UNIVERSITAT POLITÈCNICA DE CATALUNYA

DEPARTMENT OF COMPUTER ARCHITECTURE

File System Metadata Virtualization

by Ernest Artiaga i Amouroux

Advisor: Toni Cortés Rosselló

A dissertation submitted in partial fulfilment of the
requirements for the degree of

Doctor per la Universitat Politècnica de Catalunya

File System Metadata Virtualization

Abstract:

The advance of computing systems has brought new ways to use and access the stored data that push the architecture of traditional file systems to its limits, making them inadequate to handle the new needs. Current challenges affect both the performance of high-end computing systems and the usability of general-purpose personal computing systems.

One of the elements that limit the usability of file systems is the exposure of their internal organization through rigid name spaces, that usually reflect how data is actually stored in the underlying media. Users must know and understand the nuts and bolts of specific storage systems in order to achieve good results, receiving the burden of taking into account the complexities and restrictions of the new environments.

This burden can be alleviated by providing dynamic, flexible name spaces adapted to specific application needs, instead of representing how data is stored in the underlying storage systems.

This thesis contributes to the goal above by proposing a mechanism to decouple the user view of the storage from its underlying structure. This mechanism consists in the virtualization of file system metadata and the introduction of a sensible layer able to take automatic decisions on where and how the files should be stored in order to take advantage of the underlying storage system features.

More specifically, the first contribution of this thesis is the design of a metadata virtualization framework which is able to produce consistent and flexible views of a file system, independently from the actual underlying data organization. The second contribution consists in using this framework to alleviate the complex file system tuning requirements in high-performance environments. Finally, The third contribution of this thesis consists in applying the metadata virtualization principles to provide an ubiquity layer for cloud storage systems that extends the cloud-awareness to the whole file system, making it independent from the directory where the cloud-related files reside.

Keywords:

File system virtualization, file system metadata, parallel file systems, cloud storage, name spaces.

Place and date:

Barcelona. October, 2013

Funding:

This work was partially supported by Spanish MECD under grants TIN2007-60625 and TIN2012-34557, and the Catalan Government under the 2009-SGR-980 grant. Part of the research leading to the presented results has also received funding from the EU IST program as part of the XtreemOS project (contract FP6-033576) and the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement RI-283493, as part of the PRACE-2IP project.

Infrastructure:

Part of the research in this thesis has been possible thanks to the high-performance computing infrastructures kindly provided by the Barcelona Supercomputing Center (BSC), the Partnership for Advanced Computing in Europe (PRACE) and the Lawrence-Livermore National Laboratory (LLNL) via the Hyperion Project; additionally, part of the works in this thesis have made use of the computing infrastructure of the Department of Computer Architecture at the Technical University of Catalonia (DAC-UPC). I sincerely acknowledge their support.

Acknowledgements

This thesis is part of a long path that started many years ago, and I have met many people that have supported me along the way and helped me in critical moments. All of them have contributed in some way to this work, I sincerely acknowledge their part in the effort.

Specially, I would like to thank Mateo Valero for giving me the opportunity of working at BSC since its beginning and dedicating these last years to research. Thanks to that, I have had access to both the people and the technological resources that have enabled me to pursue my PhD. I would not be finishing my thesis now without his support.

All journeys have a start, and I would not have reached this point without the continuous support of Pep Fuertes. He was the person that many years ago started to push me into the world of research, and trusted me even before I finished my Computer Science courses at University. I have learned many valuable things from him, and I owe many of my achievements to his advice, encouragement and friendship during all these years.

The day to day efforts leading to my PhD have been assisted by Toni Cortés who, despite knowing me, accepted to be my advisor. His vision and wisdom have indeed contributed to the happy ending of my PhD, and his friendliness and optimism have created a wonderful and interesting place to work.

I want to thank the current and former members of the Storage Systems Research Group at BSC for being a wonderful team to work in. I have received many useful inputs from them. In particular, Jonathan Martí, Jan Wiberg, Juan González de Benito and Thanos Makatos have had an important participation in the technical aspects related to this thesis. I must specially mention Alberto Miranda, whose abilities and curiosity make of him an amazing source of information and technical knowledge, apart from a nice colleague to work with.

The use of the supercomputing infrastructures required for the present work has been greatly facilitated by the Operations team at BSC. I really appreciate the help from the system administrators and the support team to prepare the rather unconventional setups required for my experiments. I specially want to thank to Javier Bartolomé, David Vicente, Sergi Moré, Jordi Valls and Albert Benet their readiness to collaborate, and the help of Ferran Sellés, Toni Espinar, Pedro Gómez and Albert Riera to solve any issues in the working environment.

I want to mention the people from the Operating Systems Group at the Department of Computer Architecture who shared my first steps (and specially Xavier Martorell, whom I honestly admire for his ability to comprehend all aspects of a system at any arbitrary level of detail). Since then, a lot of people from the Department of Computer Architecture at UPC and from the BSC have made my life easier by sharing their knowledge, but also breakfasts, coffees, meals and discussions about spherical cows. I am afraid that the list is too long to mention you all, but I really appreciate your company. I specially want to thank David Carrera, Marc González

and Anna Queralt for consistently being able to cast away any worries during lunch times.

I have a very long list of reasons to say thanks to Yolanda Becerra, and excellent colleague and good friend since school time. She has that rare quality of being reliable on good and bad times, and so many times she has helped me to stay focused during trouble. Thanks for always being there!

I also want to thank my friends for helping me to have a life outside the work. Specially Pedro, Oriol, Maria José, Rosa, Sandra and Nacho who, even when life makes it difficult to meet often, continue being great friends.

And I cannot forget my family for being by my side at all times. Specially my uncles, my cousins (Edu, Anna and Núria) and my new family in Granada (Manolo and Ana). Above all, I want to thank my mother, Pilar, for always being caring and teaching me everything I needed to prepare for life, and to my grandmother, Carmen, who would have enjoyed seeing this work finished.

Finally, I want to thank Pili, my wife, for her sustained support during all these years, for her patience and for making my life a happier place. Her energy has always inspired me to go on. Thanks for sharing your life with me!

Agraïments

Aquesta tesi és part d'un llarg camí que va començar fa molts anys, durant el qual he trobat a molta gent que m'ha anat ajudant, especialment en els moments crítics. Tots ells han aportat d'alguna forma el seu gra de sorra a aquest treball i desitjo agraïr-los sincerament la seva contribució.

En particular, m'agradaria agrair a Mateo Valero que em donés l'oportunitat de treballar al BSC des de la seva creació i poder dedicar aquests últims anys a la recerca. Gràcies a això he pogut tenir accés tant a la gent com als recursos tecnològics que m'han permès fer el doctorat. Sense el seu suport, no estaria ara escrivint aquestes línies.

Tots els camins tenen un començament, i no hauria arribat a aquest moment sense el suport continu d'en Pep Fuertes. Ell va ser la persona que, fa molts anys, em va introduir en el món de la recerca, confiant en mi fins i tot abans que acabés la carrera d'Informàtica. He tingut la sort de poder aprendre moltes coses d'ell, i moltes de les coses que he aconseguit fer les dec al seus consells, el seu suport i la seva amistat durant tots aquests anys.

Els esforços del dia a dia que han portat a la conclusió del meu doctorat han estat guiats per en Toni Cortés qui, tot i que em coneixia, va acceptar dirigir-me la tesi. La seva visió de la tecnologia i la seva experiència han contribuït de forma decisiva a que el meu doctorat tingués un bon final; la seva amistat i el seu optimisme han creat un espai interessant per treballar-hi.

Vull agrair als membres i ex-membres del Storage Systems Research Group del BSC per haver format un equip fantàstic. D'ells he rebut moltes aportacions. En particular, en Jonathan Martí, en Jan Wiberg, en Juan González de Benito i en Thanos Makatos han tingut una participació destacada en els aspectes tècnics relacionats amb aquesta tesi. Vull mencionar també de forma especial a l'Alberto Miranda, les habilitats i la curiositat del qual el converteixen en una increïble font d'informació i coneixements tècnics, a més d'un bon company de feina.

L'ús de les infraestructures de supercomputació requerides per a aquest treball ha estat enormement facilitat pel grup d'Operacions al BSC. Agraeixo sincerament els esforços dels administradors de sistemes i l'equip de suport per preparar les poc convencionals configuracions que necessitaven els meus experiments. Vull mencionar especialment al Javier Bartolomé, en David Vicente, en Sergi Moré, en Jordi Valls i l'Albert Benet la seva disposició a col·laborar, i l'ajuda d'en Ferran Sellés, en Toni Espinar, en Pedro Gómez i l'Albert Riera per sol·lucionar qualsevol problema al lloc de treball.

Vull mencionar als membres del Grup de Sistemes Operatius del Departament d'Arquitectura de Computadors que van veure les meves primeres passes (especialment a en Xavier Martorell, a qui admiro per la seva capacitat de copsar tots els aspectes d'un sistema a tots els nivells de detall). Des de llavors, molta gent del Departament d'Arquitectura de Computadors a la UPC i del BSC han fet la meva vida més senzilla i agradable compartint tant els seus coneixements com esmorzars, cafès,

dinars i discussions sobre vaques esfèriques i altres amenitats. Em temo que la llista seria massa llarga per mencionar-vos a tots, però us vull dir que aprecio sincerament la vostra companyia. Vull agrair especialment al David Carrera, en Marc González i l'Anna Queralt la seva contrastada capacitat per fer desaparèixer les preocupacions durant els dinars.

Tinc una llista molt llarga de raons per estar agraït a la Yolanda Becerra, una excell·lent companya de feina i bona amiga des de l'escola. Ella és una d'aquelles persones en qui he pogut confiar tant en els bons temps com en els no tan bons, i ha estat un punt de referència en els moments complicats. Gràcies per no fallar mai!

També vull agrair els meus amics per ajudar-me a tenir una vida fora de la feina. Vull mencionar especialment al Pedro, l'Oriol, la Maria José, la Rosa, la Sandra i en Nacho, que segueixen sent grans amics, fins i tot quan les circumstàncies fan que sigui difícil que ens veiem més sovint.

I no puc oblidar la meva família, a qui sempre he tingut al costat. Especialment els meus oncles i els meus cosins (l'Edu, l'Anna i la Núria), i la meva nova família a Granada (Manolo i Ana). I sobre tot, vull agrair a la meva mare, Pilar, per preocupar-se sempre per mi i ensenyar-me tot el que he necessitat per a poder fer la meva vida; i a la meva àvia, Carmen, a qui hauria fet il·lusió veure el final d'aquest treball.

Finalment, vull agrair a la Pili el seu suport durant tots aquests anys, la seva paciència, i el fet d'haver aconseguit fer de la meva vida un lloc més feliç. La seva il·lusió i la seva energia m'han donat forces per seguir sempre endavant. Gràcies per compartir la teva vida amb mi!

Contents

List of Figures

List of Tables

Terms and Abbreviations

API	Application Programming Interface.
BSC	Barcelona Supercomputing Center.
CD	Compact Disc.
CEA	Commissariat à l'énergie atomique et aux énergies alternatives.
CIFS	Common Internet File System protocol.
COFS	COmposite File System.
DAS	Directly-Attached Storage.
DEISA	Distributed European Infrastructure for Supercomputing Applications.
EPO	European Patent Office.
Ext3	Third Extended Filesystem.
Ext4	Fourth Extended Filesystem.
FAT	File Allocation Table.
FUSE	Filesystem in USErspace.
GPFS	General Parallel File System.
Hadoop	High-Availability Distributed Object-Oriented Platform.
hFS	Hybrid File System.
HPC	High-Performance Computing.
I/O	Input/Output.
IBM	International Business Machines.
INTI	Computing cluster at CEA (128 nodes - 1,024 cores).
IOR	Interleaved Or Random I/O benchmark.
IP	Internet Protocol.
IRP	Input/Output Request Packet.
iSCSI	Internet Small Computer System Interface.
LFS	Log-structured File System.
LLNL	Lawrence-Livermore National Laboratory.
MareNostrum	Supercomputer at BSC (2,560 nodes - 10,240 cores).
MDT	Metadata Target.
MFS	Multi-structured File System.
MFT	Master File Table.
MPI	Message Passing Interface.
MPI-IO	Message Passing Interface I/O management abstractions.

NAS	Network-Attached Storage.
NCAR	National Center for Atmospheric Research.
NFS	Network File System.
NFSv4	Network File System, Version 4.
Nord	Computing cluster at BSC (70 nodes - 140 cores).
NTFS	New Technology File System.
OBFS	OSD-Based File System.
OS	Operating System.
OSD	Object Storage Device.
OSR	Open Systems Resources, Inc.
OTP	Open Telecom Platform Framework.
PLFS	Parallel Log-structured File System.
pNFS	Parallel Network File System.
POSIX	Portable Operating System Interface.
PRACE-2IP	Partnership for Advanced Computing in Europe, Second Implementation Phase.
PVFS	Parallel Virtual File System.
PVFSv2	Parallel Virtual File System, Version 2.
QDR	Quad Data Rate.
RAIF	Redundant Array of Independent Filesystems.
RAM	Random-Access Memory.
SAN	Storage Area Network.
SDSC	San Diego Supercomputer Center.
SMB	Server Message Block protocol.
SRB	Storage Resource Broker.
SSL	Secure Socket Layer.
SSU	Lustre Scalable Storage Unit.
TSS	Tactical Storage System.
UCAR	University Corporation for Atmospheric Research.
ULS	User-Level Service.
USB	Universal Serial Bus.
VFS	Virtual Filesystem Switch.

Units

Bandwidth	
Gbps	gigabits per second.
MB/s	megabytes per second.
Frequency	
GHz	gigaherz.
Storage	
b	bit.
GiB	gibibyte.
KiB	kibibyte.
MiB	mebibyte.
Time	
ms	millisecond.

First things first, but not necessarily in that order.
(The Doctor)

CHAPTER 1

Introduction

Contents

1.1 Motivation

The advance of computing systems has brought new ways to use and access the stored data that push the architecture of traditional file systems to its limits, making them inadequate to handle the new needs. Current challenges affect both the performance of high-end computing systems and its usability from the applications perspective.

On one side, high-performance computing equipment is rapidly developing into large-scale aggregations of computing elements in the form of clusters, grids or clouds. In the last few years, the size of such distributed systems has increased from tens of nodes to tens of thousands, and the number is still rising.

On the other side, there is a widening range of scientific and commercial applications which seek to exploit these new computing facilities. The requirements of such applications are also heterogeneous, usually leading to dissimilar patterns of use of the underlying file system.

These new computing environments require parallel and distributed storage systems able to provide a sustained flux of data to the distributed computing elements. Nevertheless, many of the current distributed file systems are heavily oriented to provide performance for specific software environments with distinctly predominant traits such as favouring local independent accesses, having low modification ratios, performing data streaming or executing high-performance MPI-IO codes, just to mention some possibilities. The downside of file system specialization is that it jeopardizes the ability to adapt to varying use patterns from different applications.

Data centres have tried to compensate this situation by providing several file systems to fulfil distinct requirements such as storing operating system files, users' data, long-term storage, etc. Typically, the different file systems are mounted on different branches of a directory tree, and the preferred use of each branch is publicised to users.

A similar approach is being used in personal computing devices. Typically, in a personal computer there is a visible and clear distinction between the portion of the file system name space dedicated to local storage (i.e. data residing on the personal computer itself), the part corresponding to remote file systems (such as corporate *SMB*/*CIFS shares* or other remote data servers) and, recently, the areas linked to cloud services as, for example, directories to keep data synchronized across devices, to be shared with other users, or to be remotely backed-up.

In such environments, the burden of understanding and taking into account the complexities and restrictions of each file system is passed to the user of the system (either the final user or the application developer), who must keep in mind the nuts and bolts of specific storage systems in order to achieve good results and adapt his way of working to their restrictions. Summarizing, in this scenario a user must:

- Explicitly choose which file system to use by placing the files in the corresponding branch of the directory tree (which may not correspond with a logical and/or sensible organization of the files, from user/application perspective).

- Make sure that each branch is used according to certain rules and restrictions, which may differ among file systems (e.g. a high creation/deletion ratio may be fine in a local file system, but it may collapse the system when done in a branch corresponding to a remote file system).

- Know how the file systems are organized and always remember which services/features are available or not in each branch; this feature-consciousness may be especially important when dealing with critical services such as encryption, high-availability or backup support.

In practice, this approach compromises the usability of the file systems and the possibility of exploiting all their potential benefits. Moreover, it also creates a dependency on specific configuration details, as users and applications learn to rely on a particular setup. The consequence is that ulterior changes (in case of moving to a different platform or simply re-provisioning) may incur in high re-adaptation costs.

We consider that this burden can be alleviated by determining applicable features on a per-file basis, and not associating them to the location in a static, rigid name space. Moreover, usability would be further increased by providing multiple dynamic name spaces that could be adapted to specific application needs.

This thesis contributes to the goal above by proposing a mechanism to decouple the user view of the storage from its underlying structure. This mechanism consists in the virtualization of file system metadata (including both the name space and the objects attributes) and the interposition of a sensible layer to take decisions on where and how the files should be stored in order to benefit from the underlying file systems features, without incurring on usability or performance penalties due to inadequate usage.

1.2 Contributions

1.2.1 Metadata virtualization framework

File systems are complex entities containing many components that interact to provide a consistent and well-defined way to access stored data.

In a conventional file system, the components dealing with metadata are tightly coupled with the underlying storage structures, forcing a rigid view of the data organization. In order to provide a flexible and adaptable presentation of the name space and the objects attributes, it is necessary to decouple the metadata management from the rest of the file system components, but without altering the semantics nor limiting the functionalities.

The first contribution of this thesis is the design of a metadata virtualization framework which is able to produce consistent and flexible views of a file system which are independent of the actual underlying data organization.

The novelty of the metadata virtualization framework consists in its ability to provide multiple, simultaneous, arbitrarily organized views of the data, yet keeping the consistency of the underlying file system. Additionally, the framework introduces a series of novel techniques to represent and handle metadata which overcome some of the limitations of traditional file systems. In particular, file system objects are represented as active intercommunicating processes instead of passive data structures, improving the parallelization opportunities; also, a new partitioned directory schema allows stateless yet consistent listings of directories in large-scale systems.

The work performed in this area resulted in the filing of the International Patent Application PCT/EP2011/053410 [Artiaga 2011], which is currently following the examination process in the National Phase by the European Patent Office.

1.2.2 Improved metadata service for large-scale file systems

Modern parallel file systems need to provide sustained performance able to catch up with the efficiency achieved by current large-scale computing systems. In order to do so, file system internals are heavily optimized, and their operational parameters are carefully tuned to particular installations. Nevertheless, optimization and tuning often involve trade-offs, and outstanding performance for certain cases may become poor results when the assumed conditions are not matched by a particular workload.

The situation is aggravated by the increasing number of parameters that affect the file system performance and must be tuned, requiring an in-depth knowledge of the system internals. This fine-tuning of system parameters implies that the file system must be used in a precise manner, and this responsibility is pushed to the user, who must either adapt the applications to the particular characteristics of the system (with the consequent cost in terms of application porting or even software redesign) or simply 'misuse' the environment and be ready to accept performance losses (which also

have a cost in terms of wasted computing time). In certain production environments, a user may not even have the information or capabilities required to perform such adaptations.

The second contribution of this thesis consists in alleviating the over-tuning requirements in high-performance computing systems by means of our metadata virtualization framework.

The metadata virtualization framework is able to produce a file system name space which isolates the user from the actual file system layout, so that the user does not need to be aware of the low-level details and restrictions of the underlying system. Then, sensible modules plugged into the metadata virtualization framework transparently translate applications requests into access patterns adapted to the underlying file system features. A change in the system configuration may require tuning the translation module in the framework, but user applications remain unaffected.

A framework prototype has been tested in high-performance environments using parallel file systems such as *IBM GPFS* and *Lustre*. The proposed technique has been successfully used to transparently eliminate harmful use patterns in the MareNostrum supercomputer, improving the metadata performance of the file system.

The work performed in this area resulted in the publication of the paper "Using file system virtualization to avoid metadata bottlenecks" [Artiaga 2010], and the PRACE-2IP white paper "Lessons learned about metadata performance in the PRACE file system prototype" [Artiaga 2013a].

1.2.3 Transparent ubiquity layer for cloud storage

Currently, cloud-based storage services are widely available and there are multiple commercial products offering features such as data synchronization among devices, data sharing among users or backup management based on the cloud.

These products successfully face challenges such as keeping the integrity and consistency of data and optimizing the information transfer between the cloud and the local systems. Nevertheless, they usually show a lack of flexibility: the selection of files to consign to the cloud is usually rigid and coarse-grained, based on placing the files in specific directories of the file system. The reason is that most file systems offer change detection interfaces that work on a directory level and, therefore, it is not feasible to track single files independently from their location.

The third contribution of this thesis consists in providing an ubiquity layer that extends the cloud-awareness to the whole file system, making it independent of the directory where the cloud-related files reside.

The method consists in using the metadata virtualization framework to provide alternate name spaces to the user and to the local cloud service components. Using a virtual view of the file system, the user applications can organize the files as they please; at the same time, the framework is able to export a special name space to the cloud components, where the cloud-related files appear grouped in directories as the

cloud expects them to be. In this work, we have focused our efforts in validating the feasibility of the method by developing prototypes that can be integrated in *Linux*-based and *Microsoft Windows*-based operating systems.

The work performed in this area has resulted in the publication of the paper "Better Cloud Storage Usability Through Name Space Virtualization" [Artiaga 2013b].

1.3 Organization of the document

This thesis is organized as follows: Chapter 2 introduces basic concepts about file systems and associated technologies that will be used as a reference in the rest of the document; Chapter 3 explains the principles behind the metadata virtualization model, as well as discusses the techniques used to implement a virtualization framework and overcome the potential issues; Chapter 4 presents and evaluates a technique to improve metadata performance in high-performance environments based on metadata virtualization; Chapter 5 presents method to improve the usability and flexibility of cloud-based storage through the provision of multiple virtual name spaces; Chapter 6 discusses the related work and, finally, Chapter 7 presents the conclusions of this thesis.

CHAPTER 2

Technological Background

Contents

2.1 Introduction

Traditionally, rotating disks have constituted the hardware base for computing storage systems. The principles of this technology have not changed since its inception; despite the advances that made them faster and smaller, they still work very much in the same way as their predecessors: a read/write head is mechanically moved to the selected track and the rotation of the media makes the desired data pass below it, so that the information can be accessed.

One of the driving goals in the evolution of the storage systems based on rotating disks was improving their performance by reducing the average time required to place the read/write head of the disk on the right track (seek time) and the average time to wait for the desired data to pass below the head (rotational delay).

This goal can be achieved by hardware means (e.g. increasing the speed of the disks mechanical parts), but also through software means: for example, by concentrating and accessing as many data as possible in adjacent positions, so that physical movements and idle wait times are avoided. In order to take advantage of this possibility, data to be stored in disks is usually grouped in large blocks whose contents are expected to be used at the same time.

File systems lay on top of the physical media and offer a high level view of the storage to the user, allowing the organization of data into files which are usually accessed through a hierarchical name space composed of directories and subdirectories. Maintaining these higher level abstractions requires additional information, apart from the pure data that the user wants to keep; this additional information is usually referred to as 'metadata'.

The principle of grouping pieces of information that are going to be used together has permeated into the design of file systems. Indeed, most of the file systems try to structure the users' data and the corresponding metadata in such a way that they could be stored in close locations if they are assumed to be used at the same time.

Unfortunately, the tight coupling between data and metadata implies rigidity in the file system organization, hindering the adaptability of file systems to today's changing environments, and also their usability for widely different applications.

This thesis defends the idea that decoupling the data and its physical location from the metadata used to describe and organize user files increases file system flexibility and its adaptability to varying purposes.

In order to decouple metadata in a sensible way, it is important to understand the reasons behind its current organization. This chapter describes the structure of classical file systems, how they have evolved trying to meet current needs, and why the inherited rigid tights between data and metadata strain the performance and usability of modern file systems.

