for many strong authentication mechanisms - for example, both the GSI protocol as well as the Kerberos protocols require trust in remote authorities. However, this only affects user *authentication*, while authorization decisions can be adjusted by local site administrators by adapting the gPlazma configuration. To allow local site’s administrators to override remote security settings, write access can be further restricted to few directories (based on the local namespace, the pnfs). Setting xrootd access to read-only has the highest priority, overriding all other settings.

3.5.5 The application of CMake and RPMBuild tools

The principal benefit of open source software is, as its name implies, access to the inner workings of an application. Given the source, we can study how an application works; change, improve, and extend its operation; borrow and repurpose code (per the limits of the application's license); and port the application to novel and emergent platforms.

However, such liberal access is not always wanted. For instance, a user may not want the onus of building from source code. Instead, he or she may simply want to install the software much like a traditional "shrink-wrapped" application: insert media, run setup, answer a few prompts, and go. Indeed, for most computer users, such pre-built software is preferred. Pre-built code is less sensitive to system vagaries and thus more uniform and predictable.

In general, a pre-built, open source application is called a package and bundles all the binary, data, and configuration files required to run the application. A package also includes all the steps required to deploy the application on a system, typically in the form of a script. The script might generate data, start and stop system services, or manipulate files and directories. A script might also perform operations to upgrade existing software to a new version.

Because each operating system has its idiosyncrasies, a package is typically tailored to a specific system. Moreover, each operating system provides its own package manager, a special utility to add and remove packages from the system. For example, Debian Linux-based systems use the Advanced Package Tool (APT), while Fedora Linux systems use the RPM Package Manager. The package manager precludes partial and faulty installations and "uninstalls" by adding and removing the files in a package atomically. The package manager also maintains a manifest of all packages installed on the system and can validate the existence of prerequisites and co-requisites beforehand.

If you're a software developer or a systems administrator, providing your application as a package makes installations, upgrades, and maintenance much easier. Here, you learn how to use the popular RPM Package Manager to bundle a utility. For purposes of demonstration, you'll bundle the networking utility wget, which downloads files from the Internet. The wget utility is useful but isn't commonly found standard in distributions. (An analog, curl, is often included in distributions.) Be aware that you can use RPM to distribute most anything—scripts, documentation, and data—and perform nearly any maintenance task.

CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. CMake is quite sophisticated: it is possible to support complex environments requiring system configuration, pre-processor generation, code generation, and template instantiation. CMake is controlled by writing instructions in CMakeLists.txt files. Each directory in your project should have a CMakeLists.txt file. What is nice about CMake is that CMakeLists.txt files in a sub-directory inherit properties set in the parent directory, reducing the amount of code duplication. For our sample project, we only have one subdirectory: w01-cpp. The CMakeLists.txt file for the top-level cmake directory is pretty simple but demonstrates a few key features.

3.6 Brief summary

In this part, we introduced the system design which includes the overall design of the system, the design of the wrapper, the design of the caching and the design for the new CastorFS which will use xrootd protocol for the implementation. The key technologies are introduced in detail. The actual implementation will be given in the next chapter.

Chapter 4 System Implementation and Testing

Following the phase of the requirement analysis and general design, the technical solution has been set up, according to the design of system, we can perform the work of implementation.

4.1 The environment of system implementation

4.1.1 Hardware environment

Since we develop the system firstly on the personal computer, and then migrate the system to the lxplus and the compluting cluser in LHCb, we will see the hardware environment respectively.

(1) PC Hardware configuration:

There are 2 processors in the PC and each one has an Intel(R) Pentium(R) 4 CPU 2.80GHz processor with 512KB cache.

(2) LHCb computing cluster

There are 8 processors in the PC and each one has an Intel(R) Xeon(R) CPU E5410 @ 2.33GHz. Each processor has 6144 KB cache. Each plus node will use 8G memory.

Figure 4-1 LHCb plus cluster

4.1.2 Software environment

The system will be implemented based on the SLC6 (Linux localhost.localdomain 2.6.32-71.29.1.el6.i686 #1 SMP Tue May 10 17:35:05 CDT 2011 i686 i686 i386 GNU/Linux). Linux version 2.6.32-71.29.1.el6.i686 (mockbuild@sl6.fnal.gov) with the gcc version 4.4.4 20100726 (Red Hat 4.4.4-13) (GCC)) #1 SMP Tue May 10 17:35:05 CDT 2011.

Scientific Linux CERN 6 will be a Linux distribution build within the framework of Scientific Linux which in turn is rebuilt from the freely available Red Hat Enterprise Linux 6 (Server) product sources under terms and conditions of the Red Hat EULA. Scientific Linux CERN is built to integrate into the CERN computing environment but it is not a site-specific product: all CERN site customizations are optional and can be deactivated for external users.

There are some packages which should be installed first, the FUSE pakages:

(1) fuse-libs-2.8.3-1.el6.i686

(2) gvfs-fuse-1.4.3-9.el6.i686

(3) fuse-devel-2.8.3-1.el6.i686

The Xrootd Packages:

(1) xrootd-server-devel-3.0.4-1.el6.i686

(2) xrootd-libs-3.0.4-1.el6.i686

(3) xrootd-doc-3.0.3-2.el6.noarch

(4) xrootd-server-3.0.4-1.el6.i686

(5) xrootd-libs-devel-3.0.4-1.el6.i686

(6) xrootd-client-3.0.4-1.el6.i686

(7) xrootd-client-devel-3.0.4-1.el6.i686

The other related software packages should also be installed on the machine.

The CERN SVN will be used to manage the code version. **Subversion** is a version control system that is widely used by many Open Source projects such as Apache and GCC. Subversion started as a project to implement features missing in CVS. Some of these features are: (1) Subversion tracks structure of folders. CVS doesn't have the concept of folders. (2) Subversion has a global revision number for the whole repository. CVS tracks each file individually. A commit that represents one logical change to the project code may change a group of files; in Subversion, this commit will have one revision number instead of separate revision numbers for every changed file in CVS. (3) Subversion commits are atomic. (4) Subversion retains the revision history of moved or copied files. (5) Subversion commands are very similar to CVS. It's very easy to switch for CVS users. Most of the time, it's a matter of replacing cvs with svn.

The Central SVN Service is accessible only for CERN registered computer users. After each modification of the code, it will be stored and shared in the system.

4.1.3 The implementation of wrapper

We have two file to perform the wraping. The wrap.hh declares all the functions which will be used to hook up with the FUSE APIs to perform the real operations in the filesystem. In wrap.cc, the functions in C++ will be invoked to implement the C functions. We will be able to compile it respectively using gcc and g++. After we get the related .o object files, we are able to link them together to make a virtual filesystem work.