2.2 Classical file system structure

Storage devices typically provide crude means to store unstructured raw data persistently. For example, a disk can be seen as a collection of fixed-size blocks identified by its physical position in the disk. Arguably, this is not the most comfortable support for building robust and portable applications.

In order to overcome the difficulties of working with raw storage devices, file systems provide higher-level abstractions to work with data: the files. In a modern file system, a file may have variable size, a set of attributes describing properties of the data contained in it, and a way to reference it independently from its physical location in a particular device.

Figure 2.1 represents the basic elements of a file system. The file system resides between the application and the raw storage device. It receives high-level, file-oriented requests from the application via the file system interface and, when necessary, interacts with the low-level storage device to store and retrieve persistent data.

The file system uses metadata to fulfil the necessary functionalities. We may distinguish three different types of metadata information: the name space metadata keeps information about the naming of the files and its organization, the file-specific metadata describes properties of individual files (such as who has access to it), and the global metadata handles information about the storage system as a whole (e.g. how much free space remains available).

The name space defines how individual files can be referenced from an application, and how they are organized. For example, a typical file system allows human-readable file names and usually permits the organization of such files into a hierarchical directory tree. In this case, the name space metadata would mainly comprise the means to map the entries (the visible file system object names) into the internal file system objects, the information about the hierarchical relationship between the directories, and the lists of entries belonging to each directory.

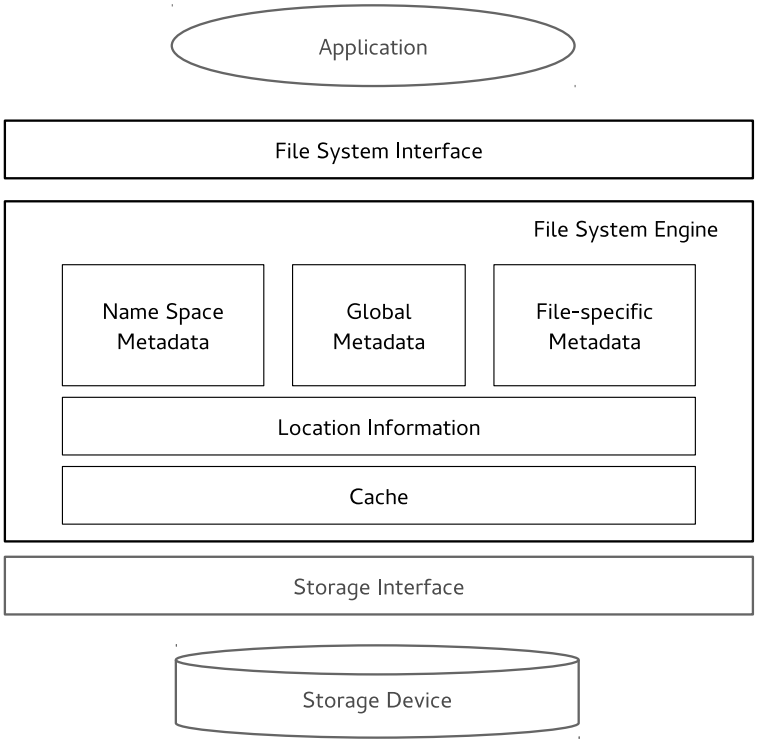

Figure 2.1: Basic file system structure.

The file-specific metadata comprises attributes that represent properties of a file system object. These properties include small information items such as access control information (who has rights to access the object and how), statistical and auditing information (e.g. when the object was last modified) and functional information (such as the size of the valid data contained in a file). Modern file systems also provide mechanisms to handle so-called extended attributes; these are user-defined pieces of arbitrary information that can be attached to file system objects and can be subject to application-specific semantics (for instance, extended attributes can be used to carry tags for an external classification system).

The global metadata represents functional data about the storage system as a whole. This may include the list of attached storage devices, their particular addressing units, the available space in them, etc.

Eventually, the actual file data has to be stored and retrieved from the storage devices; and the same applies to the metadata, which also has to be stored in persistent media in order to sustain the view of the file system organization. To this end, the location information indicates where a particular piece of data or metadata is to be found inside the storage devices (for example, in a block-based device, such information could comprise the identifier of the block, the offset of the interesting data from the beginning of the block, and the size of the piece of data).

Historically, cache management has also been a centric aspect of file systems. Persistent mass storage devices have been traditionally slower than the rest of components of computer systems, so it is customary for file systems to cache the most used data into main memory in order to avoid as many storage device accesses as possible. The cache is used for both data and metadata, and one of the challenges for file systems is to make sure that data in the storage device is kept consistent (for instance, when data has been modified in the cache but not yet completely transferred to the persistent storage).

The level of intricacy in the relationship between the different types of metadata and the data itself depends on the implementation of the file system. However, this relationship is usually quite tight.

The reason is that most classical computer systems tend to group the relatively small pieces of metadata together, in such a way that they can be stored and retrieved from slow storage devices in a single shot, and then be kept in memory. Consequently, the data structures intermix file attributes with location information and, possibly, with the file contents themselves.

This principle also applies to information related to groups of files that are going to be used together. To this end, some file systems tend to interpret directories as a hint to indicate that there is a relation between their files, and also try to put information about files in the same directory in close locations, so it can be retrieved at once.

Examples of this trend are found in the most popular file systems. For instance, entries in the old FAT-based file systems consisted of the name of the file or directory, its attributes, and its starting physical location. In the modern *Microsoft NTFS* file system, files and directories are represented by entries in a Master File Table (MFT) that also contain the name of the object, some attributes and, even, the contents themselves, if they are small. Directories are represented as special files consisting in a collection of entries (which, in the case of *Microsoft NTFS*, may replicate some of the attributes present in the MFT).

Unix-based file systems usually exercise a better separation between the name space and the file attributes. The directories of the name space are represented also as files with collections of entries, but these entries do not contain file attributes; they only contain a reference to another structure representing the file: the *i-node* structure. This *i-node* still contains all the attributes and also the location information to access the file contents in the storage device, and modern implementations try to store *i-nodes* near the actual data of the file they represent.

This amalgamation of metadata and data has been useful to speed-up the storage systems by enabling an efficient use of cache mechanisms, specially in environments where the storage devices where directly attached to the computer system using their data.

Nevertheless, modern computer systems have brought new ways to use and access stored data. For example, parallel and distributed systems are a commodity today, and these systems try to perform simultaneous accesses (and updates) to the same

pieces of data and metadata residing in remote locations. As a consequence, complex locking mechanisms must be put in place to prevent these simultaneous accesses from leaving the information in an inconsistent state; and the more tightly coupled the data and metadata are, the higher is the probability of a conflicting access.

Also, modern users demand flexible organizations of files adapted to specific application needs and goals. Therefore, a permanent, and rigid, directory tree hierarchy closely related to the location of data on a storage device does not fit the user ways any more. On the contrary, there is a demand for flexible mechanisms allowing the user to choose the most appropriate file organization for each different task.

Naturally, file systems have tried to adapt to the new environments and demands. Next sections summarize some of the approaches taken to address the new challenges.

2.3 Distributed storage

The ability to access data from multiple locations is an emerging need for current distributed systems (clusters and grids). Ideally, a storage architecture should provide strong security, data sharing across platforms, high performance and scalability in terms of clients and devices. Different architectures try to address these needs, and all of them involve certain trade-offs in terms of the provided features [Mesnier 2003].

Storage Area Networks (SAN) were introduced to deal with the connectivity limitations of traditional Directly-Attached Storage (block-based storage devices directly connected to the I/O bus of a host machine). A SAN provides a fast and scalable interconnect network (such as iSCSI or FibreChannel) that allows the connection of a large number of storage devices and host machines, so that hosts see the block devices as if they were locally attached; for example, in the case of iSCSI, each set of blocks is exposed as an *iSCSI Target* and can be mounted by an *iSCSI Initiator*. This setup has been shown competitive in terms of performance [Aiken 2003].

Some projects like *Peabody* [Morrey III 2003] use the *iSCSI Target* mechanism to virtualize the disk and provide extended features behind the scenes, such as automatic check-pointing, coalescing blocks with the same content, or eliminating silent writes (i.e. avoiding writing sectors to disk for which the new content is the same as the old one). As the *iSCSI Target* exports a low-level block interface, any file system running on top of any *iSCSI Initiator* benefits from the added features in a transparent way.

The downside of SAN architectures is that they, per se, do not provide support for letting multiple hosts access the data: they simply provide means to attach multiple devices to a host. If access is to be shared among several hosts, the information to map blocks to high-level abstractions (files and directories) and the corresponding access coordination must be handled by an external entity (e.g. a software component able to share this information among the participating hosts). The complexity of this task is one of the major drawbacks for considering SAN architectures as a stand-alone solution for scalable distributed systems.

A common approach to improve the scalability of storage systems in terms of accessing the data from multiple hosts is the so-called Network-Attached Storage architecture (NAS). This architecture is based on a reduced set of hosts acting as file servers (usually dedicated to this task), and which are directly attached to the storage devices via SAN. The number of servers is reduced so that the complexity of coordinating their access to the shared storage and handling the necessary metadata is kept bounded. On the other hand, these file servers are also configured to export higher-level storage abstractions to multiple clients. Using high-level interfaces instead of basic block addressing facilitates the implementation of shared access and the storage can be made available to many more client hosts (which access the data indirectly through the file servers). *NFS* (in its different versions [Pawlowski 1994] [Pawlowski 2000]) is a clear example of a NAS architecture.

The arrival of NAS environments moved the pressure of scalability from the storage devices to the file servers, which funnel all the storage traffic and, eventually, may constitute a bottleneck preventing the full use of the storage device bandwidth. To mitigate this effect, SAN file systems appeared as a new step in storage design.

In a SAN file system, both file servers and clients are connected to a SAN storage network, but metadata remains under the control of the servers. Commonly, clients ask the servers for metadata, and then recover the data directly from devices; this way, consistency is guaranteed by a small number of servers, without hindering the I/O bandwidth.

Nevertheless, as the number of participating hosts and storage devices increased, security and integrity in SAN file systems became a concern. The reason was that classical storage devices export data as fixed-size blocks, without any knowledge about their contents. The organization of these blocks into high-level constructs (such as files) is the task of an external entity (the file system), but it has little or no support at all from the block devices. As all hosts have access to storage devices, there is a risk that a malicious or malfunctioning client causes havoc on the storage infrastructure.

Some systems, such as *Farsite* [Adya 2002], address this issue by making heavy use of encryption and digital signatures; another approach consists in replacing the back-end block-based devices with object storage devices (OSD). An OSD exports variable-length containers of data and offers the possibility to enrich them with semantic and security information in the form of attributes [Factor 2005] (including ownership, capabilities, or striping and replication policies). File systems such as *Ceph* [Weil 2006], *zFS* [Rodeh 2003] and *Lustre* [Braam 2007] use OSDs as back-end storage devices.

The trend resulting from the need to add higher-level functionalities to the devices is that clients do not access low-level devices directly any more, but file system data servers that provide the extra needed support such as, for example, security.

Taking all the above into account, Figure 2.2 shows the architecture of a modern distributed file system. This is the architecture we will use as a reference, though it is possible that some file system implementations have minor variations (that will be indicated when relevant).

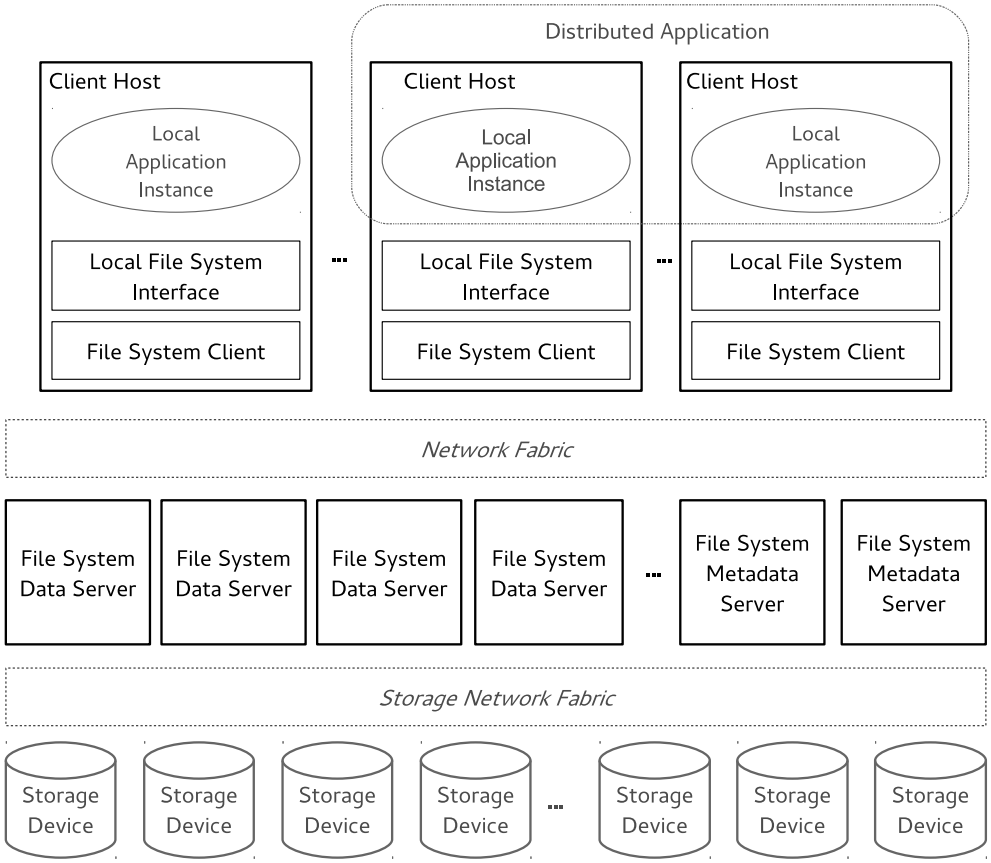

Figure 2.2: Distributed file system architecture.

In Figure 2.2, the raw storage devices lie at the bottom and are never accessed directly from the client hosts. Instead, they are used as storage media by the file system servers, possibly through a dedicated storage network (e.g. a SAN).

The file system servers are divided into data servers and metadata servers. The data servers are responsible to provide data to clients and, at the same time, to enforce security and to provide any required high-level functionalities (e.g. an OSD interface). Some distributed systems rely on classic local file systems to handle the data inside the data servers. For example, the *TSS* [Thain 2005] builds high-level storage abstractions on top of a common *Unix* file system; distributed file systems such as *Google File System* [Ghemawat 2003] and *PVFSv2* [PVF 2003] also use conventional file systems to store data chunks.

On the other side, the metadata servers handle the metadata and provide the client hosts with a consistent name space and semantics. Regarding metadata organization, current file systems are mainly divided into two groups: those which opt for distributing metadata among multiple metadata servers to achieve scalability, and those which use centralized management in a single metadata server (replicated only as failover backup).

The distributed approach require complex techniques to keep the consistency, such as effective distributed locking algorithms, leases and control delegation to speed-up independent operations on different parts of the file system; nevertheless, it may incur in performance penalties when a global synchronization is actually required or, simply, when connectivity failures isolate some of the participants in the distributed algorithms. On the contrary, centralized management seeks to compensate the lack of parallelism with simpler (and faster) procedures for both consistency management and failure recovery.

We may find systems like *IBM GPFS* [Schmuck 2002] and *xFS* [Wang 1993] using the distributed metadata approach but, up to now, most of the relevant file systems were using centralized mechanisms as, for example, *Lustre* [Braam 2007], *Hadoop* [Borthakur 2007], *Google File System* [Ghemawat 2003] and *PVFSv2* [PVF 2003]. However, efforts are being done to incorporate distributed metadata management mechanisms into systems initially designed as centralized; this is the case of *Farsite* [Douceur 2006] or recent *Lustre* versions (version 2.4 and beyond).

The remaining participants in Figure 2.2 are the client hosts, which access the file system servers (both data and metadata) through a network. A common approach for general-purpose systems is to hide the file system-specific client behind a standard local file system interface; this way, local processes can make use of the distributed file systems transparently as if it were a conventional local file system.

In this kind of environments, distributed applications usually consist of aggregations of local instances represented by processes that run locally in a single host. The local processes in different hosts are able to see the same name space, thanks to the distributed file system running below.

2.4 Metadata issues

The development of improved file systems has been traditionally focused on data transfers. Nevertheless, the increase in size and capacity of the distributed file systems are currently putting the stress in the metadata handling.

It is interesting to note that most of the current file systems use the *i-node* (or an equivalent set of attributes, such as the MFT file entries in *Microsoft NTFS*) as the smallest granularity unit for metadata. Handling all file attributes together made sense as a means of gathering structures large enough to be efficient when transferring them to and from block-based devices.

Nevertheless, the set of attributes of a file system object contain information with different volatility ratios that may cause unnecessary cache invalidations and synchronization traffic when they are partially modified. (Just to mention an example, an *i-node* contains the object type, which is immutable, the ownership and access permission information, which change rarely but are checked often, and access times, which may change often, but are rarely needed.)

A similar situation occurs for directories. A single directory may contain many

unrelated files with subsets of them being used by different applications. Therefore, treating a directory as a unit may cause *false-sharing* situations leading to potential access conflicts.

2.5 Layered architectures

Some modern file system architectures go one step beyond the models described in Section 2.2 and Section 2.3 and allow structuring file system services as a composition of layers. These layers are typically interposed in the interfaces between components and allow providing additional functionalities (such as encryption or replication) or even completely replacing the original file system (for example, by transparently redirecting requests to a different file system).

The *mount points* in classical *Unix*-based file systems can be considered a primitive way of layering file systems. The *Unix* view of the storage system usually consists in a single-rooted tree of directories, representing a hierarchy of files. Different file systems can be attached to this unique tree by using specific directories as *mount points*. When a file system is mounted on a directory, the previous contents are not visible any more and, when an application crosses that directory, it transparently accesses the root directory of the attached file system.

The *Plan 9* operating system [Pike 1990] made extensive use of this feature to offer personalized directory trees: different file services were mounted in the same directory by different processes, so that each process viewed different contents. This feature was later extended to allow the mix of several directory contents, effectively creating a *union file system* [Pike 1992].

Since then, union mounts became available in different *Unix* flavours [Pendry 1995] [Wright 2006], and different directories could be stacked in layers. The main issue related to these systems was dealing with conflicts between layers while maintaining the standard semantics; specifically: how to choose the right file version when it existed in several layers, where to write the changes and how to deal with deletions.

A common approach consisted in making all layers read-only but the top-most. When accessing a file, the path was checked for all layers from top to bottom, and the first instance found was the one returned. The modification of a file involved copying it to a read-write layer (usually the top one) and performing the modifications in the copy. Finally, deleting a file was usually implemented by creating *white-out* entries that hid underlying entries with the same name.

Still, some issues remain: atomicity of certain operations is difficult to get when combining several file systems, and performance tends to suffer when the number of layers is increased (in the worst case, each operation has to be done on all layers, as in the case of searching for a non-existing file) [Wright 2006].

Currently, most operating systems offer some level of support for layered file systems as a means to extend the functionalities of a file system or to facilitate the

integration of different file systems [Zadok 2006]. Common uses of this technology include adding encryption support for stored data, transparently accessing remote storage systems and providing file system-like interfaces to non-storage systems.

The work in this thesis borrows some of the concepts from layered file systems but, instead of composing fully-fledged file systems, we focus on providing and combining different metadata views. The result are virtual name spaces and file attribute sets that can be adapted to specific application needs without having to change how data is handled and stored in the low-level devices.

CHAPTER 3

The Approach: Metadata Virtualization

Contents

3.1 Introduction

The advance of computing systems has brought new ways to use and access the stored data that push the architecture of traditional file systems to the limits, making them inadequate to handle the new needs.

One of the causes of the inadequacy is the interlinked nature of the relationship between metadata and the actual physical data location in the storage media. The consequence of this relation is a rigid file system organization where, by default, a specific file or directory is available only in a unique precise location in a directory tree. Moreover, the view of the file system organization is fixed for the whole computer: although it is possible to protect or restrict access to certain areas (e.g. via access

permissions), it is impractical to offer radically distinct views to different users or applications.

Modern file systems should allow users to organize their data in multiple co-existing views adapted to different applications. Moreover, they should be able to access and share their files from anywhere in a transparent way, independently of the storage device where they actually reside, as if they were always using a conventional disk permanently attached to their computer.

Summarizing, a modern file system should fulfil the following requirements:

Flexibility: Files and folders must be accessible as pleases the user at each moment, allowing multiple simultaneous organizations or views of the same file systems adapted to each particular need. A view may consist in selecting certain files with specific characteristics and making them available in different directories, while other views may consist, for example, in completely hiding portions of the file system when the user is in a specific location or assumes a certain role (for example, removing access to work-related data while browsing the web at home).

Uniform file-based features: File-related features should be available in the whole file system and be selectable on a per-file basis; specifically, their functionality should not depend on the directory where a particular file is located. This principle also includes the ability to decide which files are to be shared, with whom, when, and what can be done by the person to whom the access is granted, without these decisions affecting the way in which files are organized.

Transparency: Users must not notice any difference between the way of working with the new service and the traditional file systems. Conventional applications must work on the new platform without changes.

Additionally, modern storage systems should be able to provide seamless support for remote facilities (e.g. distributed file systems or cloud services). The integration with remote services also means that the system must be able to handle a much larger storage capacity which, besides, can be dynamically increased.

One of the contributions of this thesis is the provision of a metadata virtualization framework that facilitates the achievement of a file system fulfilling the above-mentioned requirements. In this work, we have focused on two key elements: first, the flexibility in the organization of files, allowing different simultaneous views of those files adapted to specific needs; second, the seamless integration of the new functionalities into the conventional systems, hiding the complexity of new storage services and features and allowing them to be transparently used by conventional applications.

3.1.1 The case for multiple views

We present two use cases that have guided the design of our metadata virtualization framework and that illustrate the potential benefits of allowing multiple co-existing file organizations.

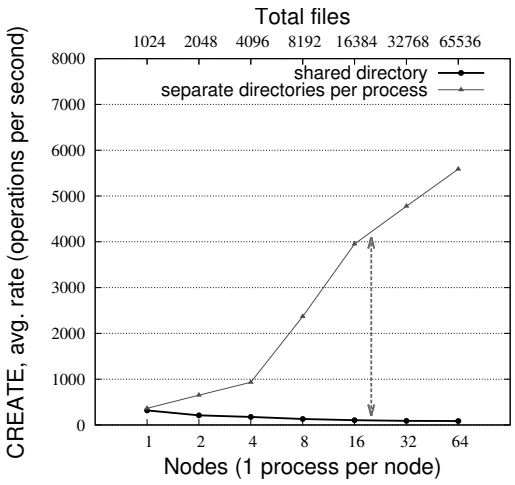

Figure 3.1: Example of variation in the operation rates for parallel file creation at MareNostrum, depending on files being organized in a shared directory or in separated directories dedicated to each participating process.

Avoidance of performance bottlenecks. Modern file systems, specially in high-performance parallel environments, try to keep pace with the growing performance demands by identifying common use patterns and optimizing them. A common hint used by these systems is the fact that unrelated data tend to be in separated directories while related files are usually in close locations, and applications are expected to abide by this principle. Unfortunately, legacy applications may not follow this kind of rules, or a user or developer may not have the necessary information or skills to adapt an application to specific file system expectations regarding the organization of the files.

A system supporting multiple views of the file system can solve this situation by providing the application with a file system view without restrictions, and then convert that view to an organization adapted to the file system optimizations. For example, a parallel application creating many independent files could be transparently redirected to multiple directories, to avoid the overhead associated to parallel creations in a single directory.

An example of how file system organization can affect performance was found at the MareNostrum supercomputer at BSC, where it was observed that deciding to organize files in a single shared directory vs. multiple dedicated directories had a substantial impact on performance (see, for example, Figure 3.1). This particular case will be studied in detail in Chapter 4.

Globally available features. Different storage systems provide different features and, in order to use them, users must place their files in the appropriate area of the name space. For example, files with sensitive information may need to placed into a specific encrypted directory, and files to be shared via a cloud service or a shared file system may also need to be stored in particular directories. Clearly, this generates a conflict with an ideal function-conscious organization where files would be organized in the most useful way according to users needs, instead of according to the specific features the files require.

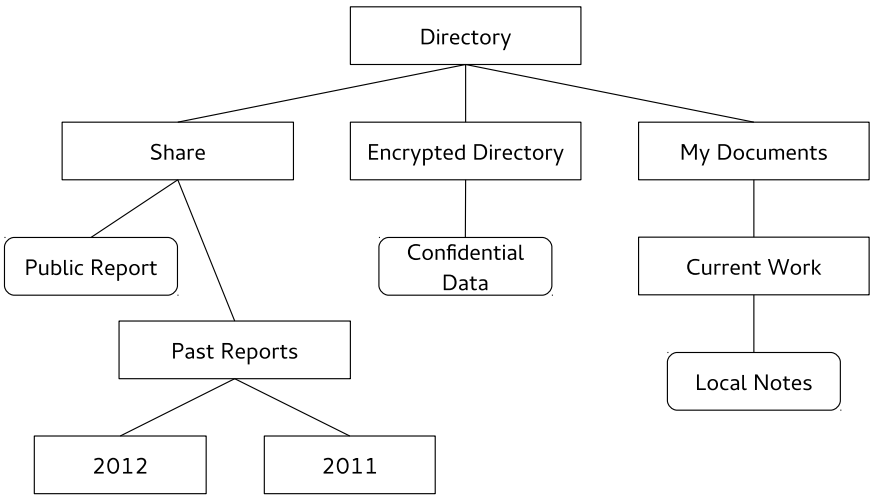

Figure 3.2: Example of directory-based feature variations: the location of a file in the hierarchical tree determines if a particular file is encrypted, if its made available through a remote file server (share) or kept local in the user's workstation documents folder.

Figure 3.2 represents a possible organization in an employee's workstation, where a report has to be written and made available through a corporate remote file service (the "`Share`" directory) with a well-defined and rigid structure, some of the necessary data is confidential and have to be encrypted, and some temporary notes are kept in the user's documents folder. These requirements force the user to distribute the work files manually across the directory hierarchy, so that the proper features are applied to each file. However, it would be much more comfortable if the employee could have all the files in his "`Current Work`" folder and simply indicate the desired features on a per-file basis.

A system supporting multiple views of the file system can offer to an application a name space where files can be organized arbitrarily, independently from the features they require from the underlying system; at the same time, a different view can convert this arbitrary organization into a distribution of files where each file is in the necessary location to benefit from the required functionalities. The consequence is that, for the application, all features appear to be available globally, instead of being limited to certain directories.

3.1.2 The need of seamless system integration

Storage for modern computing does not rely any more on a single file system on a locally attached disk. Even a simple home computer may have several traditional disks, as well as USB connected drives, or smaller devices such as pen-drives.

Additionally, distributed file systems can also be used (such as *NFS* in *Unix* environments, or *SMB*/*CIFS* in the *Microsoft Windows* world, to mention two widely used

examples). In this case, data does not reside locally on the computer where they are being used, but in disks attached to remote servers. This has several advantages: first, the aggregated storage capacity of the servers can be considerably larger than the local disks (and can be extended in a transparent way in case of need); on the other hand, providing service to a potentially larger amount of users allows space to be used with greater efficacy, and cost-effective solutions for backup and replication can be implemented.

Unfortunately, the architecture and internal behaviour of file systems, as well as the differences in their respective metadata, prevent their integration at a low level. As a consequence, users are forced to deal with separated file and directory hierarchies for each file system and handle them in different ways.

This is especially true for distributed file systems, which present additional difficulties. These systems require an active network connection; otherwise, data is not accessible. Even when network is available, bandwidth limitations usually produce noticeable drops in data transfer performance compared to locally attached disks. Such limitations and the lack of integration between the local and remote systems force users to move data from the remote file system to the local area and the other way around constantly, replicate files, apply explicit synchronization mechanisms and usually maintain similar directory trees in the different systems.

Ideally, users should be able to handle a unified name space seamlessly integrating different file systems (either local or remote). It ought to be possible to keep in the same directory some remote files while others (for example, temporary files or processing by-products) are kept local. Again, the tight connection between the metadata (and, in particular, the name spaces) and the actual data location in file systems makes this very difficult to achieve with conventional approaches. Some of the techniques presented in this thesis aim to facilitate such seamlessly integrated view across file systems.

3.1.3 Previous approaches

We observe that there is a need for multiple, dynamic, integrated views of the file system that can enable the users to organize and access their data in the most convenient way at each moment, without arbitrary restrictions.

The proof that this is a real need lays in the fact that most file systems have come up with different workarounds to provide some appearance of flexibility. These mechanisms are usually static and cumbersome, as the internal file system architecture is not changed.

Several file systems offer the possibility to have ‘links’ or ‘shortcuts’. These essentially consist in named references to the original files or directories which can be placed in different locations of the name space. Nevertheless, they have to be maintained manually for each file (a cumbersome procedure if the links are to be created or updated often — even if some systems allow some degree of automation, such as the *folder actions* in *Apple Mac OS X*). Moreover, link behaviour differs from the original

file in some systems (for example, replacing a symbolic link in a POSIX file system with a new file does not change the contents of the target file, but just the link itself).

Directory *binding mounts* and *union file systems* are attempts to combine contents from different directories. They allow to stack several directories in the same position of the directory tree, uniting their contents by means of altering the in-memory metadata structures (a low-level procedure that must be kept consistent with the physically stored metadata organization). As a result, they usually need to impose access restrictions on the original directories to guarantee the consistency, and they lack the ability to combine contents, with per-file granularity, dynamically.

A more dynamic approach is obtained via saved queries or *smart folders*. These are virtual folders whose contents are dynamically collected via the execution of some file search operation. Still, the functionality is limited: first, the search criteria do not usually allow arbitrarily complex queries and may work only on a few file properties; second, results are placed in a different folder, but they cannot replace an existing one (i.e. they cannot dynamically change the visibility and grouping of files in different views); finally, being the result of a search script, these folders are read-only and files cannot be added directly to them.

Given the limitations of existing mechanisms, our objective is to provide means to allow the dynamic combination of different directories in a flexible way, selectively uniting or hiding their contents and placing the result either in a different location of the directory tree or temporarily replacing existing folders depending on environment conditions; and, of course, modifications to the original directories should be reflected in the resulting view.

Unfortunately, conventional file system architectures make it difficult to provide such features. Therefore, the metadata virtualization framework presented in this thesis aims to provide the mechanisms to make those requirements feasible.

3.2 The overall concept

In general, file systems try to adapt to the new demands by specializing and targeting specific kinds of workloads. As a counterpart, there is a cost in terms of performance for non-optimized cases.

As mentioned in Chapter 2, conventional file system architectures suffer from the historical tightening of the relationship between name spaces, object attributes and physical data location, which was introduced to optimize accesses to physical media and, in particular, to improve the performance of file systems on locally attached disks.

Handling metadata at the low-level system layers exposes the inner limitations and complexities to the feature implementers, the applications and, at the end, to the users. On the contrary, decoupling the file system metadata (including the name space and the object attributes) from the actual data layout is a key tool to provide flexibility, integration and extensibility.

The foundation of this thesis precisely consists in introducing such a decoupled metadata layer into the file system architecture. In this chapter, we present a metadata virtualization framework that is able to provide to the applications a virtual name space and its corresponding set of file system object attributes, independent of the actual metadata in the underlying file system. This provides a convenient view of the file system for the users and transparently converts their requests into the appropriate low-level file system operations on the actual file locations.

Additionally, this technology makes possible to offer a well-defined interface where independent modules can be plugged in order to make specific file system extensions globally available or, simply, to provide new features.

It is important to note that the new techniques presented here are not aimed to build a new file system from scratch, but to increase usability by leveraging existing file systems, combining their strong points and mitigating their disadvantages. Even if some control over physical data layout is apparently lost, the multiple benefits of this technology largely compensates any possible negative effects.

The rest of this section discusses the principles guiding the design of the technology enabling metadata decoupling and mentions some of the tools that make these techniques feasible.

3.2.1 Design principles

Our main goal is to be able to decouple the user view of the file hierarchy (i.e. the name space) and the rest of the metadata management from the physical placement of data in the storage devices. In practice, we present the users a virtual view of the file system organization adapted to their needs while the actual layout is optimized for the underlying file system.

In order to facilitate the integration of our framework into existing systems, we need to be compatible with current standards, both official and 'de facto'. This enables our system to support unmodified conventional applications.

POSIX compliance is a strong requirement. This is still the dominant model for most applications, and our intention is not limiting the semantics of current systems; so, if POSIX semantics is required and the underlying file system supports it, then the system resulting from applying our framework should also be able to deal with it.

The same applies for *Microsoft NTFS*, which is the other largely used non-POSIX file system. *Microsoft NTFS* is the most commonly used file system in devices using the *Microsoft Windows* series of operating systems. Therefore, the development of our techniques should take it into account and also be compatible with its semantics.

The metadata virtualization framework should not deal with low-level data storage. There is no explicit management of disks, blocks or storage objects: our technology simply forwards data requests to the underlying file systems and suggests an appropriate low-level path when a file is created. Then, it is up to the underlying file system to take decisions on low-level data server selection, striping, object placement,

etc. In this sense, the techniques proposed in this thesis do not form a complete file system, but a tool to leverage the capabilities of underlying file systems.

On the contrary, our framework does take the responsibility on metadata management. By metadata we specifically mean access control (owner, group and related access permissions), symbolic link and hard link management, directory management (both the name space hierarchy and the individual entries) and size and time data for non-regular files (sizes and access time management for regular files rely on the underlying file system).

3.2.2 Enabling technologies

Nowadays, the state of the art of the computing technology makes feasible the implementation of a usable metadata virtualization framework on top of well-understood, robust and efficient base components, enabling multiple placement-independent views and seamless integration of file systems. The following paragraphs summarize the set of technologies that have been used as a foundation for the name space and metadata virtualization framework proposed in this thesis.

Memory. Modern computers have plenty of memory to use. When the foundations of file systems were designed, memory was a scarce resource, and its contents had to be frequently replaced by means of slow accesses to permanent storage media, so it was critical to get the most of data and its related metadata together.

Today, the size of files has grown, as well as their number. Nevertheless, the size of individual file metadata is still relatively small, and the size of the aggregated metadata from the active working set at any given time is not unbounded. As a consequence, the metadata of a usual working set can be kept for a long time in a fraction of the available memory (the experiments with our prototypes indicate that an average computer with 4 GiB of RAM can easily manage the metadata for an active working set of a few hundreds of thousands of files), so that there is no need to mix them with the actual data for combined storage and retrieval from media; on the contrary, separating them enables the use of more efficient techniques to handle the metadata. In particular, our studies indicate that it is feasible to organize the metadata as a series of light-weight database tables and take advantage of database engines to speed-up the metadata processing.

Distributed consistency mechanisms. Large aggregations of computers are becoming common; therefore, there has been an effort to develop distributed coordination techniques. It has been observed that, despite the increasing number of collaborating nodes, there is a considerable amount of locality in data accesses, and aggressive caching and intelligent synchronization algorithms can reduce the costs of shared metadata when integrating the data views from different computers.

Local storage capacity. Despite their limitations as a standalone storage media, locally attached disks have continued increasing their capacity and speed. Therefore, they constitute a valuable asset that can be used as a giant cache for remote storage systems. Considering the usual locality and access patterns for personal computers, it is feasible to devise a caching policy that converts most of the remote accesses into local cache hits. This is a key element to achieve a seamless integration between local and remote file systems.

Interception and extension support. The recent improvements in virtualization techniques provide us with mechanisms to transparently alter the users view of a computer system. The refinement of these techniques, the advance of the hardware capacity, and also the increased support from the operating system kernels, have turned a costly and cumbersome technology in a widely extended and completely usable mechanism. Specifically for our needs, the most extended operating systems offer ways to intercept every aspect of file system operation and introduce new features and services, from simple extensions up to providing whole file system-like environments to interact with varied systems (the *Filter Manager* [MSDN 2012] in *Microsoft Windows* and FUSE [Szeredi 2005] in *Linux* are good examples).

Processor capacity. The processors speed and capacity have increased at a higher ratio than the rest of the computer components. This means that there are more processor cycles available in a given unit of time, and it is possible to execute more complex software stacks without noticeable overheads. In particular, virtualization and file system interception techniques can be used to add new features at a very low cost in absolute time.

Lightweight tools for embedded systems. The emergence of embedded systems with relatively small individual capacities, and their need for interconnection, have brought forth the development and improvement of specialized and highly efficient light-weight programming models and tools. The Erlang/OTP [Erl 2013] environment is an example of such tools. Interestingly, file system metadata activity can be modelled in a very similar way to embedded telecommunications systems (thousands or millions of small elements that have to interact in a consistent way); therefore, the same tools provide a robust foundation for a decoupled metadata engine.

Network capacity. Last but not least, the capacity of current communications systems (from ADSL to 3G, Wi-Fi, corporate networks and high-performance interconnects) is enough to permit the transfer of metadata (which is relatively small) and a continuous flow of file data updates. This enables the maintenance of a local cache with the current working set, the reception of updates from remote servers and the progressive back-up of modified data, favouring the integration of local and remote storage.

3.3 Architecture

The architecture of our system must provide the means to separate the name space, the file system object attributes and the global metadata from the internals of a particular file system. This will enable us to offer virtual views of the file system organization that do not necessarily match the actual layout of files in the underlying file system and, additionally, to transparently divert files into different storage systems.

In order to achieve this goal, different architectures are possible. The two main options are modifying and extending a conventional file system to provide the desired new features, and building a layer on top of the original file system, complementing its functionalities. In this section, we discuss both possibilities.

3.3.1 Extending a traditional file system

A first approach to provide decoupled metadata management is to extend a traditional file system to offer such functionality. Figure 3.3 shows a diagram with the extensions needed to make this transformation.

Compared with the basic structure depicted in Figure 2.1 (Chapter 2), we may see that some of the elements are equivalent (for example, we still have name space information, location information and file-specific metadata). However, apart from the physical metadata needed to handle the physical storage devices, we have to add the corresponding information for the virtual views. This involves an increased level of complexity because the file system components must be able to determine which set of metadata must be used for each request (while maintaining the consistency among the different metadata sets).

The other important change is that, in order to be able to integrate multiple file systems under the same name space, the extended file system need to be able to interface with those other file systems for data storage, instead of relying only on its own low-level storage devices. This is represented in Figure 3.3 by the data file system interface and the underlying file system, which lays side by side with the interface elements required to access a physical low-level storage device.

As a whole, we have the physical metadata and location information to handle the attached low-level storage devices (as in the traditional file system) and the virtual metadata which is used to integrate multiple underlying file systems and low-level storage devices and to provide different views of their combined contents to the user applications.

The following items describe in more detail the changes affecting to each element:

Application file system interface. In order to be transparent and let unmodified applications work on the new system, the interface between the applications and the file system engine must be compatible with the non-extended version. However, it is possible to incorporate new calls to the interface to make use of the new

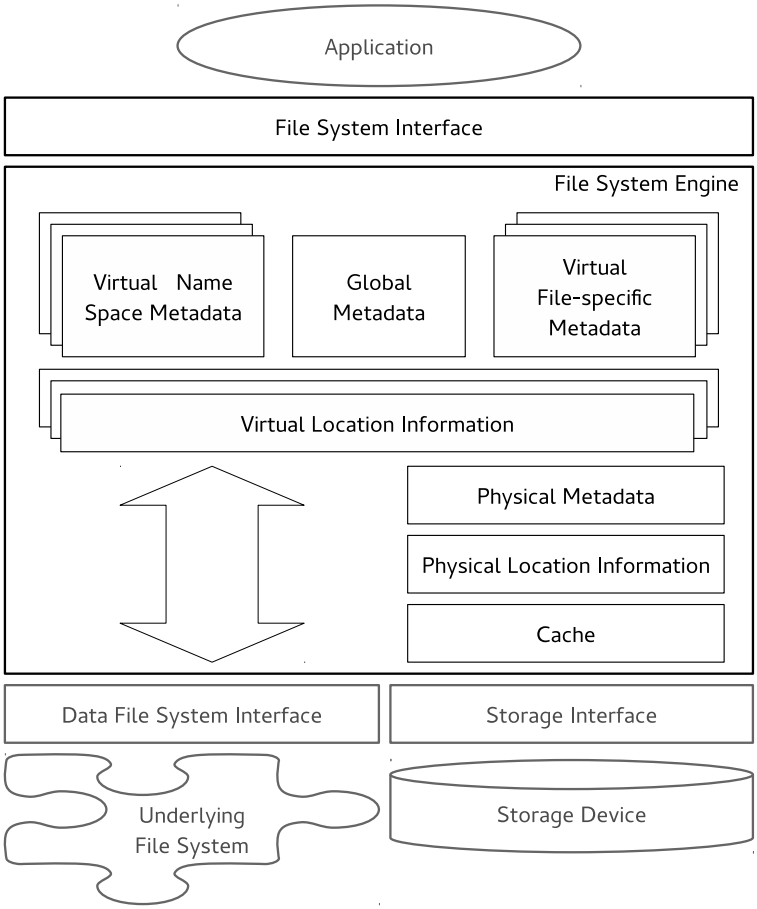

Figure 3.3: Extending a basic file system for decoupled metadata management.

features (for example, to choose a specific name space to perform a particular operation, or to set the rules to decide automatically which set of metadata should be used at each moment).

Name space metadata. Traditional file systems as the ones described in Section 2.2 offer a single view of the directory tree to the user; thus a single name space is provided. In our extended file system, we need to maintain a set of name spaces to be able to provide multiple views to the user.

It is important to note that these name spaces are not completely unrelated to each other. Indeed, the different name spaces export different organizations of the same set of objects in the underlying storage; therefore, certain changes made to an object through a particular name space will be reflected in other name spaces. For example, a particular semantics may allow deleting a file from a particular name space; in that case, the file system should locate the entries in other name spaces referring to the same file and also remove them.

File-specific metadata. File system object attributes managed by our extended file system fall into two groups. On one side, there are the physical attributes required to manage the data residing in the low-level storage device; these are essentially equivalent to the file-specific metadata in the basic file system structure. On the other side, there are the so-called virtual file-specific metadata attributes, which are the ones actually exported to the users.

The virtual attributes may differ from the low-level ones and provide support to unify the underlying data systems. This is useful to integrate different storage devices and/or underlying file systems under a common view. For example, virtual file-specific metadata may provide attributes supporting ownership and access permission information which are not available in the underlying physical metadata (such as the metadata in a FAT-based file system).

Additionally, it is possible to store different sets of values to virtual attributes of a specific object, associated to the name space through which they are accessed. For example, a file meant to be read-only under some views and modifiable in others could have two sets of access permission values to reflect its properties under each view.

Location information. As with the file-specific metadata, our extended file system has to deal with two different types of location information: the information required to physically access data in the low-level storage device and the virtual location information meant divert access from the common interface to the multiple underlying file systems and low-level storage devices.

The nature of both types of location is quite different. While physical location information is device-dependent (e.g. block address and offset in a block-based device), the virtual location information is device-independent and typically contains a reference to a particular underlying storage system and locator inside such data repository (e.g. a file system identifier and a path inside that file system); in the latter case, the underlying file system is assumed to contain the low-level location information to actually access the required physical data.

Global metadata. The global metadata does not have significant changes compared to the basic file system structure from Section 2.2. The information provided to the user applications is the same, and the only difference resides in the fact that, in a extended file system architecture, it must provide a sensible aggregation of the global metadata from the underlying file systems and storage devices.

This model of extended file system has the necessary elements to provide multiple views of the underlying file systems objects to the user applications.

Nevertheless, integrating all the elements is not an easy task. The file system engine has to maintain and coordinate multiple name spaces with their associated sets of virtual attributes and virtual location information. At the same time, this virtual information has to be combined with the handling of physical metadata to manage the attached storage devices and, additionally, it must also incorporate the capacity

to use other underlying file systems. Mixing all this elements may lead to complex implementations that are difficult to maintain and prone to error situations.

Therefore, a more suitable approach would consist in splitting the system into several modules addressing different functionalities. Fortunately, the model reveals a clear distinction between the virtual view management (including the virtual name spaces and the virtual attributes) and the underlying data management (composed of storage devices and the associated management information, as well as additional underlying file systems).

In the next section, we present an architecture that structures the elements of the extended file system into separated modules that form a *Metadata Virtualization Layer* on top of conventional file system interfaces.

3.3.2 Introducing a metadata file system layer

A practical approach to offer metadata decoupling and all its benefits consists in building a specialized *Metadata Virtualization Layer* that runs on top of other file systems. Figure 3.4 presents the architecture for such a system.

In this architecture, the application requests are captured by an *Interception Engine* that publishes a conventional file system interface. The requests are directed to the *Metadata Manager*, which translates them into adequate requests for the underlying file systems. The *Data Manager* takes these translated requests and actually forwards them to the underlying file systems, taking into account their specific characteristics.

This generic procedure may be altered in some situations. For example, some requests can be completely resolved by the *Metadata Manager* without reaching the *Data Manager* if they involve just virtual elements. In other cases, a request can bypass the *Metadata Manager* and be immediately diverted to the *Data Manager* if it has nothing to do with metadata management (e.g. an actual `read` or `write` data request).

The elements in this architecture are essentially the same as the ones discussed in Subsection 3.3.1. The only difference is that, given that this metadata layer is deployed on top of other file systems instead of raw low-level storage devices, it does not need to keep physical metadata and location information.

This way, the components in the specialized metadata layer are independent of the existing file systems. Though the lack of knowledge about the low-level storage systems may introduce a greater overhead because the metadata management is not so tightly integrated with the underlying file system, it also offers greater flexibility, as it is not necessary to maintain compatibility with specific file system structures.

The tasks of each component are summarized in the following items:

File system interface. The file system interface is associated with an *Interception Engine* to capture the requests issued from the applications to the actual file system. This module is responsible for redirecting the original requests to either the *Metadata Manager* or the *Data Manager* of the *Metadata Virtualization Layer*.

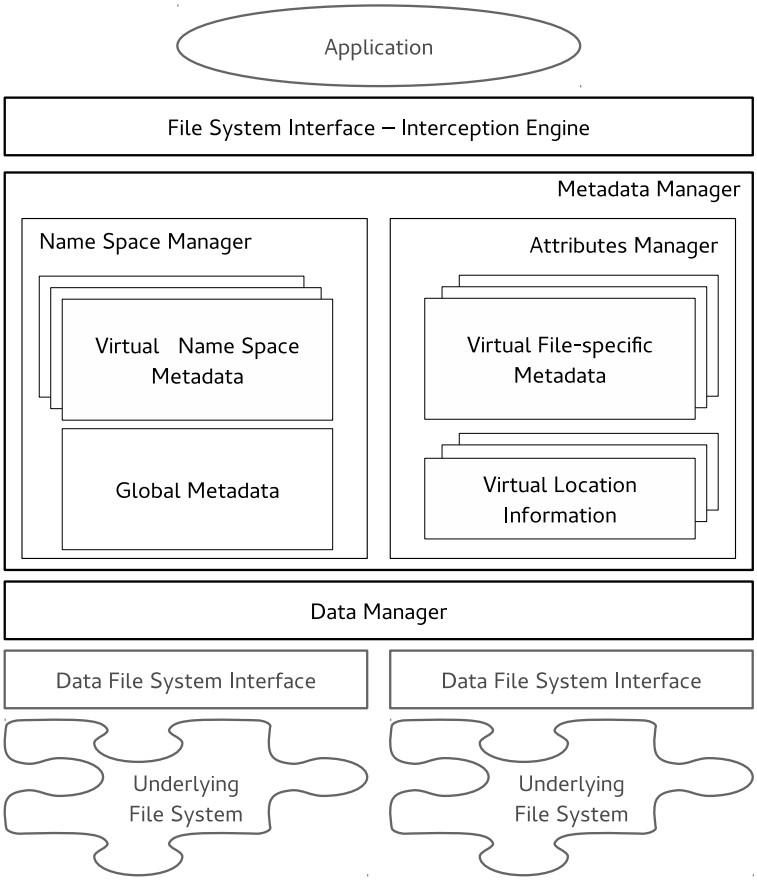

Figure 3.4: Deploying a *Metadata Virtualization Layer*.