We can see the code as follow.

```
#include "wrap.hh"
#include "castorfs.hh"
int init_castorfs(int argc, char* argv[]){
        return initCastorFS(argc, argv);
}
void set_cfuseoper(struct fuse_operations* oper_pointer){
        setCfuseoper(oper_pointer);
}
int wrap_getattr(const char *path, struct stat *statbuf) {
          return CastorFS::Instance()->Get_attr(path, statbuf);
}
int wrap_mkdir(const char *path, mode_t mode) {
          return CastorFS::Instance()->Mkdir(path, mode);
}
int wrap_rmdir(const char *path) {
          return CastorFS::Instance()->Rmdir(path);
}
int wrap_mknod(const char* path, mode_t mode, dev_t rdev){
        return CastorFS::Instance()->Mknod(path, mode, rdev);
```

```
    }
    int wrap_unlink(const char* path){
            return CastorFS::Instance()->Unlink(path);
    }
    int wrap_chown(const char *path, uid_t uid, gid_t gid) {
              return CastorFS::Instance()->Chown(path, uid, gid);
    }
    int wrap_truncate(const char *path, off_t newSize) {
              return CastorFS::Instance()->Truncate(path, newSize);
    }
    int wrap_utimens(const char *path, const struct timespec ts[2]) {
              return CastorFS::Instance()->Utimens(path, ts);
    }
    int wrap_open(const char *path, struct fuse_file_info *fileInfo) {
              return CastorFS::Instance()->Open(path, fileInfo);
    }
    int wrap_read(const char *path, char *buf, size_t size, off_t offset, struct
fuse_file_info *fileInfo) {
              return CastorFS::Instance()->Read(path, buf, size, offset, fileInfo);
    }
    int wrap_write(const char *path, const char *buf, size_t size, off_t offset, struct
fuse_file_info *fileInfo) {
              return CastorFS::Instance()->Write(path, buf, size, offset, fileInfo);
    }
    int wrap_release(const char *path, struct fuse_file_info *fileInfo) {
              return CastorFS::Instance()->Release(path, fileInfo);
    }
    int wrap_getxattr(const char *path, const char *name, char *value, size_t size)
   {
              return CastorFS::Instance()->Getxattr(path, name, value, size);
    }
    int wrap_removexattr(const char *path, const char *name) {
              return CastorFS::Instance()->Removexattr(path, name);
    }
```

```
int wrap_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fileInfo) {
        return CastorFS::Instance()->Readdir(path, buf, filler, offset, fileInfo);
}
...
```

4.1.4 The implementation of caching

We use C++ standard library to implement caching. We use a template - template <typename Key, typename Value> class to provide an implementation for providing the application for a series of the application of the least recently used caching mechanism.

In our design and implementation, we will record the already retrieved meta-data for each entry of the files in Castor. The key will be the fullpath of one file and the value will be the related meta-data, normally a structure (struct stat). The reason we do that is based on the fact that the load on the server side could be very high and we can ease the burden on the server side by storing the most probablly retrieved data in the cache to provide a local storage of the data.

Here we will see the basic data structure which is used in the class and then we will see the logic inside the implementation.

```
(1) typedef std::map<
        K,
        std::pair<std::pair<V,typename key_tracker_type::iterator>,int>
    > key_to_value_type;
```

First, we define this map container to hold all the key and value pairs. From each key, we can find the related pair which is composed anthother pair. The int type of value will be used to record the system time at when the pair is stored in the cache. Then we will be able to set a time limitation for distinguishing the expired data. The std::pair<V,typename key_tracker_type::iterator> is used to build a relationship between the value and iterator in order we can trace back from the list which will be introduced as follow to the map container.

(2) typedef std::list<K> key_tracker_type: This is a list which will be used to track the key. Each time, when we use retrieved data in the cache. We will ajust the sequence of this list to be able to put the least recently used data in the end of the

list. Therefore, when the number of the caching reaches to its limitation, we can evict the head of the list to get extra space for storing the new data.

(3) int (* _fn)(const char*, V*): This is defined as a function which will be used to retrieve the related data if we can’t find them in the cache.

(4) time_t timer: this will be used to define the expiring time for the cache. Since we will not be able to predict the time of modifications of the files in a system, we need to set a expiring time for the cached data.

(5) const size_t _capacity: this will be used to indicate the number of information entries we can store in the cache. Since the meta-data will be cached in the system, the capacity of all the data will not be very large, so we should take full advantage of that. In other words, we can set the _capacity a big one.

(6) key_tracker_type _key_tracker; this is a list to track a sequence of the key data in the map. It will be changed dynamicly according to the accessing history of the map container.

(7) key_to_value_type _key_to_value; this is a map container. All the information will be hold by that. We will perform many operation based on that.

The implementation flow chart is shown in figure 4-2.

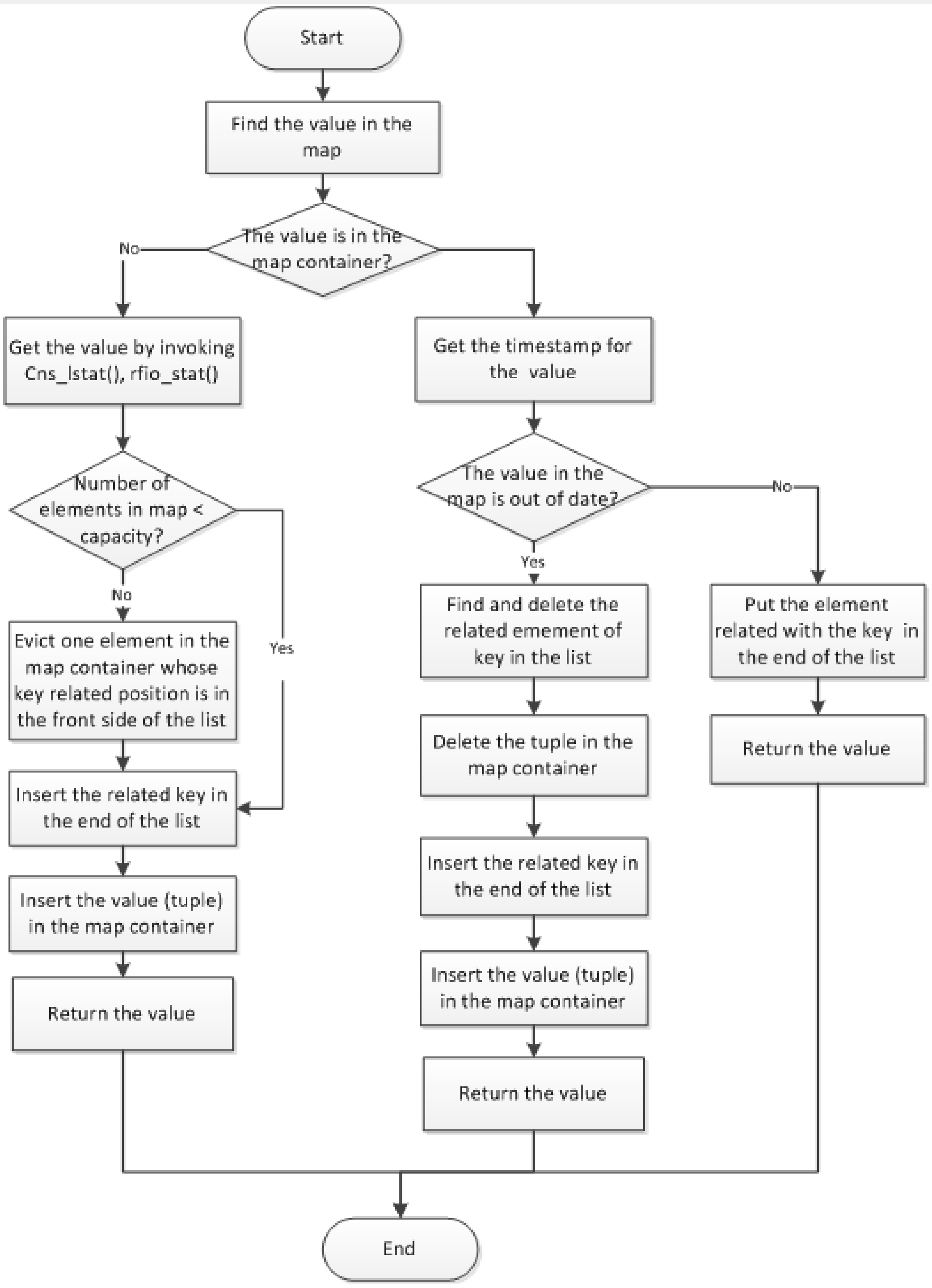

Start
Find the value in the map
The value is in the map container?
No
Get the value by invoking Cns_lstat(), rfio_stat()
Get the timestamp for the value
Number of elements in map < capacity?
No
Yes
Evict one element in the map container whose key related position is in the front side of the list
Insert the related key in the end of the list
Insert the value (tuple) in the map container
Return the value
The value in the map is out of date?
Yes
No
Find and delete the related emement of key in the list
Delete the tuple in the map container
Insert the related key in the end of the list
Insert the value (tuple) in the map container
Return the value
Put the element related with the key in the end of the list
Return the value
End