Metadata Manager. This module is the responsible for providing virtual views of the underlying file systems to the applications. It can be conceptually divided into several subcomponents handling the different types of metadata:

Name Space Manager. The *Name Space Manager* subcomponent is in charge of translating paths in a directory tree into internal identifiers used by the *Metadata Virtualization Layer* to reference the actual objects. Additionally, it also handles requests related to directory content management (including, for example, directory listing, file renaming and file existence checks). Directory content operations are completely independent of the actual directory layout in any of the underlying file systems and storage mechanisms; therefore, they can be completely performed by the *Name Space Manager* without needing to reach the underlying file systems.

The *Name Space Manager* has also to be able to choose the right name space (from the available ones) to perform the required operations.

Attributes Manager. The *Attributes Manager* subcomponent is responsible for handling the individual file system object attributes. The attributes provided

by the *Metadata Virtualization Layer* are very similar to those found in traditional file systems and may include items such as security elements (owner, group, access permissions and control lists), statistical information (access times and other utilization indicators), infrastructure-related fields (such as file system identifiers, hard link counters and symbolic link paths) and extended attributes.

Location information is considered a kind of attribute and also handled by the *Attributes Manager*. The main difference with the location information in a traditional file system architecture (such as the one described in Section 2.2) is that the locations in the *Metadata Virtualization Layer* are device-independent. Because of not being restricted to a specific low-level format, the *Metadata Manager* can be easily extended to provide advanced location functionalities. For example, several locations containing replicas could be associated to a file's data or to fragments of it, enabling replication across several underlying file systems; in a similar way, a single file data could be transparently stripped across several storage systems.

Data Manager. In a traditional file system, the internal data management modules are in charge of reading the data (or writing the modifications) in the low-level storage devices. The *Data Manager* in the *Metadata Virtualization Layer* performs a similar task, but working on top of another file system instead of a raw device. Actually, its function is to provide the *Metadata Manager* with a unified interface that eliminates the differences between the underlying file systems. Fortunately, the advantage is that, being higher-level entities, the differences between file systems are easier to overcome than the differences between low-level devices.

Depending on the operation and its environment settings, the *Data Manager* may let the request go forward to an underlying file system, it may alter it (for example, encrypting or compressing data), it may decide to split the contents into different underlying files, or it may even duplicate the data or redirect it to several storage systems.

The *Data Manager* can also perform operations on the underlying file systems autonomously from applications, enabling advanced features such as automatic maintenance, information gathering or data reorganization and migration.

In summary, a *Metadata Virtualization Layer* is able to handle the elements mentioned in the extended file system model, but in a simplified way (as it delegates all low-level aspects to the underlying file systems). Being loosely coupled with the underlying storage may increase the overhead risks but, at the same time, the lack of integration allows more robustness and flexibility, due to not having to abide by low-level compatibility issues.

As a consequence, in this thesis we have decided to use the *Metadata Virtualization Layer* as the reference model for our developments. Section 3.5 discusses details about several aspects that must be taken into account in order to convert this model into an implementable and functional system.

3.4 Terminology

Though virtualization techniques are currently applied to a multitude of areas, it is also true that the precise meaning attributed to the term 'virtual' usually varies from case to case. Therefore, we deem important to specify how 'virtual' and other related terms are going to be used in this work.

We will refer to the directory hierarchy of the underlying file system as the physical name space, which will be composed of 'physical' files and directories.

Oppositely, the *Metadata Virtualization Layer* will provide alternate views of the file system layout that may differ, at least in some points, from the physical name space. We will designate these alternate views as virtual name spaces.

We will use the expression 'virtual file' to indicate an entry in the virtual name space representing a file, and 'virtual directory' to refer to a folder in the virtual view. The expression 'virtual object' generically refers to objects related to a virtual name space (either a virtual file, a virtual directory, a symbolic link or any other entity), and 'virtual entry' indicates the name of that object (i.e. the last component of its path).

In the implementations discussed in Chapter 4 and Chapter 5, virtual files are usually backed by physical files in the underlying file system, while the rest of virtual objects does not have corresponding underlying objects.

Nevertheless, in some occasions a virtual name space may expose part of the physical name space. One of the ways to achieve this feature is to match a virtual directory with a corresponding physical directory; the resulting entity can expose the physical directory contents through the *Metadata Virtualization Layer* and it is called an anchor directory. (This particular use case is discussed in Chapter 5.)

3.5 Practical applicability aspects

One of the goals of this thesis is to provide a metadata virtualization framework able to offer multiple views of the file system organization to applications and, at the same time, able to integrate several underlying file systems seamlessly.

In Section 3.3, we have outlined the changes and additions required, compared to a conventional file system architecture. These modifications affected both the nature and type of the metadata information as well as the components that handle them.

Nevertheless, file systems are complex entities and converting the proposals into fully functional systems involves taking care of a multiplicity of aspects that may impact the feasibility of practical implementations.

In this section, we identify the relevant aspects that affect the implementation of a file system *Metadata Virtualization Layer* and discuss the possible ways to deal with them.

3.5.1 Dealing with underlying file system information

The *Metadata Virtualization Layer* relies on an underlying file system to keep the actual data. This data is usually represented in terms of files and other objects in the underlying file system.

These 'underlying' objects have attributes that may be unrelated to particular name spaces or virtual metadata items (for example, the size of a file or the time of its last modification is independent of the name space through which it is being accessed — and changes to these values can be visible across multiple name spaces). However, updating the attributes of the underlying objects may require complex protocols (for example, all nodes in a distributed file system should be made consistently aware of a modification of the size of a file).

Usually, the underlying file systems are specially designed to take care of these attributes. Therefore, duplicating such mechanisms in the *Metadata Virtualization Layer* does not offer clear benefits. For this reason, the implementations of the metadata virtualization framework described in this thesis delegate the management of attributes tightly related to underlying objects to the underlying file systems.

On the contrary, 'virtual' objects (e.g. files in a virtual view) are actually associated with high-level virtual attributes with a more restricted scope and which effectively depend on the particular entry or name space used to access them. These attributes are kept in the *Metadata Virtualization Layer* instead of the underlying file systems.

So, we establish a distinction between the underlying objects and their attributes (actually stored in the underlying file systems) and the virtual objects accessible through a virtual view, which may have virtual attributes that can be independent of the related underlying objects.

The coordination of metadata access to underlying objects and their attributes, as well as the implementation of any consistency protocol required to coordinate multiple underlying file systems, is a responsibility of the *Data Manager* component from the architecture described in Subsection 3.3.2 (see Figure 3.4). This module is also responsible for accessing the actual underlying file data whenever necessary.

The handling of virtual attributes not directly related to underlying objects is carried out by the *Metadata Manager* component. This module may able to use simplified protocols and consistency mechanisms to access and update such attributes, resulting in optimized performance (for example, changing the permissions of a file in a virtual view which is accessible from a single node does not require any distributed consistency mechanism, even if the underlying file system is a distributed file system deployed across multiple nodes).

An actual example of this management separation is the implementation of a *Metadata Virtualization Layer* providing POSIX semantics on top of an underlying file system. A request to access certain file attributes (e.g. `utime` to modify a file's access and modification times, or `stat` to retrieve a file's size) requires accessing the underlying file related to the 'virtual' file in the specific view being used. In

the case of access and modification times, the reason is that these values are tied to actual underlying activity, and the underlying file systems usually handle them well; duplicating them in the *Metadata Virtualization Layer* and keeping them synchronized with the underlying file attributes is an unnecessary cost. A similar situation occurs for the file size requested via `stat`: it is much cheaper to check with the actual underlying object when needed, than to keep a value in the *Metadata Virtualization Layer* and maintain it synchronized at all times.

Attributes related to objects not associated to any underlying object (such as directories and symbolic links) are not bound to this attribute delegation policy. As they are independent of the underlying file system, their size and time-related information is directly maintained as attributes of a virtual object in the views provided by the *Metadata Virtualization Layer*.

The actual location of the files in the underlying file system is stored by the *Metadata Manager*. This location information is an opaque value, in the sense that it is to be interpreted by the *Data Manager* to reach the actual file data.

In a general case, the location information is composed of a storage repository identifier and a reference to the underlying object inside that particular storage (in a simple case this could be mapped to an underlying file system identifier — or a *mount point* — and the path of the target file inside that file system).

In more complex scenarios, the location opaque value can be used to keep arbitrary information allowing the *Data Manager* to locate the actual data. For example, a file in the user's view of the file system can correspond to the concatenation of several files, possibly in different underlying file systems (file striping); then, the location information would contain information about such files indicating, for example, the file systems where they reside, the path of the files and, if necessary, the offsets of the user file represented by each underlying file.

A similar approach can be used to handle file replication (either of the complete file or part of it). In this case, the location information would contain a list of the replica locations.

3.5.2 System-specific identifiers

Typically, users refer to an object in a file system by means of a path, which represents the location of the object inside a particular name space (e.g. the sequence of folders that need to be traversed in a directory tree to reach the object, plus the name of the object inside the final folder).

Internally, the file systems typically refer to objects by means of numerical identifiers. As a matter of fact, one of the main tasks of the *Name Space Manager* in a file system is to provide means to translate a path provided by a user into the internal identifier referring to the actual file system object structures.

POSIX-based systems assign an internal identifier (the so-called *i-node number*) to each file system object. This identifier is assumed to be unique for a given file

system during the life time of the object. Furthermore, in some environments, it is sometimes recommended to not immediately re-use the identifier once the object has been destroyed, in order to avoid conflicts due to stale references to that identifier (for example, in case it has been cached remotely and a network issue does not allow to guarantee that the cached value has been invalidated).

The POSIX *i-node number* must be made available to the system and to the applications using the file system. Therefore, if the *Metadata Virtualization Layer* has to provide POSIX semantics, it needs to generate the proper *i-node numbers* for the objects in its views. The internal identifiers from the underlying file systems cannot be directly re-used for this purpose: for example, there may be several underlying file systems (with possibly duplicated *i-node numbers*) or, simply, some of the underlying file systems may not be POSIX-compliant and not provide compatible identifiers.

The valid range for *i-node numbers* is somewhat dependant on the operating system because they have to be fed back to it. For example, one of the possible requests from the VFS component in a *Linux* system to a POSIX file system is a request for translation from an entry name into an *i-node number*; the provided *i-node numbers* are stored by the *Linux* kernel to be directly used as identifiers in future requests [Bovet 2008].

The *i-node numbers* are typically represented as integer values. Most systems use integers with 32 bit or 64 bit lengths, while a few can use larger integer sizes. It is usually assumed by the system that an *i-node number* represents a unique object in a given file system, and that such *i-node number* will not be reused until the previous object is destroyed. Nevertheless, some systems allow a more flexible scheme: an *i-node number* could be reused as far as the underlying system keeps no reference about the previous object. To this end, *Linux* VFS instantiations use a `forget` request from the kernel to notify that some references to a particular *i-node number* have been released. Additionally, some systems use a *generation number* associated to the *i-node number* which is increased each time the corresponding *i-node number* is re-used; nevertheless, this is not usually exported outside the file system.

The implementations of the *Metadata Virtualization Layer* internally identify and refer to virtual objects accessible through views by means of identifier (internally called *inums* — a contraction of "identification number"). These identifiers are unique, at least for the life time of the object; they are handled internally, so they do not necessarily match the *i-node numbers* that must be provided to POSIX-based systems. Therefore, an *inum* value has to be mapped to a locally unused *i-node number* adequate for the user.

Of course, if it can be guaranteed that the possible range of *inum* values is always a subset of the values accepted as *i-node numbers* by the user's operating system kernel, a possible map consists in directly using the *inum* value as the *i-node number*.

Nevertheless, it is possible that the *inums* do not match the valid *i-node number* range (for example, some of the implementations derived from this work use 128 bit identifiers, while current *Linux* kernels support, at most, 64 bit *i-node numbers*). In this case, the *Metadata Virtualization Layer* has to explicitly map local *i-node numbers* to

internal *inums* in order to be able to translate one into the other, in both senses. The data structures supporting this translation can be local and even temporary (e.g. they could be cleaned on system restart because no assumptions are made regarding the persistence of the local identifiers once the system has been terminated).

The identifier translation mechanisms also help in situations when the limited size of *i-node numbers* is a handicap to provide the required functionalities. For example, the *Metadata Virtualization Layer* may interface to an underlying file system with a very large set of objects exceeding the local identifier range (as could be the case of a distributed or a globally available cloud-based file system); in this case, large identifiers can be used to identify the underlying objects, but only those in the 'active' working set would be assigned local *i-node numbers*, keeping them in a reduced value range. Another reason for having arbitrary large identifiers not fitting in the locally available ranges is the use of algorithms for distributed generation of universally unique identifiers; such identifiers are usually large, in order to reduce the chances of clashing.

The implementations of the metadata virtualization framework that we have developed as part of this thesis use different approaches depending on the environment where they are going to be used. For example, the HPC implementation discussed in Chapter 4 is mainly designed to boost specific application runs; therefore, it uses a 64 bit monotonically increasing counter as *inum*, which can be reset when a particular workload is completed and, due to its size, can be directly mapped into local *i-node numbers*. On the contrary, the implementations described in Chapter 5 are meant to deal with a potentially large number of files in the cloud that need persistent identifiers; therefore, they use 128 bit *inums* which are mapped to local identifiers via non-persistent local hashes.

3.5.3 Using databases for metadata storage

Most file systems store information related to the file system objects and the name spaces using specific structures (for example, *i-nodes* and directories) which are usually grouped together (e.g. the *i-nodes* are packed together in certain sections of disks, and directories are represented as huge lists of entries stored in consecutive blocks in a disk).

When developing implementations of the *Metadata Virtualization Layer*, we decided to use a different approach. The data required by the different components is organized as records in large tables indexed by a key value (or, at most, the combination of a few values) without any explicit grouping. Therefore, any hash-like structure (a set of entries accessible via a key) would be adequate for storing the necessary pieces of data, such as entry information for the *Name Space Manager* module, object attributes for the virtual *Attributes Manager* and other pieces of information (such as paths for the symbolic link support).

Databases fulfil the functionality of hashes, with some additional features such as multiple keys, or atomic transactions, for example. Moreover, advanced database

engines also provide fault tolerance mechanisms or distribution support. For these reasons, our implementations of the *Metadata Virtualization Layer* use databases as back-end for the data required by the different components.

Specifically, we have used *Mnesia*, a database that is part of the Erlang/OTP suite, to support some of the distributed implementations of the *Metadata Virtualization Layer*. Said database is optimized for simple queries in soft real time distributed environments and has built-in support for transactions, fault tolerance mechanisms and data distribution. An interesting property is that said database is able to keep and manipulate its tables in memory (for efficiency) while sending the information in the background to persistent media (safety).

Despite the fact that databases may seem too cumbersome for the fast pace required in file system implementations, we have found that adequately tuned databases do not have a substantial impact in file system metadata performance and, at the same time, they provide a level of flexibility that is not present in block-based metadata storage in traditional file systems. The results obtained with the *Metadata Virtualization Layer* implementations using databases will be shown in the next chapters.

3.5.4 Metadata volatility

It was mentioned in Chapter 2 that the trends in traditional file systems tended to group all metadata (i.e. all attributes) together so that, being usually small, they could be packed and transferred in an efficient way.

Nevertheless, such packing also has negative consequences: for example, updating an attribute usually involves locking the attribute set in order to keep the consistency but, at the same time, this may prevent another process from accessing an unmodified attribute while another attribute is being updated (causing lock contention).

This situation is often aggravated by the fact that the values of the different attributes of file system objects have different levels of volatility, from immutable to highly volatile values.

In distributed file systems, the impact is more noticeable. A common technique in distributed implementations is to use a cache mechanism to avoid unnecessary transfers and, usually, this involves keeping a local copy of the attributes from a remote repository. Then, issues arise when frequently used attributes with low volatility levels are packed and handled together with rarely needed attributes with high volatility: changes in the rarely needed attributes may require the invalidation of the group of cached attributes, removing also the highly needed — and probably unmodified — low volatility attributes (thus, limiting the effectiveness of some techniques such as caching).

We may find a concrete example in POSIX-based file systems. POSIX-based file systems use the *i-node* structure to keep all the information related to a file system object. Such information include some immutable information that is needed very often (e.g. the *i-node number*, and the file object type), information which is also used

very often (possibly by multiple clients) and rarely updated (e.g. the owner, the group, and the access permissions) and information which is rarely needed but updated very often (e.g. the file size, or the access and modification times).

A solution for said type of situations is to classify and group the attributes in 'volatility levels' according to how often they are updated (and possibly also taking into account how often they are needed). Using such classification, the *Metadata Manager* in a *Metadata Virtualization Layer* can treat each set of attributes in a specific way, according to their characteristics (for example, it may use different caching policies — e.g. with different lease times).

The implementations of the metadata virtualization framework presented in this thesis are designed to take advantage of the volatility classification and break the traditional grouping of attributes. Unfortunately, some traditional interfaces limit the effectiveness of this approach; for example, in the POSIX standard, `stat` is the most commonly used operation to retrieve the attributes of a file system object — and it retrieves all the attributes at the same time (the application just ignores the attributes that it does not need). Nevertheless, other modern file systems, not strictly POSIX-compliant, offer tools to access attributes individually and can take advantage of the different attribute behaviours; this is the case, for example, of *Microsoft NTFS*.

3.5.5 Metadata caching

In cases where the modules of the *Metadata Virtualization Layer* are distributed, caching is a useful technique to avoid some communication among components and improve the performance. Nevertheless, care must be taken to keep the cache synchronized with the actual data.

The components of the *Metadata Virtualization Layer* could maintain a cache by themselves, or try to take advantage of mechanisms already implemented in the base technology. For instance, FUSE, when used as one of the base technologies, allows the *Metadata Manager* module to specify the expiration times for directory entries and some of the file system object attributes cached in the operating system kernel.

Unfortunately, we observed that most of the available caching mechanisms in the base technologies were not adapted to our specific needs. In particular, at the time of the implementation development, FUSE did not offer means to arbitrarily invalidate cached items in the kernel, which was a feature required to maintain consistency when metadata was modified from a remote node in a distributed deployment.

Eventually, a specific cache mechanism was implemented to keep object attributes and entry-related data locally, in order to reduce interactions with remote nodes. The cached data was provided with a limited-time lease and, additionally, the provider could issue specific invalidation requests whenever the affected pieces of metadata were going to be modified by a different node.

A particular detail of the cache implementation is that lease handling can be decoupled from data requests; in other words, the lease is not sent back to the caching

component together with the response to a request: instead, the response is sent as fast as possible from the component having the desired data, and it triggers a decoupled concurrent mechanism that will end up in the corresponding lease being sent to the requesting component afterwards.

Though this might seem counter-intuitive (as the data is sent twice to the client), we observed that it resulted in better response times when back-end components of the *Metadata Virtualization Layer* were executed in low-end systems (where processing speed was slow compared to network speed), while the cache efficacy was not affected in a significant way. When tested in systems with higher processing capacity, differences between decoupled and non-decoupled lease handling disappeared.

The number of leases granted for a specific piece of data is limited. This limitation helps to keep the synchronization and invalidation costs bounded. Beyond this limit, the cost of synchronization would outweigh the benefits of caching, so the system moves to work as a no-cache system.

3.5.6 Metadata represented as active objects

The separation between metadata and the underlying file system introduced by our *Metadata Virtualization Layer* has allowed us to explore alternative possibilities for metadata representation.

In particular, this subsection discusses an unconventional method consisting in representing the file system objects as active objects (i.e. threads or lightweight processes) controlling their related pieces of metadata. This method has been particularly effective for the implementation of a *Metadata Virtualization Layer* designed for distributed HPC environments (see Chapter 4).

The basic idea consists in creating a process for each virtual object in a view, with such process having exclusive rights for performing updates to the attributes of the corresponding object. The following paragraphs explain the rationale.

In modern computing environments, it is usual that several users or applications issue requests to the file system simultaneously. Therefore, different requests may cause conflicts because they need to access the same object in different ways (for example, trying to create a subdirectory while the parent directory is being listed). Moreover, it is not uncommon that certain requests may affect several objects (for example, a file removal involves destroying the target virtual object representing the file and modifying the object representing the parent directory).

A well known technique to deal with these situations is using explicit locking mechanisms: the code implementing a request tries to acquire locks on the required data structures; if the locks are granted, the code may continue; otherwise, it has to wait until the current lock owner releases them. Nevertheless, in situations with lots of activity, or when large groups of structures have to be manipulated, this technique can lead to lock contention. Even with no conflicts, acquiring and releasing locks may produce some overhead.

Representing the file system objects as processes allows using a novel approach to deal with concurrent activity: instead of having to lock data structures before using them from different applications, each data structure has an assigned thread or process which is the only one with permissions to modify the data.

In the implementations of the *Metadata Virtualization Layer*, the *Metadata Manager* associates one of these processes to each virtual object representing a file system object in a particular view. Such exclusive process is the only one with permissions to modify the attributes of the virtual object associated to it (e.g. the access permissions, or the location of the actual file in the underlying file system).

The fact that only one process is allowed to modify a specific attribute of a particular object and a strict order for updating attributes allow to avoid the need to enclose certain operations in large and costly synchronization procedures (such as database transactions), even when attributes are read from non-owner processes. For example, when a new file is created, the attributes of the virtual object representing the new file are updated first, then the object corresponding to the parent directory is also updated and, finally, a new entry is added to the name space; therefore, if an external process is able to reach the object following a path, it is sure that the related attributes have been already set and are consistent.

There are some database transaction properties that are not maintained with this mode of operation; namely, there is no automatic roll-back capability when errors prevent the completion of a complex operation. In these situations, the construction of the framework guarantees that the resulting view of the system is consistent, but it is possible that stale structures remain allocated. Clean-up of such structures is enabled by logging the objects involved in error situations and making them candidates for a posterior check-and-clean swept.

The method to operate with processes instead of data structures is as follows. Whenever a request arrives to the *Metadata Manager* to update a virtual object, it is forwarded to the process associated to the target virtual object. If the request involves a modification, then the process performs such modification (e.g. by updating a repository containing the actual attributes such as, for instance, a database); otherwise, the requested data is included in the response. Once the process sends the response back to the caller, it waits for the next request. Note that, as the process is the only updater, the pieces of metadata can be safely cached in the process local memory without consistency risks.

In cases where several virtual objects are involved (for example, removing a file, which may affect the virtual object representing the file itself and the virtual object representing the parent directory) the request is sent to the 'main' object process (the parent directory) which then interacts with the process of the other object involved (the file to be removed) by any convenient means (e.g. via message passing).

One of the implementations of the *Metadata Virtualization Layer* presented in this thesis is based on the Erlang/OTP environment, which provides support for extremely lightweight processes (a single Erlang node may handle up to a few million processes).

In such environment, the data related to virtual objects (including attributes) was kept in a repository based on the *Mnesia* database. This particular environment was used for the implementation described in Chapter 4, deployed on top of distributed file systems for HPC environments, where the potentially huge number of files in a working set required the ability to handle a large number of active object processes.

In other environments, such as the *Microsoft Windows*-based systems described in Chapter 5, dedicating an exclusive process to each object can rapidly exhaust the available resources (due to the heavier nature of threads in that operating system). In these circumstances, a single process may be used to handle more than one virtual object. In order to achieve similar benefits to having a dedicated process per object, the subsets of virtual objects handled by different processes are disjoint (i.e. two different processes with any overlap in time do not have ownership of the same virtual object, even at different times).