Figure 4-2 Cache mechanism

For the caching, when we want to retrieve the meta-data in the cache, first we will search the map to find the data.

const typename key_to_value_type::iterator it =_key_to_value.find(k)

We will use the “find” method provided in the algorithm.h to perform this operation. If we can’t find the value which we want, we will invoke the related function to get the data and store them in the form of key, value, time which indicate the time point at when they are retrieved. At the same time, the retrived value will be returned to the function which actually handles the system call. If we can find the key and the related value in the cache, it will be returned to the function which called for the information of that entry. At that moment, the key of that entry in the key list will be moved to the end of the list. With the growing of the number of the data in the cache, we will meet the limitation of the cache. In that situation, if we retrieve the information of a new entry and we want to store the related meta-data in the map, we need to evict the least used element whose position is indicated in the head of the key list. By doing that, we implemented the caching.

4.1.5 The implementation of new CastorFS with Xrootd

(1) The initialization of FUSE

The fuse library provides support for analysing parameters passed from the command line. It is desirable to use this since fuse_main itself needs command line parameters and it is best that the user get a consistent interface.

The basic idea seems to be that the file system calls a fuse library routine to parse the parameters, classify them and (depending on classification) call back to the file system to let it action those parameters which it is interested in. It also assembles a modified list of parameters to be passed into the fuse_main interface. The parameter list is expected to have the following form:

[-ooption[,option]*] [-flag]* [-key[ ]|=value]* [filesysargument]* mountpoint -- anything

flag and key can be any string other than o or - followed by space. Options, flags and keys may be in any order preceding the fixed parameters.

Initiating parameter processing: int fuse_opt_parse(struct fuse_args *args, void *data, const struct fuse_opt opts[], fuse_opt_proc_t proc) should be called to initiate the process of parameter analysis. “args” is a structure that initially should contain the input args and argument count (there is a macro for defining this (Q.V.)). On return it contains the output args list to be passed to fuse_main. “data” is a pointer to any object the file system requires it to be (it may be NULL) it is passed into the call back procedure but it can also be used as a pointer to an area to receive values from certain types of argument. “opts” is a row of templates describing the options available (again there are macros to assist in defining this (Q.V.)). “proc” is a call back procedure called by fuse_opt_parse as it processes the parameters. Parameter analysis callback procedure

typedef int (*fuse_opt_proc_t)(void *data, const char *arg, int key,struct fuse_args *outargs);

For certain options (determined by values in the opts array) the call back procedure is called with:

* data (the pointer passed into fuse_opt_parse).

* arg the argument or option in question. (Some processing is done so that -x yz becomes -xyz note -oxyz is a special case and becomes xyz).

* key the value used in declaring the option or FUSE_OPT_KEY_NONOPT (for items not in the form of an option) or FUSE_OPT_KEY_OPT (for options items not matching any option).

* The output argument list is the current output arguments.

The call back routine should reply -1 on error, 0 to discard the argument (presumably having processed it in some way), 1 to retain it so that it will be passed to fuse_main. The call back procedure can also add arguments to the argument array.

int fuse_opt_add_arg(struct fuse_args *args, const char *arg) - can be called by the option call back procedure to add an argument to the output arguments for fuse_main. The add_arg procedure takes the outargs parameter passed into the call back procedure and a string containing the new argument. An instance of the use of this is shown in the passfs example where the -m flag forces foreground processing by appending -f to the argument list.

Adding an extra option to the output option list - int fuse_opt_add_opt(char **opts, const char *opt); this is meant to be called by the option call back procedure to add an option to the comma separated list of output options for fuse_main but the semantics are unclear as it is not clear where the **opts parameter would 'come from'. The procedure can take a null opts parameter and a string containing the new option. <to do: further research>. Initialising the fuse_args structure - a macro procedure is provided to initialise the structure: FUSE_ARGS_INIT(argc, argv) where argc and argv are the corresponding values from the main() procedure of the file system as in:

```
int main(int argc, char *argv[]) {
struct fuse_args args = FUSE_ARGS_INIT(argc, argv);
  ...
}
```

Setting up the fuse_opts array - the fuse opts array is used to define a template and a key for each parameter. The key is used to identify the parameter in calls to the parameter analysis call back. Two macro procedures are provided to assist in setting up the array: FUSE_OPT_KEY(templ, key) and FUSE_OPT_END as in

```
static struct fuse_opt passfs_opts[] = {
FUSE_OPT_KEY("--help", KEY_HELP),
FUSE_OPT_KEY("--version", KEY_VERSION),
FUSE_OPT_KEY("-h", KEY_HELP),
FUSE_OPT_KEY("-V", KEY_VERSION),
FUSE_OPT_KEY("stats", KEY_STATS),
FUSE_OPT_KEY("-log=",KEY_LOGFILE),
FUSE_OPT_KEY("-root ",KEY_ROOT),
FUSE_OPT_END
};
```

The key parameter needs to be a positive integer that uniquely identifies the particular option (typical generated as part of an enum). Thus in the example above --version and -V are synonyms.

Form of the template parameter in the fuse_opt structure

This is quite complex and it shoul be noted that the fuse_opt.h file documents other forms than can be created with the FUSE_OPT_KEY macro. Ignoring these features the possible forms are:

1. "-string" (where string can be anything provided it doesn't start with o or consist of only - or contain =). This is a flag type parameter with no value.

2. "string" (where string can be anything provided it doesn't start with - or contain =). This will match one of a list of options after -o.

3. "-string=" as 1 except that a value is expected to follow the =.

4. "string=" as 2 except that a value is expected to follow the =.

5. "-string " as 1 except that a parameter is expected after the string. Note that this matches both -stringvalue and -string value (the intervening space is ignored).

The additional features provide for formatted templates with either %s or%lu appearing after the = or space. In this case the value (a pointer to a string or an unsigned integer) is stored at an offset relative to the data parameter passed into fuse_opt_parse. The offset value is held in the matching fuse_opt structure of the fuse_opt array. The call back procedure is not called.

The forms "-string=" and "-string " are different, even though they both mean that a parameter value follows. If you want the user to be able to use either form and provide both templates then you need to be aware that both templates may be matched and your call back procedure called twice. Thus for instance if the user supplies -string= then your routine will be entered once with =value and once with value

(2) Initialize FUSE account

We use getgrnam_r("fuse",&fuse_group,buf,bufsize,&pfuse_group) to get a pointer to a structure containing the broken-out fields of the record in the group database (e.g., the local group file /etc/group, NIS, and LDAP) that matches the group name name.

We use setgroups(1,&fuse_group.gr_gid) to set the supplementary group IDs for the process. Function fuse_main() is for the lazy. This is all that has to be called from the main() function. This function does the following: 1) parses command line options (-d -s and -h) 2) passes relevant mount options to the fuse_mount() 3) installs signal handlers for INT, HUP, TERM and PIPE 4) registers an exit handler to unmount the filesystem on program exit 5) creates a fuse handle 6) registers the operations calls either the single-threaded or the multi-threaded event loop

(3) The function of CastorFS::GetAttr(const char* , struct stat*) is used for retrieving the meta-data of the the certain files. Before we use the xrootd function to get the data, first we need to get the full path of a file. Since all the files are managed in the unified way of root://castorlhcb//castor, we need to add the URL before each path we want to use. In this function, we will deal directly with the cache, the function calls will not be done in the function but in the cache. The related function in xrootd will be XrdPosixXrootd::Stat(char*, struct stat*). It will return all the information about one directory in the form of *struct stat*.

(4) The function of int CastorFS::Readdir(const char* path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi) is used for get all the entries of one directory. The readdir implementation keeps track of the offsets of the

directory entries. It uses the offset parameter and always passes non-zero offset to the filler function. When the buffer is full (or an error happens) the filler function will return '1'. The function typedef int(* fuse_fill_dir_t)(void *buf, const char *name, const struct stat *stbuf, off_t off) is defined by FUSE to add an entry in a readdir() operation. One DIR pointer will be used to point to xrdPosixXrootd::Opendir(FULLPATH). Then the XrdPosixXrootd::Readdir(dp) should be called to get the related file information organized with struct dirent.
struct dirent { long d_ino; off_t d_off; unsigned short d_reclen; char d_name; }. In the end of the implementation of the function, XrdPosixXrootd::Closedir(dp) will be called to make sure the directory will be safely closed.

(5) int CastorFS::Mknod(const char* path, mode_t mode, dev_t rdev) is used to create a new node (file) for the filesystem. This function will be implemented by using XrdPosixXrootd::Open(rootpath, O_CREAT | O_EXCL | O_WRONLY, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH). By provide special parameter for the Open function, it will perform the right function as we want to create a non-existance file. If the file exists, O_CREAT flag has no effect except as noted under O_EXCL below. Otherwise, the file shall be created; the user ID of the file shall be set to the effective user ID of the process; the group ID of the file shall be set to the group ID of the file's parent directory or to the effective group ID of the process; and the access permission bits (see <sys/stat.h>) of the file mode shall be set to the value of the third argument taken as type mode_t modified as follows: a bitwise AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. When bits other than the file permission bits are set, the effect is unspecified. The third argument does not affect whether the file is open for reading, writing, or for both. Implementations shall provide a way to initialize the file's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the file's group ID to the effective group ID of the calling process. If O_CREAT and O_EXCL are set, *open*() shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing *open*() naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and *path* names a symbolic link, *open*() shall fail

and set *errno* to [EEXIST], regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined. O_WRONLY is used for indicating that the file is open for writing only. S_IRUSR is for reading permission for owner. S_IWUSR is to give write permission to owner. S_IXUSR is to give execute/search permission to owner. S_IROTH is used for give read permission to the others.

(6) int CastorFS::Open(const char* path, struct fuse_file_info *fi) is implementd by invoking the function int fd = XrdPosixXrootd::Open(rootpath, fi->flags, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH). Here we use the same arguments as those in the last function for the third parameter to indicate the way we operate the files.

(7) int CastorFS::Read(const char* path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi). In this function, we mainly used int XrdPosixXrootd::Pread(fd, buf, size, offset) to fill the buf to read the file.

(8) int CastorFS::Write(const char* path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi). In this function, we use the xrdPosixXrootd::Pwrite(fd, buf, size, offset) to send the related parameters from the linux operating system to the Xrootd function.

(9) int CastorFS::Unlink(const char* path) deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse. If the name was the last link to a file but any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed. If the name referred to a symbolic link the link is removed. If the name referred to a socket, fifo or device the name for it is removed but processes which have the object open may continue to use it. Unlink function is used to delete one node and we use XrdPosixXrootd::Unlink(rootpath) directly to perform the certain task.

(10) int CastorFS::Mkdir(const char* path, mode_t mode) is used to create a directory. It is called when we use the Linux command to create a new directory. The related XrdPosixXrootd::Mkdir(rootpath, mode) is used to implement this function.

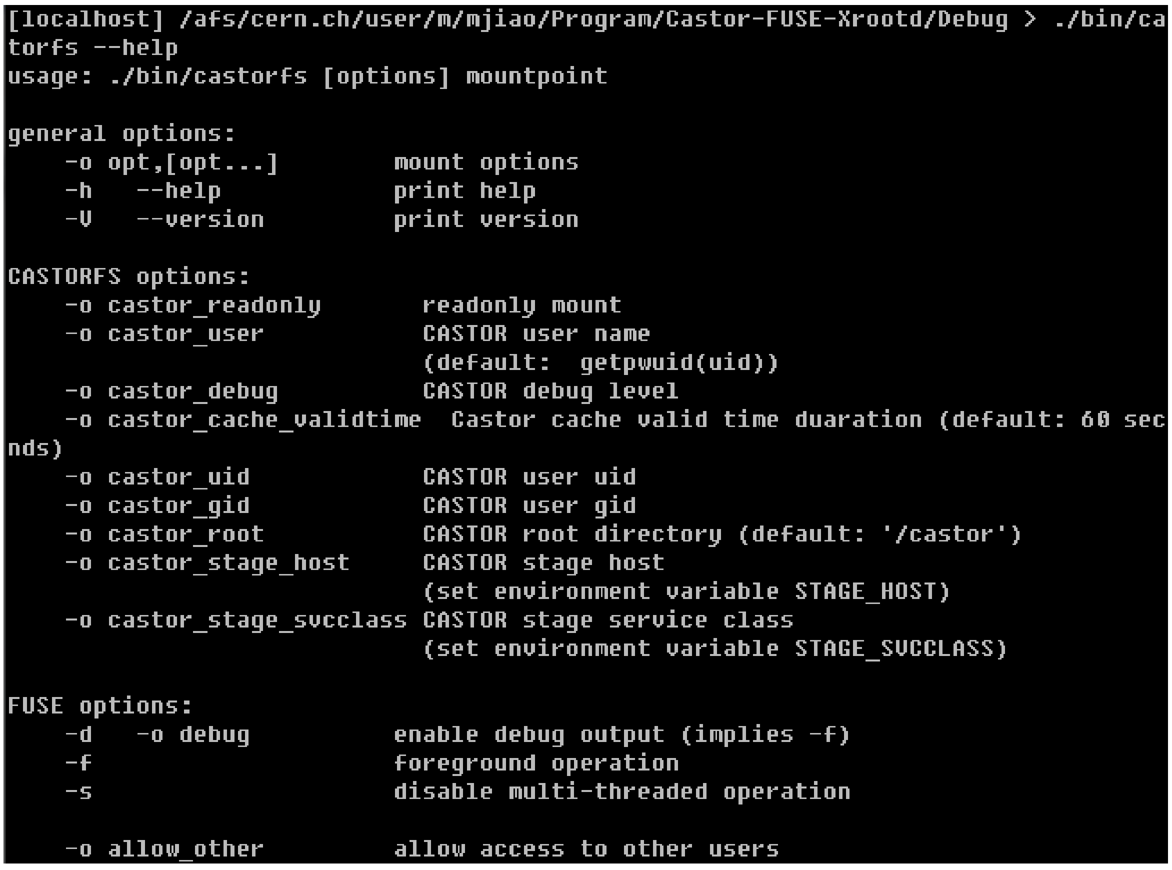
(11) int CastorFS::Rmdir(const char* path) is implemented to provide a function to rmove a empty directory and the XrdPosixXrootd::Rmdir(rootpath) is called in the function.