In summary, contention problems and costly synchronization mechanisms can be reduced by using a dedicated process for each virtual object. This mechanism can improve the performance and scalability of the *Metadata Virtualization Layer*, especially when this layer operates in environments where several users may issue simultaneous requests (for example, parallel and/or distributed systems). In particular, the Erlang/OTP framework provides support for transparent process migration across a set of node, facilitating the scalability of a *Metadata Manager* based on this technology.

The processes associated to virtual objects can be activated or deactivated when necessary, so that not all virtual objects must have an associated active process at all times. The process can be eliminated and disappear after a period of inactivity, and be re-created when the associated object is needed again.

The responsibility for the activation or deactivation is shared by the *Metadata Virtualization Layer* components and the object processes themselves. The ability to activate and deactivate object processes as needed allows the *Metadata Virtualization Layer* to maintain a greater control on the amount of resources being used to handle virtual objects. For example, when there is a request to update a virtual object and its dedicated process does not exist, a new process will be created to represent such virtual object. Once the process is created, the *Name Space Manager* or the *Attributes Manager* components can forward the request to the new process.

The decision to deactivate a dedicated object process depends on the availability of resources and an inactivity time threshold. The inactivity time threshold for different object processes may also be different, being adapted to different usage patterns. Another reason for deactivation is provided by the file system semantics itself: for example, a requirement to deactivate an object process representing a directory in a file system (e.g. because of a removal operation), may cause that process to request the deactivation of other dedicated processes associated to virtual objects representing the children of the first directory.

In our implementations of the *Metadata Virtualization Layer*, each object process is able to deactivate itself when certain conditions occur. The advantage of having an

object process to deactivate itself is that each process may keep its own state and track the conditions that make it a candidate for deactivation, instead of moving the burden to an external component having to maintain global information about all the object processes and their state and conditions in order to deactivate them when necessary.

3.5.7 Stateless large directory handling

Directory listing is one of the common operations that users perform on a file system. This operation returns the contents of a given folder in a directory tree. The *Name Space Manager* is the responsible for generating a list of entries contained in a given folder in a particular view upon request.

As the *Metadata Virtualization Layer* may receive several requests simultaneously (possibly from different users), it may occur that entries from the set being listed are to be removed while the listing is in progress, or where new entries that could appear in the listing are to be created while the listing is in progress. In these situations, most systems demand a certain degree of consistency in the listing; usually, this means, at least, that the listing should not contain the same entries more than once (i.e. entries with the same name, regardless of them referring to the same or to different objects). In other words, removed entries may or may not appear in the listing, and newly created entries may either not appear at all, or appear in the listing as far as a previously existing entry with the same name (but already removed) was not already listed.

A possible approach consists in using a lock mechanism (or a database transaction if the entries are stored in a database) to prevent modifications to the affected entries while producing the listing. For short listings of entries, this may be effective; nevertheless for very large listings, this has a number of problems: for example, generating a large listing may take a considerable amount of time, causing performance problems due to delays to other requests waiting for the locks to be released. Also, in distributed environments where the listing may have to be transferred between components in different nodes, other potential problems arise: either all the listing is transferred at once (probably causing long response times due to having to wait for all data to arrive) or the listing is fragmented (facing complex error situations in case the communication fails at the middle of the transfer, leaving the locks acquired).

An additional problem is posed by file system semantics as POSIX, which defines an interface allowing to list a directory fragment by fragment (entry by entry, specifically) and permits a user starting the listing, leaving it unfinished, and continuing with the listing after an arbitrary amount of time. Furthermore, POSIX requires the ability to replay a partial directory listing keeping the previous listing order (via `telldir/seekdir` interface).

The traditional and well-known approach in the field of file systems to handle large directories is to maintain the entries of a directory in an ordered tree structure, usually stored as special file. By having the entries ordered, it is relatively simple to keep a pointer to the last entry listed and generate the listing fragment by fragment. If a new entry is added before the current pointer, it will be ignored; if an already listed

entry is removed and then re-created, it will be placed before the current pointer, so it will not be listed twice; and, finally if an entry is either created or removed after the current pointer, it will not affect the part of the listing already generated.

Nevertheless, the trees also have issues, especially when growing to large scales: in order to be efficient, trees should be kept balanced when new entries are added and when existing entries are removed. Balancing a tree while keeping it ordered can be a costly operation, which in the case of file systems is aggravated by the fact that, in order to take advantage of block devices, the leaves of the trees are not entries, but groups of entries packaged in one or more consecutive blocks. As such groups can keep only a limited amount of entries, adding or removing an entry may involve splitting or uniting blocks of entries, causing non-trivial (and potentially time consuming) reorganizations of data. Furthermore, in current distributed environments, maintaining an ordered tree which could be distributed across several nodes for scalability purposes even adds more complexity and costs, making it a poorly scalable solution.

An alternative solution would be storing the entries in a table in a database. Current database technologies allow large-scale distribution of the data (so that the mechanism is scalable and can operate on distributed systems easily, contrary to what happened with ordered trees), and storage and retrieval of data (entries, in our case) is fast and effective. The issue with database tables is that, in general, they do not guarantee a particular ordering of the data contained in them. So, the obvious way to generate listings from them would be locking the table or using transactions (with the cost and potential problems commented above).

In order to address such issues, we devised a method that allows to store entries in non-explicitly ordered repositories (such as database tables), while allowing the implementation of entry listings (even fulfilling POSIX semantics). The mechanism is based on the fact that all entries have some distinctive characteristic that allows to differentiate them (for example, all entries in a directory must have different names), so that such characteristic can be used as input to a hash function, and so that the set of distinct results obtained from applying such function to the entries to be listed is smaller than the set of entries to be listed.

The procedure to list the directory entries starts by generating the list of distinct results of the selected hash function applied to the names of the entries to be listed. Each distinct hash value defines a disjoint subset of the entries to be listed (said subset containing the entries for which the hash function returns the same hash value).

The hash function must be chosen in such a way that the list of distinct hash values has an adequate size to be transferred in a single shot from a remote server having the set of entries to be listed to the local client generating the list for the requester. Additionally, each subset of entries defined by each hash value should also have a size adequate for being transferred in a single shot with a reasonable cost, and sorted in the local client to generate an ordered listing (note that a relatively large number of hash values with relatively few bits can be packed in a small space, and be used to define a large number of subsets, which can be potentially small).

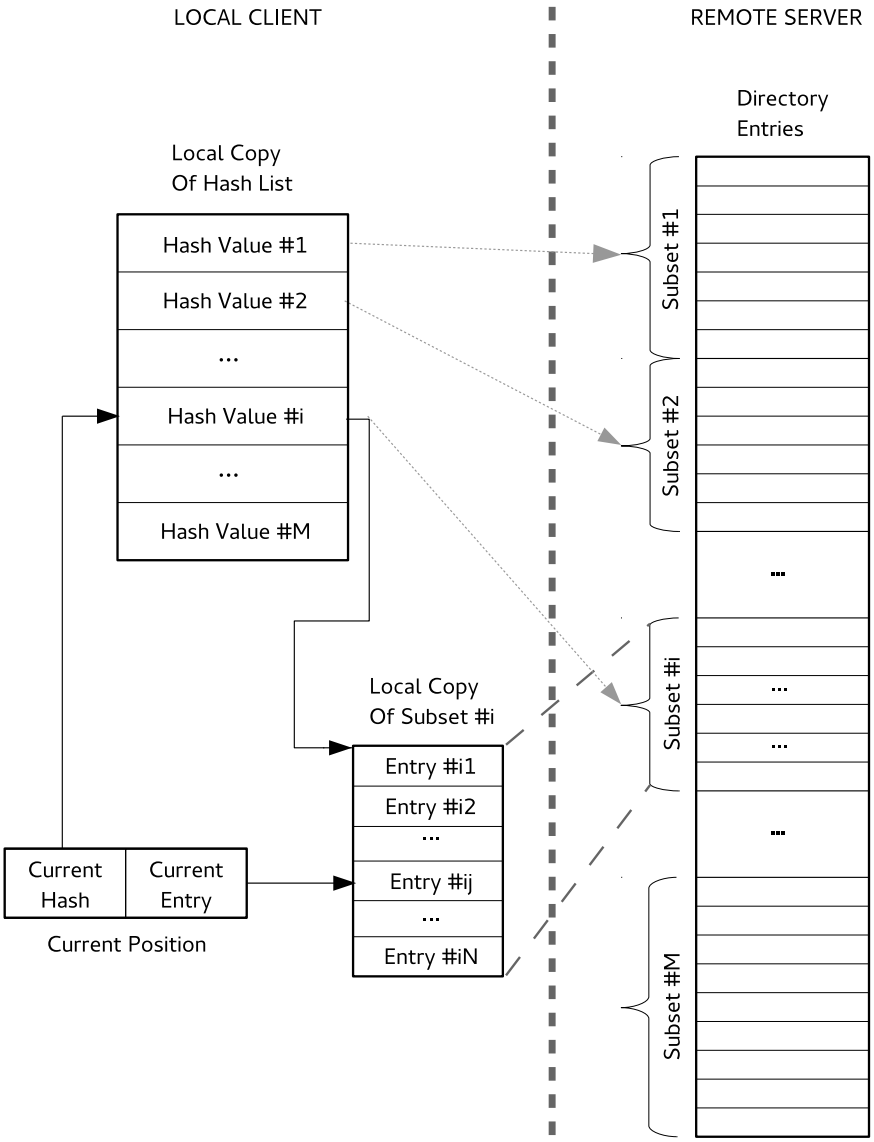

Figure 3.5: Hash-based directory partitioning schema for stateless listing.

Once the set of distinct hash values for all directory entries is received, the local client will sort said hash values following any arbitrary sorting criteria. Once the hash values are sorted, it will contact the remote server to request the subset of entries corresponding to the first hash value; when the subset of entries is received, it is also ordered according to an arbitrary criterion and can be forwarded to the requester as part of the whole listing (entry by entry, if so demanded). The same procedure is then repeated for each distinct hash value defining a subset of entries until all subsets of entries have been obtained and, thus, the list of all entries has been generated.

Figure 3.5 illustrates a situation where a hash function distributes the entries in a directory into `M` sets. Considering `i` as the current subset being processed, entries

added or removed having hash values identifying previous subsets will be ignored (in particular, if an entry from a previous subset is removed and a new entry with the same name is added, it will fall within the same subset — already processed — so it will not appear twice). On the other hand, entries added or removed to the current set do not affect the listing (as they are added into or removed from the remote repository, while the client generating the listing is working with its own local copy). Finally, entries added or removed from unprocessed subsets will appear or disappear, respectively, from the final listing (but they will not produce duplicated results).

There is a possibility that a new entry is added and its corresponding hash value is not in the initial list of distinct hash values cached by a client; in this case, such entry will not appear in the final listing (which is also compatible with most of the file systems semantics — and particularly with POSIX).

Re-playing the entry listing in the same order from an arbitrary point is possible as far as the original list of distinct hash values has been kept locally (or recovered and re-organized into the original order). A handle to indicate a given position in the listing to start the replay in an efficient way would consist of the name of the desired entry and the hash value corresponding to it. Such handle could be generated and stored locally whenever a `telldir` operation is performed.

In summary, the mechanism described above improves the system robustness because the user can request listings of large amounts of entries without jeopardizing the *Metadata Virtualization Layer* efficiency and reliability. By using the above-described method, large directory listings can be generated in a reproducible way, fragment by fragment, without the need of locking large amounts of data during arbitrary amounts of time. Additionally, the mechanism does not require any state in the remote server storing the entries (e.g. to track the last entry returned), thus avoiding potential causes of errors in distributed environments. Finally, the repository used to keep the entries to be listed does not need to have any predefined ordering semantics (in particular, tables in arbitrary databases can be used, enabling the system to take advantage of the database technology, including scalability and distributed operation).

In order to avoid the cost of computing hash values for the entries to be listed, it is possible to calculate the hash value associated to given entry when the entry is created, and store the value of the hash value together with the entry.

The *Name Space Manager* can use a variable number of bits of the hash function in order to adjust the number of fragments of the directory. This allows to adapt the algorithm to the number of entries: large directories will use more bits and, therefore, will be divided into more subsets, while small directories can use fewer bits and be divided into a smaller number of fragments. The goal is to keep the number of entries in each fragment reasonable (not too small, to reduce the overhead of sending each fragment, and not too large to be handled effectively).

Additionally, there may be other factors that may influence the choice of a given hash function, so that the resulting subsets are smaller or larger. For example, if the component generating the listing (e.g. the user client) is running on a low capacity

hardware (e.g. a mobile device), it may be interesting to have smaller subsets, so that they can be adequately handled by such a device; on the contrary, in an environment where the components generating the listing and storing the entries have high capacity and are connected through a robust network with large bandwidth, then it may be useful to generate larger subsets to reduce the overhead of multiple requests.

3.5.8 Coordinating distributed file creation

File creation in a distributed environment is a delicate operation that requires careful coordination between the different components of the *Metadata Virtualization Layer*. On one side, the *Metadata Manager* will have to check the access permissions and create a new entry in a particular view. On the other side, an actual underlying file must be created somewhere in the underlying file system. Eventually, the new entry and the new underlying object will be linked by attaching the path of the underlying object to the virtual object. Last but not least, the semantics of the file system operation may require to generate a handle to the new file, so that further requests on it (e.g. `reads` and `writes`) can be tracked and associated to the corresponding file (and to any particular flags indicated during the creation of such a file) without needing to repeat searches of the path in the name space.

The first step to fulfil a file creation request is to generate an identifier for the new object (the *inum*). This identifier must be unique across a distributed system where users may issue simultaneous requests to create files, and where the components of the *Metadata Virtualization Layer* can also be distributed across multiple nodes.

In our implementations of the metadata virtualization framework, the local client generates the identification numbers (*inums*) by requesting a range of identifiers to a global server, which gets requests from all participating clients and guarantees that they receive different sets of identifiers; then, the client consumes the set of identifiers as it needs. The identifier service uses a monotonically increasing counter to generate the identifier sets, and it has the particularity that assigned identifiers are never reused, even if the corresponding object has been deleted.

A file creation request also requires generating and identifying an actual underlying object. If the underlying storage is a file system, then the object is usually identified by a path for such file system. In our implementation, the actual path is determined by the local *Data Manager*, which has to make sure that it does not collide with paths generated from remote *Data Managers* in other clients.

Once the path is determined, the next step consists in requesting the underlying file system to create the corresponding underlying object. If the underlying object creation is successful, the *Data Manager* will have a valid reference (usually a path) to an underlying object that can be associated to the corresponding virtual object as an attribute. This association is carried out by the *Name Space Manager*, which creates the corresponding entry in the specified view.

In case the *Name Space Manager* cannot create the entry (for example, because an entry in the desired name space already existed with the same name), a notification

is sent back to the *Data Manager*, so that the created underlying file can be removed. Additionally, the discarded identifiers (e.g. the *inum* for the new virtual object) can be stored locally to be re-used later (as it was not associated to any entry, the identifier was not officially assigned and, therefore, the convenient rule about not re-using assigned identifiers is still maintained).

The procedure just described can be used not only for files, but also for other types of file system objects (in particular, those that can be created with a `mknod` system call in a POSIX-based file system), such as named pipes, *Unix* domain sockets, or character and block devices. Directories and symbolic links, on the contrary, do not require the existence of corresponding underlying objects (all necessary information is kept by the *Metadata Virtualization Layer* and is not related to any underlying file system).

Apart from simply creating the file system object, some file systems may have more elaborated semantics. In particular, some of them may allow a conditional behaviour: preparing a file for access if it exists, and creating it if it does not exist and then preparing it for access. For instance, this is the case of the POSIX `open` call.

Such a combined operation might produce a race situation in a *Metadata Virtualization Layer* running on top of a distributed system; therefore, the procedure is adapted to avoid inconsistencies. In the case that at least two nodes are trying to create and prepare a virtual object for a non-existing file, the initial procedure is similar to the simple create-only method: each component generates a new virtual object with its corresponding underlying object and, then, tries to register it in a name space via the *Name Space Manager*. At least one of them should succeed, but the other (or others, if more than two nodes were trying to create files), instead of just getting the error, will get the path corresponding to the file that was successfully created and registered; then, they will discard its own virtual object and the corresponding underlying file and proceed to prepare the access to the registered one (which involves interaction with the *Metadata Manager* in order to make the necessary checks on the registered file system object type and permissions — as it could happen that different types of file system objects were intended to be created with the same entry name: for instance, a file and a directory).

When the local components of the *Metadata Virtualization Layer* receive someone else's file path to open during an `open/create` request (instead of the path generated by themselves), there is a chance that the actual file in the underlying repository is removed before it can be effectively opened. This situation can be handled in different ways. For example, an "open-in-progress" counter can be maintained for the data structures related to the file and, despite being removed from the name space, the removal of the virtual object and the corresponding underlying file could be delayed until such counter reaches zero.

In the implementations of the framework presented in this thesis, we opted for a simplified alternative that just discards the information and re-tries to create its own file again (theoretically, in a highly volatile environment, the described failure situation could be repeated indefinitely, leading to starvation; to avoid that, a retry counter is used to return an error after a certain number of retries).

3.5.9 Kernel vs. user-space implementations

Considering the operating system boundary, the components of the *Metadata Virtualization Layer* can be placed at either kernel level or at the user-space. Both approaches have their advantages and their drawbacks. Kernel-based implementation offers, a-priori, the advantage of a less expensive interface (no user-kernel cross-boundary calls); it may provide access to lower-level information, allowing for educated decisions; and may also permit hooks to low-level operations, permitting aggressive optimizations. Nevertheless, kernels tend to contain non-standard interfaces that are often subject to unannounced changes; integrating code in such environment is complex and error prone, and may jeopardize the stability of the whole system; and last but not least, low-level hooks can make the system too dependent on specific components and compromise the desired flexibility of the system.

Oppositely, building the service in user-space is more appealing. Of course, the low-level virtualization techniques allowing the interception the file system calls may require some support mechanisms at the kernel level; but these can be kept to a minimum. Running the bulk of the service operation at user-space has several advantages. First, it would make the development easier and would allow the introduction of standard failure isolation techniques. Second, the technology can be more easily applied to different platforms; by reducing the dependencies on kernel particularities, most of the code can be reused and porting it to a new platform would be easier. Finally, user-space may enable the use of standard interfaces and the possibility of offering a convenient software development kit for writing extensions or third-party applications based on our system (increasing its added value).

Our experiments (see Chapter 4) demonstrated that the possible performance drawbacks caused by cross-boundary calls were not substantial. Therefore, most of the developments in this thesis use user-space implementations whenever possible (relying on kernel support only for request interception and some low-level features).

3.6 Summary

In this chapter, we have presented the elements needed to support a storage system (namely a file system) that is flexible enough to provide multiple simultaneous views of the file organization adapted to specific application needs. We have also proposed an architecture for metadata virtualization framework that can be implemented as a layer running on top of conventional file systems, seamlessly integrating them.

The decoupling between the metadata management in the framework and the underlying file systems made us consider a series of practical issues that needed to be resolved to provide a consistent file system semantics. In Section 3.5, we discussed possible solutions to such issues, determining the most adequate ones for our goals.

The following chapters will present specialized implementations of the framework discussed here, adapted to specific environments.

CHAPTER 4

Virtualized Metadata Management for Large-Scale File Systems

Contents

4.1 Introduction

4.1.1 Motivation

High-performance computing is rapidly developing into large aggregations of computing elements in the form of big clusters. In the last few years, the size of such distributed systems has increased from tens of nodes to thousands of nodes, and the number is still rising. Trying to keep pace with this evolution, parallel file systems have emerged, providing mechanisms for distributing data across a range of storage servers and making them readily available to the computing elements.

At the other end, the final user's view of the storage systems has not changed significantly: for the usual case, files are organized in a hierarchical name space, much in the same way as they were placed in a classical local file system using an attached disk in a single computer.

In this classical view, a directory was tacitly used as a hint to indicate locality. Indeed, it is common to find directories containing files that are going to be used together, or in a very related way (for example, a directory containing a program's code, the corresponding executable, configuration files and the output of its execution).

Trying to exploit this affinity, file systems tended to group together the management information (the metadata) about directory contents. This was favoured by the fact that this information is relatively small, so that it is feasible to pack together information about access permissions, statistics, and also the physical location of the data, not only for a single file, but also for a set of related files (i.e. for files present in the same directory).

The approach of treating directory contents as a group of related objects with similar properties is still present in modern file systems (with the logical adaptations due to the evolution of the technology). It is common to see parallel file systems to use directories as *mount points* to access different volumes or partitions with different functionalities or, in finer grain file systems, to be able to specify directory-wide rules that apply to directory contents.

The problem appears when, in large clusters with parallel file systems, directory contents are semantically related according user's view, without that meaning that they are going to be used together or in similar ways from the file system perspective. Reusing the example above, now the program code in a directory will be compiled and linked in a single node, to generate a binary that will be read and executed simultaneously in 2,000 nodes, each of them generating an output file to be left in the same directory, which will be collected and read by a single post-processing tool generating some summary information. Files are indeed related, but patterns of use are totally dissimilar.

File systems trying to keep close the apparently related fragments of metadata will end up trying to keep consistent a relatively small pack of miscellaneous information (not easy to distribute and share) while it is being simultaneously used, and probably modified, by a large number of nodes. As a result, the pressure on metadata handling in large scale systems will rise, producing delays comparable to the actual data transfer times to the storage systems, and jeopardizing the overall system performance.

The *Metadata Virtualization Layer* proposed in this thesis can help in this situation by breaking the ties between the user-visible hierarchical organization (the directory tree as seen by the application) and the underlying file system structure, not only by decoupling it from the storage location of data blocks, but also by transparently re-structuring the actual directory and *i-node* information, so that the pressure on metadata handling is reduced and critical bottlenecks can be avoided.

A specific implementation of the *Metadata Virtualization Layer* was developed to demonstrate the feasibility of adding a new layer above raw file systems without harming the performance. This prototype is called *COFS* (COmposite File System). *COFS* is a framework to decouple the file system metadata and the name space hierarchy from the actual data handling in the underlying file system.

One of the specific motivations of this development was the file system performance drop observed on one of the BSC large production clusters, MareNostrum when using *IBM GPFS* as parallel file system.

A prototype of *COFS* was developed and tested on a small cluster (Nord), where it boosted the performance of its native file system by transparently re-arranging the user file hierarchy without altering the user view. The prototype was eventually deployed on MareNostrum, where it was tested under production conditions. This prototype was able to boost the performance of the file system metadata at MareNostrum, specially in scenarios where shared directories are accessed from multiple nodes simultaneously (the detailed results are shown in Subsection 4.4.2).

COFS was also deployed in the CEA INTI test-bed for the PRACE-2IP project, in order to test the suitability of the *Metadata Virtualization Layer* features for the *Lustre* parallel file system. In this case, *COFS* was able to improve the performance of `create` and `open` operations on shared directories, as shown in Subsection 4.4.3.

4.1.2 Approach

A usual way to deal with non-optimized cases is to add even more specific optimizations, either by means of file system specific modifications (for example, the parallel creation policies in *PVFSv2* [Devulapalli 2007]), or by means of a middleware targeted to fulfil the needs of specific classes of applications (e.g. MPI-IO implementations using hierarchical striping for *Lustre* [Yu 2007]). Unfortunately, this approach does not deal with performance penalties caused by unforeseen or inadequate access patterns from arbitrary applications, which are likely to occur in heterogeneous workloads.

We have decided to take a novel approach to mitigate the effects of non-optimized situations. Our technique to handle the metadata issues consists in placing a *Metadata Virtualization Layer* on top of the file system for decoupling the user view from the actual low-level file system organization, allowing us to convert the application access patterns into something that can be dealt with by the underlying file system without harming the performance, instead of trying to adapt the file system to any possible workload. In other words, the mechanism consists in improving the file system behaviour by avoiding bottlenecks caused by application patterns, instead of focusing only on optimizing the file system for a very specific workload.

Running on top of the file system (instead of integrating new code into it) makes development much easier and simplifies the integration of other state-of-the-art technologies which may help to address the issues under study. Additionally, it allows us to adapt the same solutions to multiple file systems and environments easily.