(12) int CastorFS::Truncate(const char* path, off_t size) causes the regular file named by path or referenced by fd to be truncated to a size of precisely length bytes. If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as zero bytes. The file pointer is not changed. If the size changed, then the ctime and mtime fields for the file are updated, and suid and sgid mode bits may be cleared.

(13) int CastorFS::Getxattr(const char *path, const char *name, char *value, size_t size) retrieves the value of the extended attribute identified by name and associated with the given path in the filesystem. The length of the attribute value is returned. And the XrdPosixXrootd::Getxattr(rootpath, name, xattr, size) is used for implement this function.

4.2 Key Interfaces of the software system

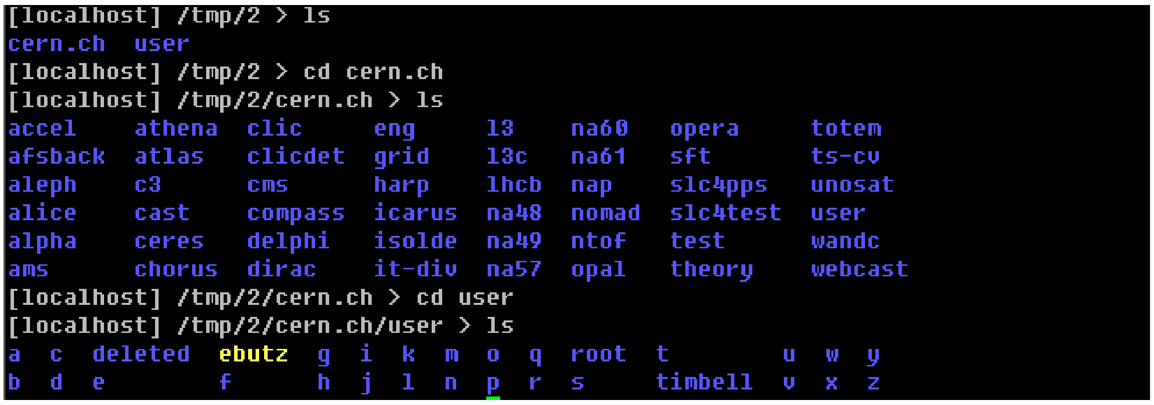
The first interface shows options for Castor filesystem and FUSE, we provide options for the filesystem and the cache valide time for the users to configure the file system.



```
[localhost] /afs/cern.ch/user/m/mjiao/Program/Castor-FUSE-Xrootd/Debug > ./bin/ca
torfs --help
usage: ./bin/castorfs [options] mountpoint

general options:
    -o opt,[opt...]        mount options
    -h   --help            print help
    -V   --version         print version

CASTORFS options:
    -o castor_readonly       readonly mount
    -o castor_user           CASTOR user name
                             (default:  getpwuid(uid))
    -o castor_debug          CASTOR debug level
    -o castor_cache_validtime  Castor cache valid time duaration (default: 60 sec
nds)
    -o castor_uid            CASTOR user uid
    -o castor_gid            CASTOR user gid
    -o castor_root           CASTOR root directory (default: '/castor')
    -o castor_stage_host     CASTOR stage host
                             (set environment variable STAGE_HOST)
    -o castor_stage_svcclass CASTOR stage service class
                             (set environment variable STAGE_SVCCLASS)

FUSE options:
    -d   -o debug          enable debug output (implies -f)
    -f                     foreground operation
    -s                     disable multi-threaded operation

    -o allow_other         allow access to other users
```

Figure 4-3 CastorFS options

The file system is fully integrated in the Linux system and the filesystem operations are the same as the standard filesystem (figure 4-4).



```
[localhost] /tmp/2 > ls
cern.ch  user
[localhost] /tmp/2 > cd cern.ch
[localhost] /tmp/2/cern.ch > ls
accel    athena  clic     eng     l3    na60   opera     totem
afsback  atlas   clicdet  grid    l3c   na61   sft       ts-cv
aleph    c3      cms      harp    lhcb  nap    slc4pps   unosat
alice    cast    compass  icarus  na48  nomad  slc4test  user
alpha    ceres   delphi   isolde  na49  ntof   test      wandc
ams      chorus  dirac    it-div  na57  opal   theory    webcast
[localhost] /tmp/2/cern.ch > cd user
[localhost] /tmp/2/cern.ch/user > ls
a  c  deleted  ebutz  g  i  k  m  o  q  root  t        u  w  y
b  d  e        f      h  j  l  n  p  r  s     timbell  v  x  z
```

Figure 4-4 CastorFS interface

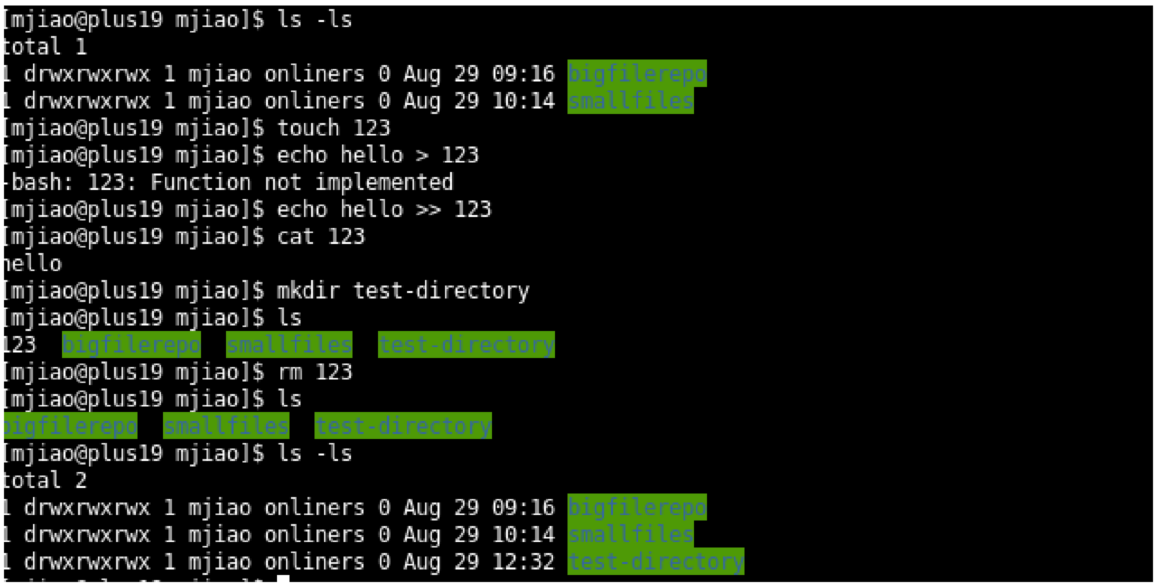
4.3 System Testing and Performance evaluation

4.3.1 System Testing

For the system testing, we performed the white box testing to check the program sequence during all the phases of system implementation. The white box testing is maily performed by checking the implementation of each virtual filesystem functions. Those function are: getattr(), readdir(), open(), write(), unlink(), mkdir(), rmdir(), truncate() and the functions implemented for the caching, the one that is to do the retrieving opration-the overloaded operater ().