4.2 Case study

As mentioned in Section 4.1, one of the motivations of our work was the performance drop observed at BSC clusters; they had *IBM GPFS* file systems and an heteroge-

neous workload corresponding to different projects, comprising both large parallel applications spanning across many nodes, and large amounts of relatively small jobs.

Large parallel applications usually create per-node auxiliary files and/or generate checkpoints by having each participating node dumping its relevant data into a different file; not unlikely, applications place these files in a common directory.

On the other hand, smaller applications are typically launched in large bunches, and users configure them to write the different output files in a shared directory, creating something similar to a file-based results database; the overall access pattern is similar to that from a parallel application: lots of files are being created in parallel from a large number of nodes in a single shared directory.

So, observation indicates that typical modus operandi ends up creating large amounts of files in the same directory; and very large directories, specially when populated in parallel, require *IBM GPFS* to use a complex and costly locking mechanism to guarantee the consistency [Schmuck 2002], resulting in far-from-optimal performance. For example, the operations team at BSC noticed that a parallel application spanning across a large number of nodes can use an important portion of its execution time creating and writing checkpoint files (significantly greater than what should be expected for the simple transfer of data) [Bent 2009] [Frings 2009]. To mention another example, trace collection software used by computer science researchers to obtain per-node application execution traces for performance analysis also suffer from this inadequate metadata handling [Lab 2007].

As an additional concern, the overhead is not limited to the infringing applications, but affects the whole system because file servers are busy with synchronization and all file system requests are delayed.

Our proposed use of *Metadata Virtualization Layer* to decouple the name space from the actual low-level file system layout can mitigate the issues, offering, for example, large directory views to the user while internally splitting them to reduce the synchronization pressure on the *IBM GPFS*. This leads to overall performance improvements while avoiding the need to change how the individual applications work (which is difficult to do for some legacy codes).

To verify our assumptions and evaluate their impact, we conducted a series of tests; first, under a controlled environment in a small cluster and then, in the MareNostrum supercomputer itself. The rest of this section describes the execution environments and show the metadata behaviour on a bare *IBM GPFS* file system.

4.2.1 Execution environments

The first controlled tests to determine the *IBM GPFS* metadata behaviour were conducted at a test-bed cluster at BSC called Nord. The results obtained at Nord were afterwards validated at a production-grade system: the MareNostrum supercomputer.

The Nord test environment consisted in a cluster of 70 IBM JS20 *blades*, with 2 processors each (PPC970FX) and 4 GiB of RAM. Blades were distributed in several

blade centres: each blade center had an internal 1 Gbps switch, and the different blade centres were interconnected through non-uniform network links (so that available bandwidth was not the same for all blades). We conducted our tests on a *IBM GPFS* file system, based on two external Intel-based servers connected to the cluster by 1 Gbps link each.

The MareNostrum setup at the time of the experiments consisted in a cluster of 2,500 IBM JS21 blades, with 2 dual core processors PPC970MP at 2.3 GHz and 8 GiB of RAM per blade. The interconnection network is a 1 Gbps Ethernet, and the tested file system was *IBM GPFS* version 3.2.1, served by 14 combined data/metadata servers.

IBM GPFS offers a standard POSIX interface, but it also has the possibility to use non-POSIX advanced features for increased performance (e.g. for MPI-IO). A typical configuration consists of clients which access to a set of file servers connected to the storage devices via a SAN. Metadata is also distributed and consistency is guaranteed by distributed locking, with the possibility of delegating control to a particular client to increase performance in case of exclusive access [Schmuck 2002].

The situation we wanted to evaluate essentially involved parallel metadata operations, so we used *Metarates* [Met 2004] as the main benchmark. *Metarates* was developed by UCAR and the NCAR Scientific Computing Division, and measures the rate at which metadata transactions can be performed on a file system. It also measures aggregate transaction rates when multiple processes read or write metadata concurrently. We used this application to exercise `create`, `stat` and `utime` on a number of files in the same directory in parallel. As an addition to the original code, we also measure the time needed to `open` and `close` a file. The cost of `create` is measured by having each participating process create the same number of files in parallel in either a shared directory or in dedicated directories and, then, dividing the elapsed time into the number of created files. For the rest of metadata operations, files are initially created sequentially from a single process (in either a shared directory or in dedicated directories for each participating process) and, then, each participating process performs in parallel the tested operation on a disjoint subset of files; again, the cost of the operation is computed by dividing the elapsed time (without the creation phase) into the total number of files.

It is important to mention that neither our test-bed cluster Nord nor the MareNostrum were exclusively dedicated to benchmarking: our measurements were done while the rest of the cluster was in production (specifically, we had an allocation of 16 nodes out of 70 for most of the tests in Nord, and up to 256 nodes in MareNostrum). This was an interesting situation because it gave us the opportunity to see how the unrelated (though usual) workload affected the file system behaviour. Though we observed that the overall effect of the production workload was quite homogeneous along time, special care was taken to detect and avoid deviations due to particular situations. The methodology employed to run the experiments tried to minimize the impact of punctual variations of system behaviour to get acceptable statistical confidence levels. For example, at MareNostrum, we usually repeated each measurement between 30 and 50 times depending on the level of 'noise' caused by activity

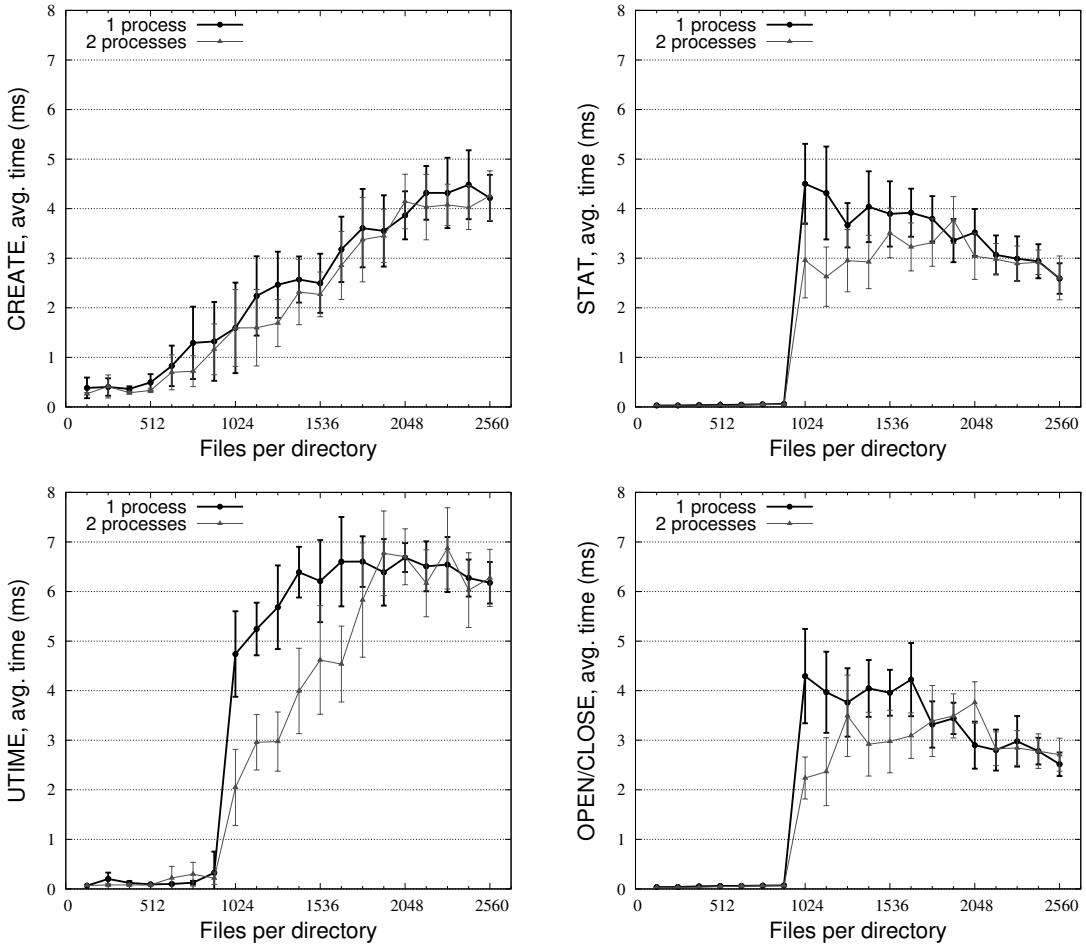

Figure 4.1: Effect of the number of entries in a directory in *IBM GPFS* at Nord, using 1 and 2 processes in a single node.

of the rest of the system at each moment; in all cases, we obtained a minimum of 16 measurements to calculate average values.

4.2.2 Base system behaviour at Nord

Understanding under which conditions a file system has a good performance, and which are the factors that negatively affect it, is the starting point for boosting it.

Figure 4.1 shows average times for *IBM GPFS* metadata operations in a single node (using 1 and 2 processes in the same node). The average operation time used here and in the other diagrams is calculated by dividing the elapsed execution time into the number of operations performed in a benchmark run (which, in the case of *Metarates*, corresponds to the total number of files); the error bars are used to represent the standard deviation, in order to give an indication of the variability of the results.

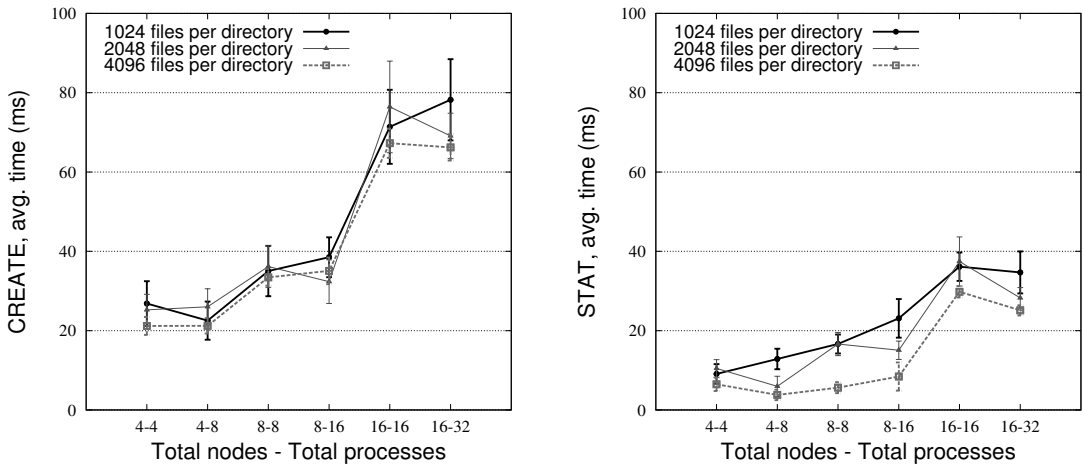

Figure 4.2: Base parallel metadata behaviour of *IBM GPFS* at Nord, using 1 and 2 processes per node.

The figures reveal an extremely good behaviour for `stat`, `utime` and `open/close` when the directory size is below 1,024 entries (with a performance comparable to local file system rates). The low operation times are related to the ability of *IBM GPFS* to delegate control to clients under certain circumstances (e.g. single-node access and data present on local cache [Schmuck 2002]) Beyond 1,024 entries, performance drops to network-compatible rates. Having two processes seems to compensate this effect slightly, as some fetched data can be re-used by the second process (though this effect disappears beyond 2,048 entries).

The pattern is different for `create` operations: there is no cached information to exploit (as we are creating new entries) and operation time follows a steady increase above 512 entries.

Figure 4.2 shows results for `create` and `stat` operations (the x axis shows the combination of nodes and processes used, and the error bars represent the standard deviation, indicating the level of variability of the measures).

The first observation is the large increase of the `create` time when operating in parallel. For a directory with 1,024 entries, it goes from slightly less than 2 ms in a single node (Figure 4.1) to more than 20 ms in 4 nodes and more than 30 ms for 8 nodes (Figure 4.2). The `stat` operation also suffers from a noticeable slowdown when executed in parallel.

Considering that these operations have a very small payload (only metadata information), we can deduce that an important portion of the relatively large operation times is consumed, not by actual information being transmitted from the server to the clients, but by consistency-related traffic (even when each process works on a different set of files). The reason for this assumption is that the amount of information actually transmitted is small (e.g. the set of attributes for each file needs less than 100 bytes),

and thus, the increase in time cannot be justified by network bandwidth limitations; additionally, specific measurements transmitting the same amount of data generated by a null test driver instead of *IBM GPFS* showed an almost negligible cost in the same hardware (see the *COFS* null driver results in Figure 4.11). That assumption would give our virtualization system enough margin to obtain a good speed-up by reorganizing the file layout and reduce the synchronization needed among *IBM GPFS* clients.

The performance drop when using 16 nodes (clearly visible for the `create` — Figure 4.2) is due to the test-bed network topology: when using up to 8 nodes, blades are located in close blade centres; for 16 nodes and above, some blades are allocated in far locations, so that the available bandwidth is shared with more nodes and the effect of possible interferences from the production workload is more noticeable. Interestingly, as the number of nodes involved increases in this environment, the potential benefits of reducing the synchronization cost by using a more appropriate file layout are also augmented.

Another interesting observation is how the `stat` time increases as the number of files in the directory decreases. Being *IBM GPFS* a black box, we can speculate that this is related to distributed locking granularity: a `stat` response from the servers contains information about several entries to take advantage of bandwidth; however, the fewer files we have, the higher is the probability of locking conflict between a larger number of nodes. Note that conflict-free access is not possible, because 'the user' (here, the *Metarates* benchmark) has decided to put all the files in the same directory and that forces the file system to handle a shared common structure (the directory).

The behaviour of the system for 2,048 files per directory deserves a special comment, as the performance varies depending on the number of processes per node. Apparently, *IBM GPFS* is able to coalesce the requests inside a client and this coalescing crosses a boundary that makes possible for *IBM GPFS* to take advantage of the situation and improve the performance (reducing the inter-node conflicts, and being able to handle intra-node sharing efficiently). Ideally, a user could use this knowledge and tune an application to take advantage of it; however, parameters are closely related to specific data center configuration, so that user-specified per-application tuning would be impractical. On the contrary, a *Metadata Virtualization Layer* as the one provided by *COFS* may have a module with this kind of information and exploit it transparently.

The experiments at Nord revealed both strong and weak points in the *IBM GPFS* behaviour. In the next subsection, we validate the information obtained from the small test-bed in a larger, production-grade system.

4.2.3 Base system behaviour at MareNostrum

As mentioned before, one of the motivations of this work was the performance drop observed in large production clusters using *IBM GPFS*, which were related to metadata management issues caused by the way in which the applications used the file system.

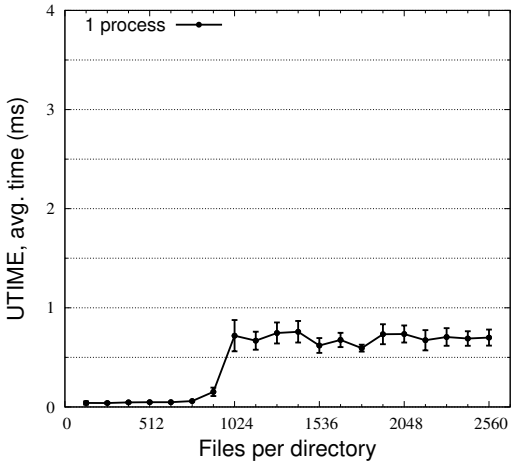

Figure 4.3: Variation of `utime` operation cost in *IBM GPFS* in a single node at MareNostrum with respect to the number of entries, using 1 process.

Therefore, it is important to validate the data obtained in the small test-bed Nord in a larger cluster such as MareNostrum.

We have conducted a series of experiments in the MareNostrum *IBM GPFS*-based cluster to confirm that metadata handling was actually a significant cause of performance drops. The situation we wanted to evaluate essentially involved parallel metadata operations, so we used *Metarates* as the main benchmark (as we did in the Nord test-bed). We will focus on presenting the results obtained for `create` and `utime`; the results obtained for the other measured operations (`stat` and `open`) closely follow the same pattern as `utime` and we will mention them only in case of substantial differences.

Following the results obtained at Nord, the observations confirmed that a substantial amount of time was consumed, not by actual information being transmitted from the server to the clients, but by consistency-related traffic (even when each process works on a different set of files). Nevertheless, the production-grade hardware at MareNostrum and the better network setup compared to Nord made us expect differences in the performance that could affect the effectiveness of our *Metadata Virtualization Layer*.

Figure 4.3 shows the average time to fulfil an `utime` request (changing the timestamp indicating when the file was last modified) on a file in a directory accessed by a single client node. The low values for small directories (about 45 microseconds) are the cost of the request when the *IBM GPFS* delegation mechanism is active (control is transferred to the client node, so that operations can be carried on locally, without server synchronization) and represent a lower boundary for the operation cost; beyond 1,024 entries per directory, delegation is no longer active and the operation requires remote server intervention, resulting in larger operation times.

The operation average times for simultaneous `utime` calls from different nodes, using 1,024 files per node, are shown in Figure 4.4. Ideally, the cost should be around 0.75 ms (the cost in a single node when a round-trip to the server is done). What

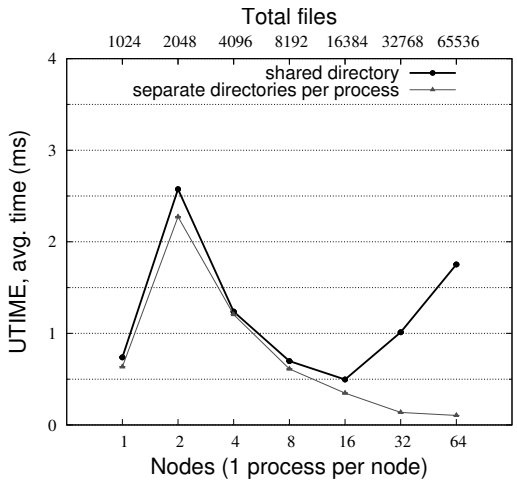

Figure 4.4: Comparison of parallel `utime` cost in *IBM GPFS* at MareNostrum on shared and non-shared directories, using 1 process and 1,024 files per node.

we actually observe is that the cost with 2 nodes is much higher, and progressively decreases when we increase the parallelism.

When all clients have their files in a shared directory, the optimum value is reached for 16 nodes; beyond that point, the system does not scale any more and increasing the number of nodes just increases the access conflicts, reducing the performance. It is worth mentioning that the performance reduction is not observed when each node works in its own directory; this fact seems to indicate that shared access management is one of the causes of the loss of performance.

Another interesting observation from *IBM GPFS* is that the average parallel `create` time also differs depending on the files being created on a single shared directory or on unique directories per processor. The difference in performance is much greater than in the `utime`, as can be observed in Figure 4.5 for the creation of 1,024 files per node.

It is important to mention that each node creates a disjoint set of files; so, the only cause of poor behaviour is the management of a shared metadata management structure: the directory. By changing the way the metadata is organized and handled, it should be possible to reduce this cost to levels similar to using unique directories per processor.

In summary, the experiences and observations in the MareNostrum supercomputer confirmed the trends observed in Nord regarding the impact of shared metadata management on the performance of parallel file systems.

4.2.4 Lessons learned

We learned some facts from the experiments:

- The best sequential behaviour in *IBM GPFS* is obtained for small directories (below 1,024 entries for `stat`, `utime` and `open/close` and about 512 entries for

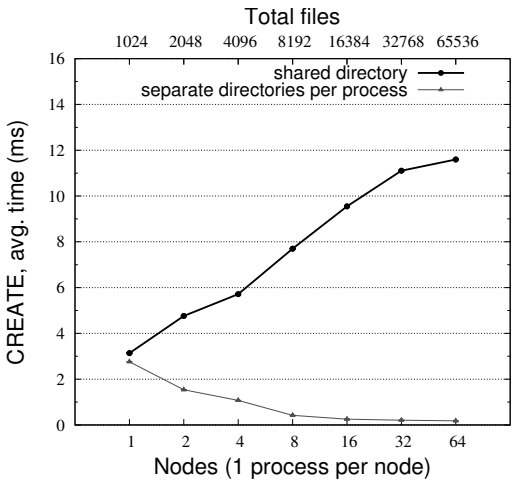

Figure 4.5: Comparison of parallel `create` cost in *IBM GPFS* at MareNostrum on shared and non-shared directories, using 1 process and 1,024 files per node.

`create`). This is due to the effective optimizations that *IBM GPFS* has in place for working sets consisting of small directories accessed from a single node. The effect of optimizations disappears for large directories and/or parallel accesses.

- Parallel file creations on shared directories suffer an important overhead. This overhead appears even for a small number of nodes. On the contrary, parallel file creation on separate directories scales reasonably well.

- Parallel non-`create` operations on shared directories are also likely to produce locking conflicts between nodes, and substantial overhead appears at 32 nodes and beyond. As in the case of file creation, operations on separate directories show a better behaviour.

Decoupling the name space from the low-level file system layout should mitigate the issues. The *Metadata Virtualization Layer* proposed in this thesis could offer large shared directory views while internally splitting them into separate directories to reduce synchronization pressure on the *IBM GPFS* servers, and keep the use patterns within the optimized boundaries.

In particular, we observe that the differences in performance between the optimized and non-optimized behaviours is large enough to cover the potential cost of adding a virtualization layer on top of the file system and, still, leave room for performance improvements.

The COmposite File System (*COFS*) prototype is an implementation of the *Metadata Virtualization Layer* discussed in Chapter 3, and it was developed to address the metadata performance issues observed in *IBM GPFS*.

In particular, the measurements presented in this section inspired the *COFS Sparse Placement Module* for the *Placement Driver* described in Section 4.3. Its goal is splitting large shared user directories into non-shared directories small enough to enable delegation and local access, so that *IBM GPFS* can improve its performance.

4.3 Prototype implementation

The use of a *Metadata Virtualization Layer* to decouple the user view of the file system organization from the actual layout of the file system can alleviate contention issues in metadata handling and, therefore, boost the performance of a file system. Nevertheless, adding new layers to a traditional file system architecture may also raise concerns: the overhead introduced by the new layers could actually harm the performance and the new components could bring forth new interactions that increased the overall system complexity, nullifying the potential benefits.

The *COFS* prototype was developed as a proof-of-concept to cast out such concerns: it demonstrates that it is feasible to add a new layer above a raw file system to decouple the file system metadata and the name space hierarchy from the underlying data handling without harming the performance nor introducing complex dependencies.

COFS presents the user a virtual view of file system directories, allowing the user to organize the files as they please while implementing a completely different directory layout in an underlying file system. The performance results obtained will be detailed in Section 4.4.

This section describes the most relevant parts of the *COFS* prototype. *COFS* provides a file system interface supporting the semantics required to run most applications on top of it. The mechanisms used are also flexible enough to easily allow any required extension.

In particular, we have taken special care to fulfil two expectations: first, the prototype could not assume oversimplifications of a file system's behaviour (i.e. a minimum set of typical file system features should be fully operative in the prototype, including, for example, data integrity and access control); and, second, the implementation should not jeopardize the extensibility of the prototype by depending on specific low-level system characteristics.

A way to guarantee that *COFS* is complete enough to allow applications to run on top of it is making it POSIX-compliant. POSIX is still the dominant interface model for most applications using file system services. It imposes a quite restrictive semantics and consistency model. Supporting it, the prototype demonstrates its ability to handle strict operation paradigms. From that point, relaxing the model is always easier, and can be done by means of specific configuration options or, for example, by indicating the desired behaviour using the extended attributes mechanism on files or directories.

The current prototype fully supports POSIX except for some elements which are not relevant for storage management and were not implemented (namely named pipes). The prototype successfully passed the POSIX compliance test suite by Pawel Jakub Dawidek [Dawidek 2008], excluding the named pipe tests.

Despite the strict POSIX support, *COFS* also permits deactivating some of the restrictions via configuration files, allowing certain optimizations to take place. This kind of relaxation is common in HPC environments, allowing applications a trade-off between the expected file system support and the potential performance obtained.