After the accomplishment of the file system, we performed the black box testing to check all paths for the data flow and all the functions in the file system. We do the test by writing the related script for each function. At the same time we get the system performance result for the next section too. We go to the command line after the system is mounted and we perform the POSIX Linux filesystem operation functions:

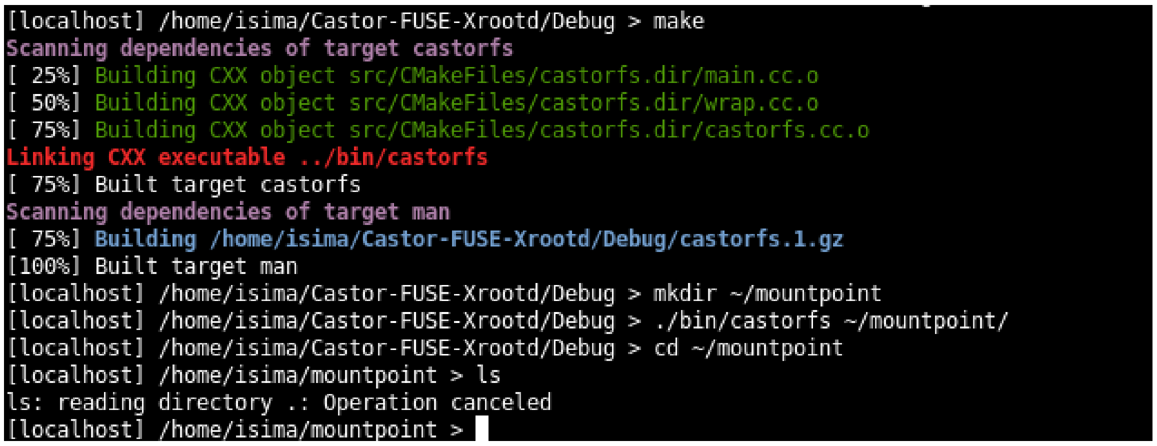
In this testing, we firstly get into mjiao directory and create a file 123 and then write "hello" to to the file and we also displayed the content of the file. After that, we created a directory calld "test-dierctory" and we can see that, it works as we expected.



```
[mjiao@plus19 mjiao]$ ls -ls
total 1
1 drwxrwxrwx 1 mjiao onliners 0 Aug 29 09:16 bigfilerepo
1 drwxrwxrwx 1 mjiao onliners 0 Aug 29 10:14 smallfiles
[mjiao@plus19 mjiao]$ touch 123
[mjiao@plus19 mjiao]$ echo hello > 123
-bash: 123: Function not implemented
[mjiao@plus19 mjiao]$ echo hello >> 123
[mjiao@plus19 mjiao]$ cat 123
hello
[mjiao@plus19 mjiao]$ mkdir test-directory
[mjiao@plus19 mjiao]$ ls
123  bigfilerepo  smallfiles  test-directory
[mjiao@plus19 mjiao]$ rm 123
[mjiao@plus19 mjiao]$ ls
bigfilerepo  smallfiles  test-directory
[mjiao@plus19 mjiao]$ ls -ls
total 2
1 drwxrwxrwx 1 mjiao onliners 0 Aug 29 09:16 bigfilerepo
1 drwxrwxrwx 1 mjiao onliners 0 Aug 29 10:14 smallfiles
1 drwxrwxrwx 1 mjiao onliners 0 Aug 29 12:32 test-directory
```

Figure 4-5 Test result

Here, we also did a test for the authentication mechanism. We put all the system implementation files to the isima machine which has no certification and we tried to mount the filesystem to mountpoint. Later, we tried to "ls" the folder of mountpoint and we can see that the operation was canceld due to the authorization policy provided by the xrootd.



```
[localhost] /home/isima/Castor-FUSE-Xrootd/Debug > make
Scanning dependencies of target castorfs
[ 25%] Building CXX object src/CMakeFiles/castorfs.dir/main.cc.o
[ 50%] Building CXX object src/CMakeFiles/castorfs.dir/wrap.cc.o
[ 75%] Building CXX object src/CMakeFiles/castorfs.dir/castorfs.cc.o
Linking CXX executable ../bin/castorfs
[ 75%] Built target castorfs
Scanning dependencies of target man
[ 75%] Building /home/isima/Castor-FUSE-Xrootd/Debug/castorfs.1.gz
[100%] Built target man
[localhost] /home/isima/Castor-FUSE-Xrootd/Debug > mkdir ~/mountpoint
[localhost] /home/isima/Castor-FUSE-Xrootd/Debug > ./bin/castorfs ~/mountpoint/
[localhost] /home/isima/Castor-FUSE-Xrootd/Debug > cd ~/mountpoint
[localhost] /home/isima/mountpoint > ls
ls: reading directory .: Operation canceled
[localhost] /home/isima/mountpoint >
```

Figure 4-6 Authentication test

4.3.2 Performance evaluation

(1) To retrieve the information of 2678 set of record in a directory, we did two types of performance evaluation against the 4 commands.

```
#!/bin/bash
for i in $(seq 1 50);
```

```
    do

    /usr/bin/time -f %e --output=third.txt -a   ls
/home/mjiao/3/cern.ch/user/m/mjiao
    /usr/bin/time -f %e --output=first.txt -a   ls
/home/mjiao/1/cern.ch/user/m/mjiao
    /usr/bin/time -f %e --output=nsls.txt -a nsls /castor/cern.ch/user/m/mjiao
    /usr/bin/time -f %e --output=second.txt -a   ls
/home/mjiao/2/cern.ch/user/m/mjiao
    done
```

We did 50 times of those 4 operations to retrieve the data with performint the simple "ls" operation. After we got all the result, we calculated the mean value for the performance shown in table 4-1.

Table 4-1 Performance of "ls" operation

Tool	time (s)
ls (xrdfs)	0.9724
ls (new castorfs)	1.76244
ls (castorfs)	2.0268
nsls	1.1382

Here is the 50 times of the performance comparison:

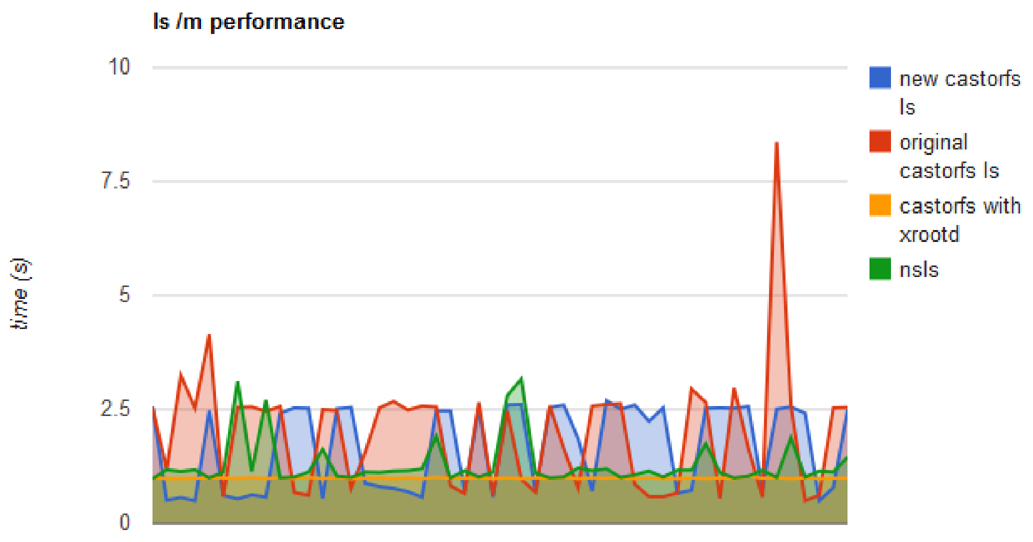


Figure 4-7 Performance diagram

Later, we did a comparison of retrieving meta-data for the same directory with the command (ls -l), and the performance is show in table 4-2.

Table 4-2 Performance of "ls -l" operation

Tool	Time(s)
Posix ls -l (xrdfs)	31.5812
Posix ls -l(new castorfs)	16.3372
Posix ls -l(castorfs)	61.608
nsls –l	1.9368

The related diagram to show the performance comparison between all those functions

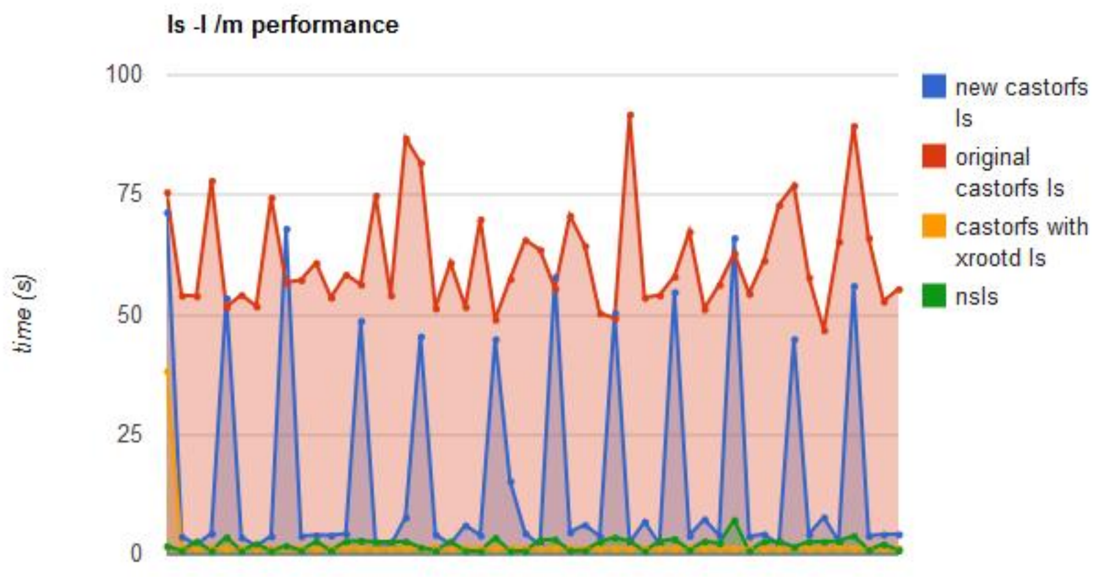


Figure 4-8 Performance comparison

We can see from figure 4-8 and figure 4-9 that, the Castorfs implemented with Xrootd protocol can operate faster than the Castorfs implemented with RFio and Ns protocols. We can also see that, with the implementation of cache, we can retrieve the meta-data much faster.

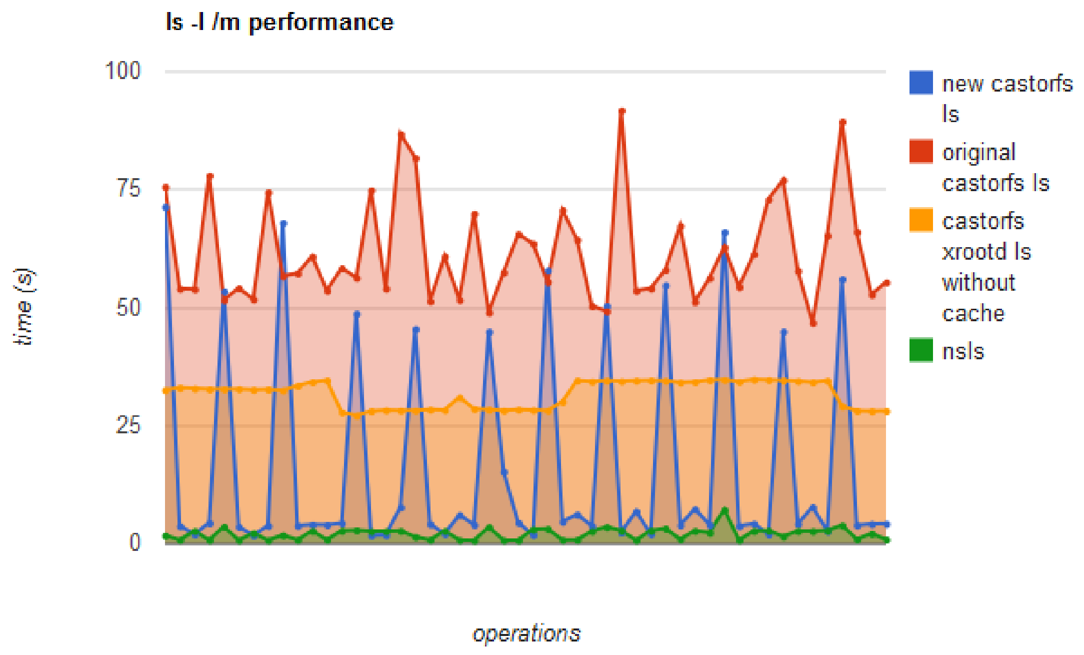


Figure 4-9 Performance comparison (castorfs without cache)

(2) The performance of reading and writing data

The reading and writing data performance is critical for the CastorFS since our users will deal a lot with the data analysis on CERN computing cluster. We need to provide a good way for them to upload and download big size files.

All the performace statistics are got from the testing on plus19 node in LHCb.

Table 4-3 Read and write peroformance

Tool	Read(Mb/s)	Write(Mb/s)
POSIX cp command on new CastorFS with Xrootd	61	35
POSIX cp command on CastorFS with NS, Rfio	31	5
rfcp command (based on CASTOR RFIO library)	100	70
xrdcp command (based on xrootd library)	112	97

We can see from table 4-3 that compared with the original CastorFS (line 2) which is implemented by using NS, Rfio libraries. The new CastorFS (line 1) improved almost twice reading speed and seven times of wrting speed on Plus node.

Table 4-4 Retrieving meta-data performance

Tool	Time(s)
POSIX ls command on new CastorFS with xrootd	33
POSIX ls command on CastorFS with NS, Rfio	53.7
nsls command (based on CASTOR Ns library)	2.4
xrd (ls) command (based on xrootd library)	25

From table 4-4, we can see that with the new implementation with xrootd to retrieve the meta-data of all the entries in the same folder the performance is better than the original one (line 2).

4.4 Brief summary

The environment of implementing the system is based on the computing environment in CERN. I used my local machine, LHCb computing cluster and lxplus to either implement the system or do the evaluation. The new CastorFS improved a lot on the meta-data retrieving, reading and writing speed compared to the original one.

Conclusion

The goal of the present work was to implement a new virtual filesystem with caching mechanism based on FUSE and the data transmitting protocol provided by Xrootd. This implied the analysis of the old file system to find the problem, making the design, doing the implementation, perfroming testing and evaluation for the system. The development was carried out within the online team of LHCb, one of the four experiments that have been approved for the future high energy collider LHC (Large Hadron Collider) at CERN. This virtual fileystem is used for giving a file system which complied with POSIX standards in order to make the manipulation of remote data easier for the user.

We successfully provided the cache mechanism for the system by adopting the LRU algorithm and implemented it with C++ container, algorithm, and iterator. Later, we implemented all the important functions which could handle most of the file system operations for the virtual file system by using Xrootd libraries. At each development phase, we did the tesing and evaluation for the system in the environment of computer in LHCb, lxplus which is maintained by CERN IT department, and LHCb inside computing cluster.