Security is another important aspect that must be taken into account in order to have a complete file system (though it is often overlooked in experimental and test systems). The prototype relies on the underlying tools and infrastructure to guarantee the appropriate data protection. The *COFS* prototype uses FUSE and Erlang/OTP as foundation, which already provide mechanisms to implement an adequate level of security [Szeredi 2005] [Erl 2013]; in particular, FUSE can obtain authentication information directly from its kernel module, and Erlang provides a *cookie*-based and SSL support to prevent the inter-node communications from being tampered. Additionally, the underlying file system also provides its own mechanisms to guarantee data integrity and also a certain degree of access control. The prototype just leverages all these components to provide global security.

4.3.1 Design aspects

The *COFS* prototype is be able to decouple the user view of the file hierarchy, the metadata handling and the physical placement of the files, following the framework architecture discussed in Chapter 3.

Its specific goal is to present the user a virtual view of the file system directories, while the actual layout can be optimized for the underlying file system.

Additionally, *COFS* contains mechanisms allowing to track operations at the file system interface to obtain detailed traces of the file system activity. This information has facilitated the exploration of different mechanisms for metadata handling and the assessment of its impact on performance, and also to test different policies and parameters for laying out data in the underlying file system.

The first thing to note is that the *COFS* prototype was initially developed for a *Unix*-like operating system (specifically *Linux*) which provided the applications with a POSIX interface as a means of interacting with the file system. In practice, this involves providing support to deal with particular structures of metadata and a specific set of operations to manipulate the files. Nevertheless, one of the design directives of the prototype was that it should be easily adaptable to other file system interfaces and semantics, being able to manage different pieces of metadata information and respond to a different set of operations. In other words, the design has no hard dependencies on a specific operating system or a specific file system interface.

The prototype uses an actual file system as a back-end for the data. The interaction of the prototype's *Data Manager* with the underlying file system is done via basic POSIX calls, which means that the prototype has no dependency on a particular underlying file system, and can operate on top of any one providing such interface. Actually, *COFS* has run without modifications on top of *Ext3*, *IBM GPFS* and *Lustre*, among other file systems.

As only a few simple operations are actually required (create and remove files, and read or write data at specific offsets), the design of the *Data Manager* should be easy to adapt to other file system interfaces, such as the native *Microsoft Windows* file system interface; in fact, the implementations presented in Chapter 5 are able to use

both *Microsoft NTFS* and *SMB/CIFS shares* as data back-ends. Similarly, it would not be hard to port the *Data Manager* to work directly on top of other storage media.

Another important point to mention is that the prototype is designed to operate in both local and distributed environments. In particular, the component corresponding to the *Metadata Manager* can be detached and run either locally or in a remote node. Moreover, it is able to provide the metadata and name space services to several clients simultaneously and coordinate them.

Moving to functional aspects, the prototype hides the underlying file system directory hierarchy and metadata management and provides its own name space and high-level metadata to the user. This approach completely detaches the user view from the features offered by the underlying file system, preventing potential compatibility issues with the back-end.

Specifically, the high-level metadata provided by *COFS* includes access control (owner, group and related access permissions), symbolic and hard link management, directory management (both the hierarchy and the individual entries) and size and time data for non-regular files (management of sizes and access times for regular files relies on the underlying file system).

The metadata handled by the prototype does not include any knowledge of low-level data storage (in particular, there are no references about disks, blocks or other storage objects — though they could be easily included if necessary). Currently, it is up to the underlying file system to take decisions about low-level data server selection, striping or block/object placement.

Whenever the actual data has to be accessed, the prototype forwards the requests to the underlying file system, indicating the actual path where the data is to be located (which, of course, may be different from the path seen by the user).

Another important design decision is that *COFS* is implemented as a user-space FUSE daemon which, in practice, is the factor that makes it independent of the underlying file system. The reasons for this are two-fold. First, even as one of the original motivation of this work was mitigating a performance drop on a specific file system (*IBM GPFS*), we believed that equivalent issues affect other file systems; so, the solving mechanism must be generic enough to be applied to any file system. Second, we planned to deploy our framework in production-grade clusters, and having a user-space drop-in package without hard requirements on kernel modifications or configurations makes it much easier to have access to such environments, as the potential impact on the rest of the system is minimum.

4.3.2 Architecture

Figure 2.2 in Section 2.3 showed an example of distributed file system environment. Applications running on the nodes accessed the the distributed file systems via local interfaces; the operations were received by local file system client components which, then, were responsible to contact the adequate file system servers through the network.

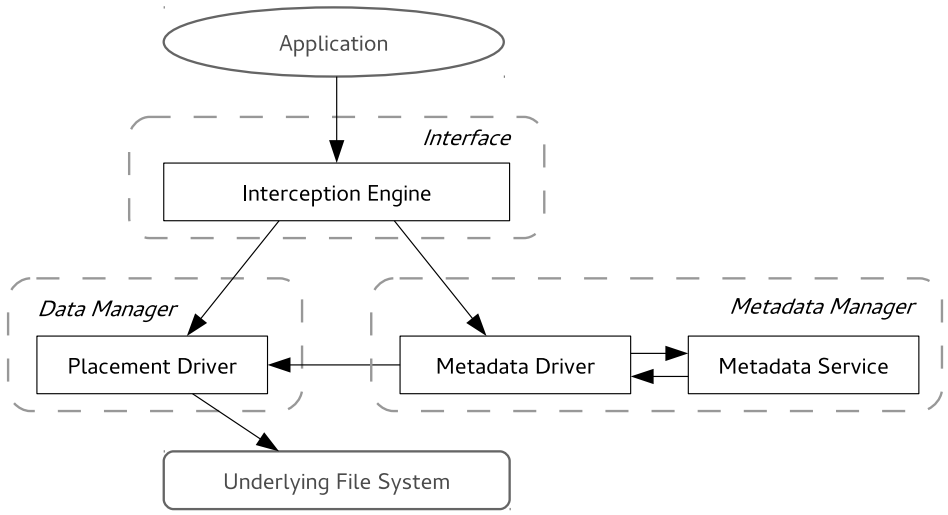

Figure 4.6: *COFS* components.

The *IBM GPFS* architecture follows the same model: client components are integrated in each node and provide a conventional file system interface to the applications. Additionally, these local components also take advantage of local node facilities such as the local page cache in order to improve performance.

Our *Metadata Virtualization Layer* prototype (*COFS*) has to be integrated into this architecture in order to provide a file system view to the applications detached from the actual system configuration. To this end, *COFS* introduces an extra layer on each client node providing a virtual view of the file system layout. This layer takes care of intercepting the applications requests, managing the name space and metadata, and communicating with the native file system for the actual data storage.

Figure 4.6 shows the actual components of the *COFS* layer and how they are mapped into the architecture discussed in Section 3.3. The *Interception Engine* provides a conventional file systems interface and captures the application requests; a *Placement Driver* fulfils the functionalities of the *Data Manager* discussed in Section 3.3 and handles the interactions with the underlying file system; finally, the functions of the *Name Space Manager* and the *Attributes Manager* from Section 3.3 are covered by the *Metadata Driver*, which can be assisted by a remote *Metadata Service*.

More specifically, the *Placement Driver* is responsible for mapping the regular files into the underlying file systems and actually accessing the data, while the *Metadata Driver* takes care of hard and symbolic links, directories, and generic attributes. Some operations need the collaboration of both drivers: for example, creating a file involves creating an actual regular file on a convenient location (a *Placement Driver* responsibility) and updating the proper entries in the directory (done by the *Metadata Driver*). So, an interface is also defined for communication between both drivers.

The extension of this structure to a fully distributed environment is shown in Figure 4.7. As indicated before, a distributed application usually consists of multiple

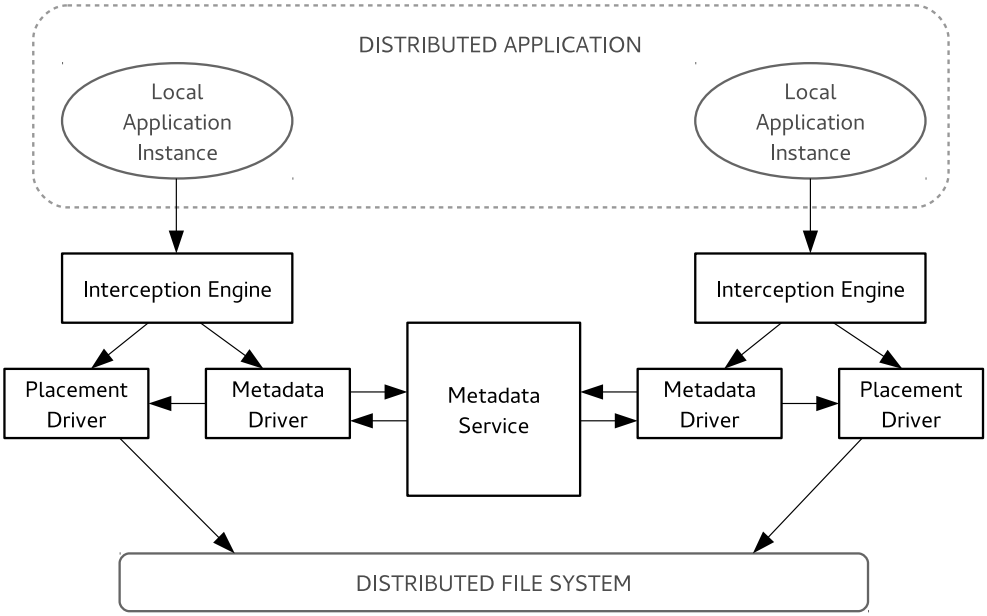

Figure 4.7: *COFS* setup for distributed environments.

local instances running on different nodes, and each instance communicates with a distributed file system via the local interface on the node where it is running. *COFS* captures this interface and intercepts the requests, diverting the metadata handling to the local *Metadata Driver*. The *Metadata Drivers* use a remote *Metadata Service* to coordinate operations on the metadata and the name space, ensuring its consistency across the system. Pure data requests are forwarded by the *Placement Driver* to a distributed file system, which is already able to deal with parallel data requests.

It is important to remark that even though we use a single metadata server in our current implementation, this is not forced by design, and the framework also admits a distributed *Metadata Service* (we will be back on this issue later).

Many of the characteristics and techniques used by the *COFS* have already been discussed in Section 3.5; nevertheless, the next section will elaborate on some relevant aspects of the implementation.

4.3.3 Implementation details

Interception Engine

An easy approach to capture the communication between the application and the file systems is making the *Interception Engine* appear as a conventional file system to the application. To this end, the prototype uses the FUSE technology [Szeredi 2005].

FUSE is a well-known and robust tool to intercept file systems operations and implement file systems replacements. Originally developed for *Linux*, it is currently

available for several operating system (including FreeBSD, NetBSD, *Apple Mac OS X*, OpenSolaris and GNU/Hurd), and has been also ported to *Microsoft Windows*.

FUSE provides a kernel module that exports VFS-like callbacks to user-space applications (Table 4.1 shows the list of callbacks available at the time of the implementation). The *COFS* prototype has been implemented as one of such applications receiving the file systems operation callbacks and resolving them from user-space. Considering our goals, the downside of missing some kernel-level information that is not exported or forwarded to user-space, and possible minor efficiency losses, is largely compensated by having a drop-in environment that can be used in most *Linux* boxes without requiring specific kernel modifications.

Apart from receiving the file systems callbacks, the *Interception Engine* based on FUSE is also responsible for providing the necessary support so that *COFS* appears as a regular file systems. This includes initializing the environment and providing the interface to attach it to the global POSIX directory tree. In particular, this allows *COFS* to be mounted as any other file systems.

Additionally the *Interception Engine* forwards the file systems callbacks to either the *Placement Driver* or the *Metadata Driver* (depending on the nature of the callback), also provides sensible defaults in case some of the callbacks are not handled by a particular implementation of the corresponding driver.

Placement Driver

The *Placement Driver* is responsible for mapping regular files into the underlying file system and also forwarding data-related operations to it.

Functionalities

More specifically, some of the functions covered by the *Placement Driver* are creating and removing regular files (and possibly other types of file system objects) in the underlying storage system, accessing the contents of files (`read` and `write` operations) and querying and setting specific attributes in the underlying file system.

It also provides complementary support for some metadata operations that require a direct interaction with the actual data (for example, getting the size attribute of a file). Additionally, it responds to *Metadata Driver* requests in case of combined operations; for example, opening a file requires checking the access permissions (a *Metadata Driver* task) and then getting a handle on the actual underlying file (which is provided by the *Placement Driver*). In the current implementation, all requests requiring both data and metadata access are sent by the *Interception Engine* to the *Metadata Driver* which, in turn, generates a request for the *Placement Driver* when necessary (this connection between both drivers is represented in both Figure 4.6 and Figure 4.7).

When a file is opened, the *Placement Driver* is responsible for returning an opaque file handle that contains the necessary information to access the file contents. This information is then stored in a FUSE internal structure, and passed back to *Placement Driver* when a subsequent operation has to be done to the open file. For instance, in

Table 4.1: FUSE callbacks (version 2.6)

Call	Description
`access`	Check file access permissions.
`create`	Atomically create and open a file.
`flush`	File handle clean-up on `close`.
`forget`	Notify that an *i-node* is no longer cached in the kernel.
`fsync`	Synchronize file contents with back-end devices.
`fsyncdir`	Synchronize directory contents with back-end devices.
`getattr`	Get file attributes.
`getlk`	Test for a POSIX file lock.
`getxattr`	Get an extended attribute.
`link`	Create a hard link.
`mkdir`	Create a directory.
`mknod`	Create a regular file, character device, block device, fifo or socket node.
`listxattr`	List extended attribute names.
`lookup`	Look-up a directory entry by name and get its attributes.
`open`	Open a file.
`opendir`	Open a directory.
`read`	Read data.
`readdir`	Read directory entries.
`readlink`	Read a symbolic link.
`release`	Notify that all references to an open file have been dropped.
`releasedir`	Release an open directory.
`removexattr`	Remove an extended attribute.
`rename`	Rename or move a file.
`rmdir`	Remove a directory.
`setattr`	Set file attributes.
`setlk`	Acquire, modify or release a POSIX file lock.
`setxattr`	Set an extended attribute.
`statfs`	Get file systems statistics.
`symlink`	Create a symbolic link.
`unlink`	Remove an non-directory entry from a directory.
`write`	Write data.

the usual case of a system backed by a POSIX file systems, such opaque file handle would contain, among other things, the file descriptor returned by an `open` system call to the actual underlying file.

Upon receiving a `release` callback, the file is closed and the *Placement Driver* is responsible for cleaning up any remaining state (including any related underlying file descriptor).

Policies

Apart from the functional aspects described so far, one of the relevant aspects of the *Placement Driver* is the policy used to determine the actual underlying path corresponding to the file systems objects in the application view. As mentioned before, the *COFS* prototype was designed to allow us to test easily different policies for both data organization and metadata handling.

During the development of *COFS*, different policy modules for the *Placement Driver* were implemented for testing and evaluation purposes. One of them was the *Null Placement Module*, which redirects all data-related requests to the local `/dev/null` device. Apart from providing an easy-to-implement data back-end for testing the metadata functionality without having to deal with an actual file systems, the *Null Placement Module* has proved to be a useful tool to determine the actual overhead that the *Metadata Virtualization Layer* adds to the underlying file system, allowing us to distinguish between the performance issues caused by the way in which the underlying file system handles data and metadata from the potential performance losses caused by *COFS* itself.

The actual module implemented to deal with the underlying file system issues was the *Sparse Placement Module*. This module was specifically designed to mitigate the issues observed in our large *IBM GPFS* clusters (see Section 4.2). It implements a very simple policy to re-organize the user's file hierarchy into something adequate to take advantage of the underlying file system, and to prevent situations that could harm the overall performance.

This placement policy consists in distributing user files into automatically generated paths that guarantee that no performance 'risk thresholds' are trespassed. Specifically, we control the maximum amount of entries per directory, so that directory caches can be exploited adequately, and directory-access synchronization is kept within reasonable limits.

The automatically generated path is obtained from a hash function applied to some input parameters (specifically the node issuing the creation request, the parent directory in the user's view of the file hierarchy and the process creating the file). Then, this base path is concatenated with some random bits, so that files with the same input values for the hash function may end up into different directories. Apart from that, the number of entries in a particular directory is bounded: if the limit is exceeded, additional directories are created.

The level of 'randomness' (i.e. the number of random bits in the generated path) can be configured. Obviously, at the minimum level, files tend to be grouped by the node issuing the creation request, the location of the file in the virtual user's view of the file hierarchy, and the process creating the file.

We have observed that slightly increasing the level of randomness mitigates the impact of *false-sharing* accesses to the same directory, which helps to balance the pressure in situations where a single process creates lots of files that are to be accessed by multiple clients, improving the performance. Nevertheless, experimentation showed that further increases do not lead to greater improvements, so that no dynamic fine-tuning is required for this parameter.

The *Sparse Placement Module* has proven useful to avoid bottlenecks originated by user's distribution of files, without harming the performance of the rest of the file systems operations. The results obtained with this module are presented in Section 4.4.

Metadata Driver and Service

The *Metadata Driver* is the responsible for responding to metadata-related requests forwarded by the *Interception Engine*, including name space management, object attributes, and global file system metadata.

Functionalities

One of the functions of the *Metadata Driver* is, therefore, translating file names given by the user to internal identifiers (the so-called *i-node numbers*). This is requested by the kernel (via FUSE) by issuing the `lookup` callback, which converts a file system object path to its corresponding *i-node number*.

Apart from translating from names to *i-node numbers*, the *Metadata Driver* takes care of creating and removing directories in the user view of the file system, managing the directory streams, renaming directory entries, and handling special objects such as symbolic and hard links (even if the actual underlying file system does not support them). The prototype's *Metadata Driver* also controls the operations to set and get object attributes and checks the access permissions.

Some operations do not consist in pure metadata manipulation, and interaction with the underlying file system may be required (for example, creating a regular file in a file system view may require an actual file being created in the underlying storage system; also, setting a file's size could involve changing the stored data, or some attributes may need being propagated to the actual file system). For this reason, the *Metadata Driver* can communicate with the *Placement Driver* to perform certain actions on the underlying storage system. The *Metadata Driver* takes care of maintaining any ancillary data needed to track this kind of operations (such as obtaining and releasing the appropriate file handles when opening and closing files).

The reason why some metadata operations (such access to file size and modification times) are delegated to the *Placement Driver* is that these values are quite related to actual data manipulation, and the underlying file system usually handles them

well; duplicating them in the metadata management and keeping them synchronized would be an unnecessary cost.

The above-mentioned operations are part of the interface between the prototype's *Interception Engine* and the *Metadata Driver*, and also between the *Metadata Driver* and the *Placement Driver*. As part of the modular design of *COFS*, these interfaces constitute a framework for different implementations of *Metadata Drivers*. A particular implementation does not need to implement all the operations of the interface, but only those required to fulfil the expected behaviour.

The Mnesia Metadata Module

As in the case of the *Placement Driver*, different modules with alternative implementations and policies can be used. For the purposes of improving the metadata behaviour of parallel file systems, the *Mnesia Metadata Module* provides a fully operative *Metadata Manager*. It has been designed to use a detached *Metadata Service* as a persistent back-end for the metadata information, including both name space data and object attributes. The implementation of this service is the *Mnesia Metadata Server*. The name *Mnesia* comes from the core database engine used by the service (*Mnesia* is a non-sql database distributed as part of the Erlang/OTP suite).

The *Mnesia Metadata Module* is able to receive metadata requests from the *Interception Engine*, retrieve and store metadata from and to the *Mnesia Metadata Server*, keep a local cache of part of the metadata, and communicate with the corresponding *Placement Driver* when necessary to interact with the underlying storage system.

The Mnesia Metadata Server

We took a conceptually centralized approach for the *Metadata Service* because of its simplicity. We dealt with scalability concerns by leveraging the well-know technology of distributed databases: the core of the *Mnesia Metadata Server* is a distributed database which, in case of need, could be distributed into several servers by the database engine itself (without the need of handling explicit file system partitions or separate volumes). Metadata is represented as a small set of tables in that database, keeping information about files, directories, and their relationship to form directory trees. As an example, Table 4.2 shows the fields of the database table containing basic file system object attributes in the initial implementation of the *COFS* prototype; Table 4.3 represents the information to locate a particular file system object in the underlying file system: the `fsid` allows the *Placement Driver* to identify a particular storage back-end (in case several of them are used simultaneously), while `path` is a reference to the object relative to such specific storage back-end; Table 4.4 stores symbolic link contents and, finally, the `ITree` table (Table 4.5) maintains the representation of the virtual name space, linking an object with the directory in which it is contained.

The `Inode` table (Table 4.2) contains an entry for each file system object (either regular file, directory, symbolic link or any other type) and keeps the object attributes. As it can be observed, it mostly mimics a typical *i-node* structure from a *Unix*-based file system. Nevertheless, there are some specific differences.

Table 4.2: Fields of the Inode table in the *Mnesia Metadata Server*

Columnn	Type	Description
`inum`	*integer*	File system object identifier.
`type`	*enumeration*	Object type.
`mode`	*bitmap*	Access permissions.
`nlink`	*integer*	Hard link count.
`owner`	*integer*	File owner user identifier.
`group`	*integer*	File owner group identifier.
`atime`	*integer*	Last access time for non-regular files.
`mtime`	*integer*	Last content modification time for non-regular files.
`ctime`	*integer*	Last attribute change time.
`size`	*integer*	Object size for non-regular files.
`parent`	*integer*	Identifier of the parent directory (directories only).
`ecount`	*integer*	Number of child entries (directories only.)

Table 4.3: Fields of the FileLocation table in the *Mnesia Metadata Server*

Columnn	Type	Description
`inum`	*integer*	File object internal identifier.
`fsid`	*integer*	*Placement Driver*-specific token.
`len`	*integer*	Length of the path field.
`path`	*string*	*Placement Driver*-specific object locator (usually a path).

Table 4.4: Fields of the SymbolicLink table in the *Mnesia Metadata Server*

Columnn	Type	Description
`inum`	*integer*	Symbolic link internal identifier.
`len`	*integer*	Length of the contents field.
`path`	*string*	Symbolic link contents (usually a path).

Table 4.5: Fields of the ITree table in the *Mnesia Metadata Server*

Columnn	Type	Description
`entry`	*tuple(integer,string)*	Parent directory internal identifier and entry name.
`inum`	*integer*	Internal identifier of the referenced file system object.
`type`	*enumeration*	Type of the referenced file system object.

One of the differences between a typical *i-node* structure and the contents of the `Inode` table is that object type and access permission bits are separated into different fields. In classical organizations, both elements are combined, usually by being stored in different groups of bits of the same value. Separating the `type` (which is immutable for a specific file system object) from the `mode` (which can be modified) allows for faster and easier type checks, as well as offering the possibility of storing both values separately and being able to cache the file object type without worrying about possible invalidations due to modifications to the `mode` section.

Regarding data access and modification times, the prototype only uses them for directories and symbolic links. For the rest of the file system objects, time-related queries and modifications are forwarded to the underlying storage system via the *Placement Driver* (as explained before).

A similar situation occurs with the `size` field: it is only maintained for directories and symbolic links (the meaning of `size` for a directory is left to the implementation and, for symbolic links, the prototype uses the size of link contents). For the rest of file system objects, size queries and/or modifications (truncations) are also forwarded to the underlying storage system.

Finally, there are two fields that are used only for directories: `parent` and `ecount`. The value of `parent` is the file system object identifier of the parent directory. For a generic object, this has no meaning because the prototype supports hard links and a file system object may potentially have several parents in the hierarchical user view. Nevertheless, directories cannot be hard-linked under the supported POSIX semantics (except for "." and ".." entries — but these are special cases and treated separately) and then a directory can only have a single parent. Keeping this information allows certain optimizations in the internal algorithms.