For the future work, we plan to investigate more about the security aspect provided by Xrootd. Currently, we can use Kerberos ticket-granting ticket to make the access for 25 hours. We want to make an extension of the time for this security mechanism by using certification properly. We also notice that the function of sendfile which is used for transferring data between file descriptor provided by Linux may help us to build a more real filesystem in the future.

References

[1] Fabrizio F. Large databases on the GRID. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 623, Issue 2, 11 November 2010

[2] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic et al. ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization. Computer Physics Communications, December 2009

[3] Luis B and Rassul A. Lazy update: An efficient implementation of LRU next term stacks. Information Processing Letters Volume 54, Issue 2, 28 April 1995.

[4] Antonis P, Athanasios V and Ioannis S. Approximate analysis of LRU in the case of short term correlations**.** Computer Networks, 24 April 2008

[5] Yang J, Bai Y, Qiu Y. To Select the Service in Context Aware Systems Using Concept Similarity Mechanism[C]. 2008 International Symposium on Electronic Commerce and Security. 2008: 143-147.

[6] A. Mazurov, N. Neufeld. CASTORFS – A Filesystem To Access CASTOR, Journal of Physics: Conference Series 219 (2010) 052023

[7] CASTOR service at CERN – http://cern.ch/castor

[8] Filesystem in user space – http://fuse.sourceforge.net

[9] Xrootd introduction – http://xrootd.slac.stanford.edu/docs.html

[10] Filesystem based on FUSE - http://sourceforge.net/apps/mediawiki/fuse/index.php?title=FileSystems

[11] Opti-Cache introduction - http://www.bsiopti.com/ocart.html

[12] MacFUSE introduction- http://code.google.com/p/macfuse/

[13] E. Driscoll, J. Beavers, H. Tokuda - FUSE-NT: Userspace File Systems for Windows NT

[14] Waterfall introduction - http://en.wikipedia.org/wiki/Waterfall_model

[15] Extreme programming - http://www.extremeprogramming.org/rules.html

[16] G. Donvito, V. Spinoso and G.P. Maggi. Interactive access and optimization of a CMS computing farm. Nuclear Physics B - Proceedings Supplements. June 2011, Pages 82-84

[17] René Brun. Summary of session 1: Computing technology and environment for physics research. Nuclear Instruments and Methods in Physics Research

Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 559, Issue 1, 1 April 2006

[18] A. Salnikov. Evolution of the configuration database design. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 1 April 2006

[19] A. L. S. Angelis, J. Bartke, M. Yu. Bogolyubsky, E. Ga dysz-Dziaduś et al. CASTOR: Centauro and strange object research in nucleus-nucleus collisions at the LHC. Nuclear Physics B - Proceedings Supplements Volume 97, Issues 1-3, April 2001

[20] Peter Göttlicher. Design and test beam studies for the CASTOR calorimeter of the CMS experiment. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 623, Issue 1, 1 November 2010

[21] Edward Haletky. Deploying LINUX on the Desktop. Deploying LINUX on the Desktop 2005, Pages 181-190

[22] N. Brook, H. Bulten, J. Closier, D. Galli, C. Gaspar et al. LHCb distributed computing and the GRID. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 502, Issues 2-3, 21 April 2003, Pages 334-338

[23] Jamie Shiers. The Worldwide LHC Computing Grid (worldwide LCG). Computer Physics Communications. Volume 177, Issues 1-2, July 2007

[24] Soha Maad, Brian Coghlan, Geoff Quigley. Towards a complete grid filesystem functionality. Future Generation Computer Systems. Volume 23, Issue 1, 1 January 2007

[25] Jim Mellander. Unix Filesystem Security. Information Security Technical Report. Volume 7, Issue 1, 31 March 2002

[26] Jürgen Branke, Pablo Funes. Evolutionary design of en-route caching strategies. Applied Soft Computing. Volume 7, Issue 3, June 2007

[27] Edith Cohen, Haim Kaplan and Uri Zwick. Connection caching: model and algorithms. Journal of Computer and System Sciences, August 2003

[28] Niels Sluijs, Frédéric Iterbeke. Cooperative caching versus proactive replication for location dependent request patterns. Journal of Network and Computer Applications Volume 34, Issue 2, March 2011

[29] Philip S. Yu and Edward A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. Computer Networks and ISDN Systems Volume 30, Issues 1-7, April 1998

[30] Mohamed F. Ahmed and Swapna S. Gokhale. Linux bugs: Life cycle, resolution and architectural analysis. Information and Software Technology Volume 51, Issue 11, November 2009

[31] M. Zilker and P. Heimann. High-speed data acquisition with the Solaris and Linux operating systems. Fusion Engineering and Design Volume 48, Issues 1-2, 1 August 2000

[32] André Neto, Filippo Sartori et al. Linux real-time framework for fusion devices. Fusion Engineering and Design Volume 84, Issues 7-11, June 2009

[33] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. Future Generation Computer Systems Volume 13, Issues 4-5, March 1998

[34] Ioana Banicescu, Ricolindo L. Cariño. Design and implementation of a novel dynamic load balancing library for cluster computing. Parallel Computing Volume 31, Issue 7, July 2005

[35] Les Robertson. The distributed data-base for the CERN SPS control system. Computer Physics Communications Volume 110, Issues 1-3, May 1998

哈尔滨工业大学硕士学位论文原创性声明

Statement of Copyright

哈尔滨工业大学硕士学位论文使用授权书

Letter of Authorization

Acknowledgement

At the end of this work, I want to thank my supervisor, Doc. Niko Neufeuld, for having trusted me, my ideas and my work and for his support and precious advice during the development of the project and the the writing of the thesis.

I want to thank my supervisors at HIT and ISIMA, Prof. Xiaofie Xu and Prof. Kun-mean HOU for their essential contribution towards my technical education, for having given me great support in this project, and especially, for the many hours they spent to read and correct this document.

I want to thank in particular Dr. Alexander Mazurov for his priceless help in understanding the architecture of computing cluster in LHCb and for his patience in teaching me all he knows about FUSE.

Many thanks also to my colleagues at CERN, especially Guoming, Christophe and Gregoire; theire positive attitude have encouraged me day by day.

I cannot forget to thank my parents and my sister. They always trusted me and supported my choices.

A special thank to my two great friends, Carson and Ben. Even if far away, I could feel their affection and theire support all the time.

Many thanks to all my friends, in particular to Louis and Pierre for their friendship during my stay at CERN.

Finally, I want to thank Jean-Pierre, without whom I woud have never been working at CERN and I would have never lived this great experience.

Resume

Manjun JIAO

(manjun.jiao@gmail.com) Male
Oct 12, 1986 born in Anhui, China

Education

Date	University	Degree	Major
09/2010 – 09/2011	ISIMA	Master 2	Software Engineering
09/2009 – 07/2010	Harbin Institute of Technology	Master 1	Software Engineering
09/2003 – 07/2008	Harbin Institute of Technology	Bachelor	Software Engineering

Working experiences and training

July 2010 – September 2010, Institut de recherché pour l'ingénierie de l'agriculture et de l'environnement, Clermont-Ferrand. Responsible for making a tool for configurating the Wireless sensor network and its nodes

- designing the graphic interface for defining the data collection and transmission policies and the energy policies by using Java

June 2007– August 2008, Beijing Wenlu Laser Technique Ltd. Beijing. Developer of management system for Audio and Video market:

- analyzing requirements, designing system,, accomplishing the system and testing.
- designing a module to collect data of the business, analyze the data and generate the report with graphic interface.

Competences

- English: Fluent; French: Intermidiate
- C++, C, Java, C#
- Embedded system, Linux