The field `ecount` contains the number of entries in the directory. Maintaining this value allows speeding up some operations that depend on the number of entries of a directory, but not on the specific contents (for example, to know if a directory can be removed, it is necessary to know if it is empty or not).

In an experimental version of the prototype, a different table was added to keep the directory specific information such as the `ecount` and `parent` fields, instead of including them in the `Inode` table. This table also contained a `name` field with the user-

given name of the directory. This, combined with the parent information, allows the reconstruction of canonical paths for directories, which is interesting for certain policy features. Nevertheless, the experience showed that the handling of the additional table increased the complexity of error management and the path reconstruction itself added noticeable overhead; as a consequence, the additional table was discarded in the *COFS* prototypes targeted to HPC systems.

The `ITree` table (Table 4.5) constitutes the basic infrastructure to provide the user views of the file system organization. It contains the directory entries that associate a name in a particular directory with an internal file system object identifier. According to the standards supported by the prototype, there cannot be two objects with the same name inside the same directory.

The `ITree` table contains all directory entries, and it is indexed by the combination of the identifier of the parent directory, and the name of the child object inside that directory. The *inum* field contains the identifier of the child object (which may be a directory, a symbolic link, a regular file, or any other kind of file system object).

There may exist several directory entries with the same *inum* value. This means that the same file system object may have several names (or the same names but accessible from different directories). Technically, this feature is known as hard link, and the model supported by the prototype permits it for all file system object types except directories (with the exception of "`.`" and "`..`" entries, which are hard links to the directory containing the entries and its parent, respectively — as special cases, they are handled separately and do not have explicit directory entries). It is important to note that, being completely handled by the *Mnesia Metadata Module* and the corresponding *Mnesia Metadata Server*, the prototype supports hard links even if the underlying file system does not support them.

The directory entry also contains the `type` of the child object. Its value corresponds to the field of the same name in the object attributes table, but it is cached here so that some basic type checks can be done without the need to recover all the attributes. (Note that being the `type` immutable during the life time of the object, no effort is needed to keep the fields in both tables synchronized.)

A remarkable difference between the *COFS* prototype using the *Mnesia Metadata Module* and most directory implementations in file systems is that there is no explicit structure actually grouping the entries belonging to the same directory. In other words, there is no real directory structure: all directory entries (from different directories) can be held together in a single table. In this scenario, a 'directory' is defined as the subset of directory entries with the same parent identifier.

Active objects engine

The *Mnesia* database is specially well-suited to handle the tables with file system metadata information. *Mnesia* provides a database environment optimized for simple queries in soft real time distributed environments (with built-in support for transactions and fault tolerance mechanisms). Additionally, the Erlang language has proven to be a good tool for fast prototyping of highly concurrent code (the language itself

internally deals with thread synchronization and provides support for transparently distributing computations across several nodes).

The *Mnesia Metadata Server* keeps the current working set of metadata information as a cache of active objects to reduce the pressure on the back-end database engine, using a concurrent caching mechanism similar to the one described by Jay Nelson [Nelson 2006]. Even as our performance tests showed that a single node is enough to handle the metadata, the used algorithm would be compatible with a distributed multi-node Erlang system if needed.

Following the principles explained in Subsection 3.5.6, the active objects cache mechanism implemented in the *Mnesia Metadata Server* is an alternative to the use of conventional locking mechanisms to deal with concurrent access to multiple objects from multiple clients. Instead of locking data structures to allow its use from different threads or processes, each data structure has an assigned thread or process which is the only one with permission to modify the data. Our experimental results (see Section 4.4) show that this technique helps to alleviate the lock contention that appear when using conventional lock methods in situations with lots of activity, or when large groups of structures have to be manipulated (even with no conflicts, acquiring and releasing locks may produce a certain overhead).

The *Mnesia Metadata Server* implemented in the prototype associates an Erlang process to each file system object. An Erlang process is a very light-weight thread (according to the specifications [Erl 2013] and our own tests, a single Erlang node can handle millions of them simultaneously); the programming model allows them to be activated by communication events and, when activated, they execute their code in a 'run-to-completion' mode, without preemptions.

The Erlang process associated to a file system object is the only one with permission to modify the data associated to the object, including the object attributes, the underlying file location information, link contents data and directory-specific fields, if the object is a directory. In practice, these Erlang processes act as 'active objects' that cache the information from the *Mnesia* database and use it as a persistent back-end. Therefore, the preferred way to obtain file system object information is not querying the database directly, but sending a message to the corresponding process asking for the desired piece of data.

Under certain circumstances, other processes may be allowed to read pieces of data that do not belong to them directly from the database. This information is usually treated as a hint. In complex operations such as the creation of a file system object, the responsible Erlang process follows a strictly ordered sequence of actions that allow other threads to check for the correctness of the data if accessing the database directly. For example, the fact that internal identifiers are never re-used allow a process to check a hinted identifier against the actual one before attempting an operation on a hinted object that may have been already deleted.

The fact that only one process is allowed to modify a specific piece of data in the database, and the strict ordering in which several parts of the data are written,

allow avoiding the need to enclose certain operations in large and costly database transactions, even when read is allowed from different processes (it is important to mention that, in *Mnesia*, row changes are atomic, even outside transactions). For example, when a new file is created, the attributes are modified first, and then the entry in the name space; therefore, if an external process is able to follow the path up to the object, it is sure that the attributes have been already set.

Naturally, not all objects need to have an associated active process at all times. The process may die and disappear after a period of inactivity, and be re-created when the associated object is needed again.

In summary, the overall operation mechanism is as follows: when a request arrives to the *Mnesia Metadata Server*, it is forwarded to the process associated to the target object (if no process exists, it is created and, then, it reads all the object information from the database, keeping it internally cached); if the request involves a modification, the database is updated and, eventually, the requested data is included in the response. Once the process sends the response back to the caller, it waits for the next request.

In cases where several objects are involved (for example, removing a file, which affects the file itself and the parent directory) the request is sent to the 'main' object process (the parent directory, in this example) which may then communicate with the process of the other involved object (the file to be removed) via messages.

Object identifiers

The *Mnesia Metadata Server* also assumes the responsibility of generating internal identifiers for the file system objects belonging to the applications file system views. When the *Mnesia Metadata Module* needs identifiers for new objects, it requests them to the *Mnesia Metadata Server*, which centralizes the requests and guarantees that all clients receive different identifiers. Specifically, the implementation in *COFS* uses a monotonically increasing counter, and it has the particularity that assigned identifiers are never re-used, even if the corresponding object has been deleted (for long running systems, counter overflow is not a problem because the number of bits used for the counter can be dynamically increased). In order to avoid frequent requests, the *Mnesia Metadata Module* can request a range of identifiers (instead of a single one), and keep them locally and use them whenever needed.

As mentioned in Subsection 3.5.2, the internal identifiers must be mapped to *i-node numbers* to be used by a POSIX system. In the particular case of the *COFS* prototype, the size of the internal identifiers matched the size of the file system *i-node numbers*; so, a direct mapping was used.

Caching

The fact that most metadata-related information is generated and kept at the remote *Mnesia Metadata Server* makes relevant the possibility of caching this information locally in the *Metadata Driver*.

Seeking simplicity, our first approach was following the *PVFSv2* model: having no client cache at all and forwarding to the *Metadata Service* [Sebepou 2008]. Though

this was useful for validating the prototype, most of the operations resulted in several round trips to the *Metadata Service* for path validation, permission checks, etc. Under pressure, the cost was affordable compared to the native file system; but resulted in a high overhead on more relaxed situations (namely when there were few nodes involved and there was high locality).

Alternatively, an appealing possibility would have been coalescing requests in the *NFSv4* style [Pawlowski 2000], or using *intents* (in the way of *Lustre*'s locking protocol [Sebepou 2008]) to have the *Metadata Service* return additional information that would be immediately needed. Unfortunately, the interface provided by the FUSE infrastructure made this particular approach not feasible (as the *intent* information was missing).

Eventually, we decided to implement our own client cache mechanism to keep metadata information (*i-node* like data, the actual location in the low level file system for the regular files, and symbolic link contents). The information is provided by the *Mnesia Metadata Server* with a limited-time lease. Additionally the *Mnesia Metadata Server* may issue an invalidation request whenever the affected metadata is going to be modified by a different client. The cached information was kept in user-space by the *Mnesia Metadata Module*.

There are two main reasons for using an ad-hoc cache mechanism instead of the *i-node* kernel cache accessible via the FUSE interface: first, the information to cache did not directly match the *i-node* structures kept by the kernel (so that some of the information could not be cached — for example, the location information); second, at the time of the implementation, there was no mechanism to invalidate the information cached in the kernel before its 'natural' expiration. Apart from this issues, using a kernel-based cache would be a valid alternative that could even reduce the cross-boundary communication between the kernel and the user-space.

A particular detail of the implementation is that lease handling can be decoupled from metadata requests; in other words, the lease is not sent back to the client together with the response: instead, the request is responded as fast as possible, and it triggers a concurrent mechanism that will end up in a lease being sent to the client afterwards. Though this may not seem intuitive, we have experimentally verified that it results in a better response time in situations where *Metadata Service* was instantiated in a low-end hardware, where processing took substantially more time than network communication. In this circumstances, the cache efficacy was not affected by the delay in a significant way. On the contrary, we observed that decoupling of lease handle had no effect on deployments of *COFS* in high-end systems.

The implementation of lease management in the *Mnesia Metadata Server* is as follows: when a cache request is accepted, an Erlang process representing the requester *Metadata Driver* is created. From then on, all the cache-related interaction with that particular *Metadata Driver* in both directions (possibly including lease revocation, voluntary releases of cached data, and lease renewal) is done through the newly created process. When no data is cached, the process may die after a period of inactivity.

The number of leases granted for a specific piece of information is limited. This keeps the synchronization and invalidation costs bounded. Beyond this limit, the system works like a no-cache system. Again, no drawback has been observed for this, as the other benefits obtained from reorganizing the directory layout compensate for the lack of cached data (in some cases because the reorganization prevented the need for large shares of cached data, and in other cases because the cost of synchronization is larger than the additional round-trips to fetch the required information).

Directory listing

Finally, another important function of the *Metadata Driver* is generating directory listings for the user views. The directory stream operations include `opendir`, `readdir` and `releasedir`. The POSIX standard leaves to the implementation to decide what happens if the directory is modified (i.e. entries are added or removed) between the `opendir` and `releasedir` operations. In that case, the `readdir` operation could return the entries before any modification (when the directory opened), the entries after all modifications, or any state in between.

In the *COFS* prototype implementations, the *Mnesia Metadata Module* requested all directory entries from the *Mnesia Metadata Server* at `opendir` time, cached them locally, and then progressively returned data from that cache whenever an application demanded it via a `readdir` operation. The cache was released on directory closing by `releasedir`.

The advantage of this method is that the whole operation is done at once, and the *Mnesia Metadata Server* does not need to maintain any state to know which entries have already been delivered. This avoids several possible locking and synchronization issues in the *Mnesia Metadata Server* (for example, in the case that someone opened a directory, but it did not read the entries or released it at the end). In extreme situations, the number of entries could be too large to be handled at once; in that case, the techniques for partitioned stateless directory listings discussed in Subsection 3.5.7 could be used.

4.3.4 Potential technology limitations

A final caveat on performance impact is related to the use of FUSE as the base infrastructure for intercepting file system calls. FUSE intercepts the file system operations in the kernel, forwarding the requests to a user-space daemon (the *COFS* client, in our case). The user-space daemon then performs the necessary operations and sends the answer back to the kernel, from where it is returned to the application originally requesting it.

Obviously, this requires additional user-space/kernel boundary crossings and more additional memory copies for data, compared with a direct file system access.

A few additional kernel boundary crossings per I/O operation are not an issue because they represent a constant overhead (in the order of nanoseconds) almost negligible in modern computer systems (compared with the overall cost of an I/O

operation, which may be up to the order of milliseconds).

Regarding the additional memory copies, the data is requested by the FUSE/*COFS* daemon to the underlying file system, temporarily stored in a daemon buffer, and then copied back again to the kernel to be forwarded to the original client application. This is intrinsic to FUSE implementation and involves, at least, a double copy of the same data. This issue was discussed regularly in the FUSE developers mailing lists [Szeredi 2005], but for a long time there was no plan to modify that behaviour, as the experiments done showed no conclusive evidence of a significant performance impact. Only the most recent versions include experimental code to avoid partially the additional memory copies of file data (but not metadata).

Our own tests indicate that the double-copy effect is not relevant for metadata operations because the amount of data being copied is small. Regarding data read and write operations, the impact in performance is noticeable for those cases where there are relatively large data transfers which can be handled completely from the local page cache (i.e. no actual disk access and no network access are performed).

Fortunately, there is a very limited window for these situations in our target environments: transfers must be big enough for the memory copy time to be significant, small enough to be cached, and should be local (to avoid the need for inter-node synchronization). For most cases, the performance drawbacks due to double copies are negligible or reduced to an acceptable small factor. In Subsection 4.4.1, we comment the test results regarding data transfers, which show a very limited impact.

4.4 Evaluation

As we have mentioned, we believe that virtualization techniques based on decoupling the name space and file metadata from the actual location in the underlying file system can be used to mitigate a series of performance issues that affect current large-scale file systems.

In this section, we present the evaluation results of a proof-of-concept *Metadata Virtualization Layer*: the *COFS* prototype. As described in Section 4.2 the specific problem motivating the experiment was a drop of performance observed in some of our production *IBM GPFS*-based clusters, where the workloads used directories with a large number of entries and produced synchronization overheads that eventually affected the servers, causing a general slow down (for example, during simultaneous file creation for parallel application check-pointing).

The following subsections show the results obtained by *COFS* in three different environments. The first environment is Nord, a test-bed cluster using *IBM GPFS* file system at BSC, where the initial development and testing of *COFS* was performed. The second environment is MareNostrum, a production-grade supercomputer at BSC also using *IBM GPFS* as file system. Finally, the third set of results show the performance of *COFS* at CEA INTI system, a test-bed cluster for the PRACE-2IP European project, using the *Lustre* file system.

4.4.1 Metadata performance on GPFS at Nord

We have evaluated the *COFS* effects on performance using the Nord test-bed cluster, described in Subsection 4.2.2. When using *COFS*, one of the blades in the blade center (with 2 PPC970FX processors and 4 GiB of RAM) was used to host the *Mnesia Metadata Server*; the back-end for the the *Mnesia* database was a 25 GiB disk locally attached to that node and formatted with the *Ext3* file system.

We have used *Metarates* [Met 2004] as the main benchmark. Directories were populated with different amounts of files (from 256 to 32,768, depending on the experiment) and accesses were done simultaneously from 4 to 16 nodes.

In order to verify that interposing the *COFS Metadata Virtualization Layer* had no significant negative impact on pure data transfers, we have also used *IOR* (Interleaved Or Random I/O benchmark [Ior 2013]) to measure data I/O performance. *IOR* v2 was developed at Lawrence-Livermore National Laboratory and provides aggregate I/O data rates for both parallel and sequential `read`/`write` operations to shared and individual files in a parallel file system. The benchmark was executed using the POSIX interface with aggregate data sizes of 256 MiB, 1 GiB and 4 GiB, in the form of a single shared file accessed in parallel and also having a separate file for each parallel process.

As in the initial study (Subsection 4.2.2), our test-bed cluster was not exclusively dedicated to benchmarking: our measurements were done while the rest of the cluster was in production (specifically, we had an allocation of 16 nodes out of 70 for most of the tests). The methodology employed to run the experiments minimized the impact of punctual variations of system behaviour, getting acceptable statistical confidence levels by repeating the experiments as many times as needed (usually between 30 and 50 times, and never less than 16 measurements).

Virtualization results

Unless stated otherwise, *COFS* measurements in this section have been done using the *Sparse Placement Module*, limiting the low level directory size to 512 entries, and using 1 random bit as randomization factor (distributing related entries into two separate directories). We have observed that there are no substantial differences in parallel performance between using 1 and 2 processes per node at Nord (*IBM GPFS* client seems to be able to parallelize some requests when using 2 processes per node, while the FUSE component in *COFS* apparently serializes them; nevertheless, differences are marginal compared with overall values). As the trends are the same, we will focus on the results using a single process per node.

Figure 4.8 shows the benefits of breaking the relationship between the virtual name space offered by the *COFS Metadata Virtualization Layer* (exporting a single shared directory to application level) and the actual layout of files in the underlying *IBM GPFS* file system. By redistributing the entries into smaller low level directories, *COFS* allows *IBM GPFS* to fully exploit its parallel capacity by converting a shared parallel workload into multiple local sections that do not require global synchronization.

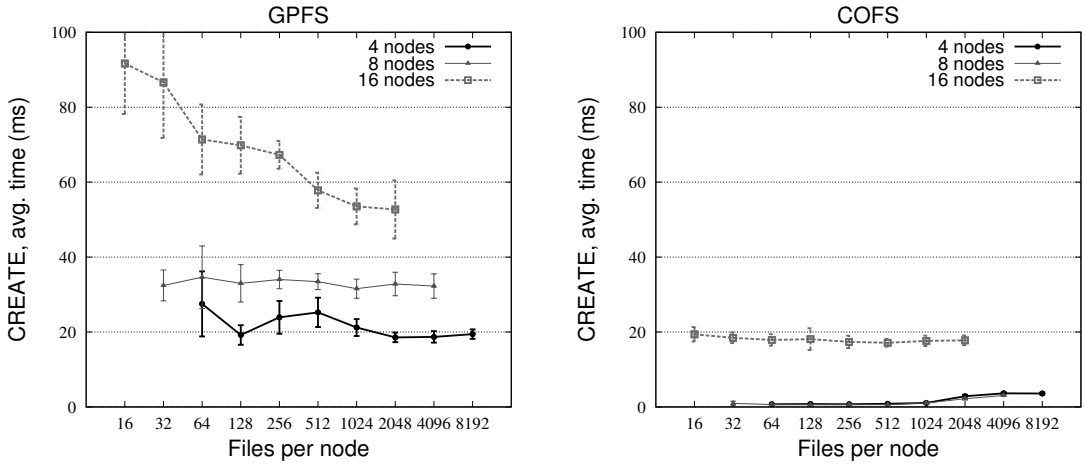

Figure 4.8: Effects on the `create` time in a shared directory of the *COFS* virtualization layer over *IBM GPFS*, using 1 process per node at Nord.

The overhead of using 8 nodes in *IBM GPFS* disappears when *COFS* is used. For 16 nodes, there is also a big improvement due to the elimination of synchronization needs, but there is a performance barrier slightly below 20 ms. Some network measurements confirmed that this was not related to *COFS* itself, but to the particular topology of our test-bed (as already mentioned in Subsection 4.2.2).

Figure 4.9 and Figure 4.10 show the average time for `stat` and `utime` operations as a function of the number of files in a shared directory accessed by each node (figures for `open/close` operations are not shown as they closely resemble the `stat` figures for both *IBM GPFS* and *COFS*).

Overall, we can see that all the figures follow a similar pattern: there is a first phase of large operation times when only a few files per node are accessed, that converges when the number of files per node increases. It is worth noting that the stable times for `stat` and `utime` are considerably lower than `create` times; this is mainly due to an effective use of caches for larger amounts of files (which cannot be exploited for `create` operations).

Times for `utime` operations are slightly higher than times for `stat` operations. One of the main reasons for these are *i-node* cache invalidations caused by the modification of data. On this respect, we must mention that we have observed noticeable *false-sharing* effects inside the *i-node* structure: some fields are rarely modified (type and permissions) but frequently used, and they are affected by modifications to more volatile fields (such as access times) which are not so often needed. Despite of this fact, *i-nodes* are usually handled and cached atomically in most implementations.

The *COFS* layer remarkably reduces the `stat` time when a directory grows beyond 512 entries per node (from approximately 7 ms to 1 ms for 8 nodes, and from 5 ms to 1 ms for 4 nodes). Some of the variability (indicated by the error bars representing

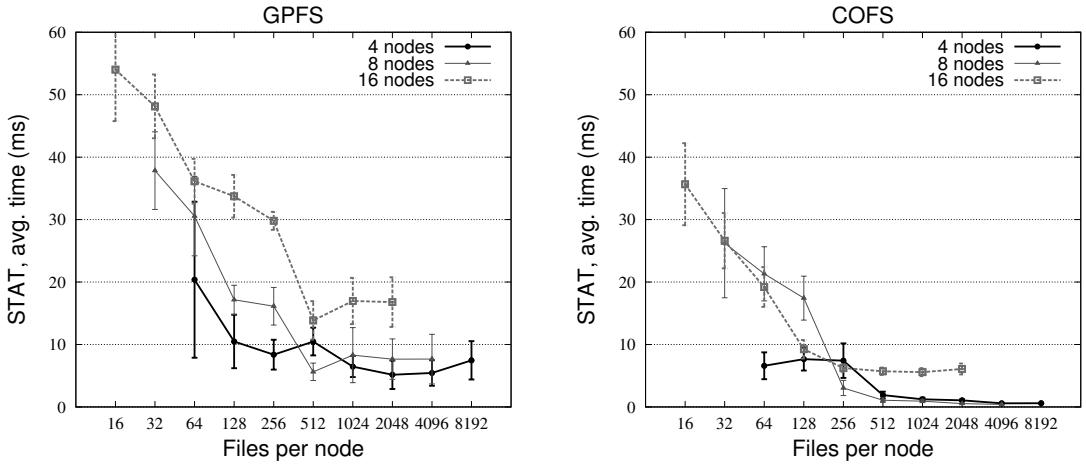

Figure 4.9: Effects on the `stat` time in a shared directory of the *COFS* virtualization layer over *IBM GPFS*, using 1 process per node at Nord.

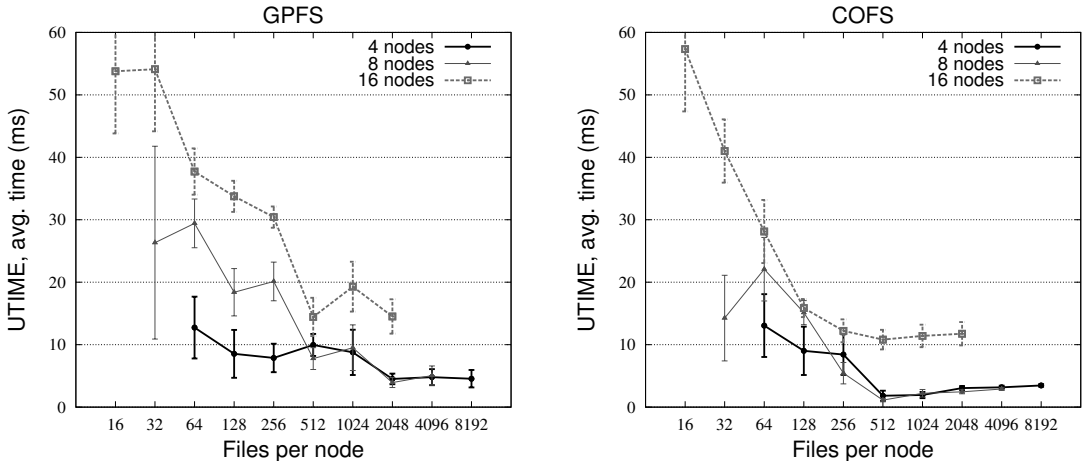

Figure 4.10: Effects on the `utime` time in a shared directory of the *COFS* virtualization layer over *IBM GPFS*, using 1 process per node at Nord.

the standard deviation) caused by network noise is also eliminated, and Figure 4.9 shows that *COFS* operation times for 4 and 8 nodes are nearly identical. Even for 16 nodes, where the network bandwidth is reduced, the speed-up for large directories is noticeable (from approximately 17 ms to 6 ms per operation).

The `utime` time is also improved with *COFS* (from about 5 ms to 1–3 ms for 4 and 8 nodes). Again, the effect is clearer for 16 nodes (from approximately 18 ms to 11 ms, as seen in Figure 4.10).

Below 128 operations per node, the *COFS* setup is only able to reduce slightly the high access cost for small shared directories, but it does not incur in any additional

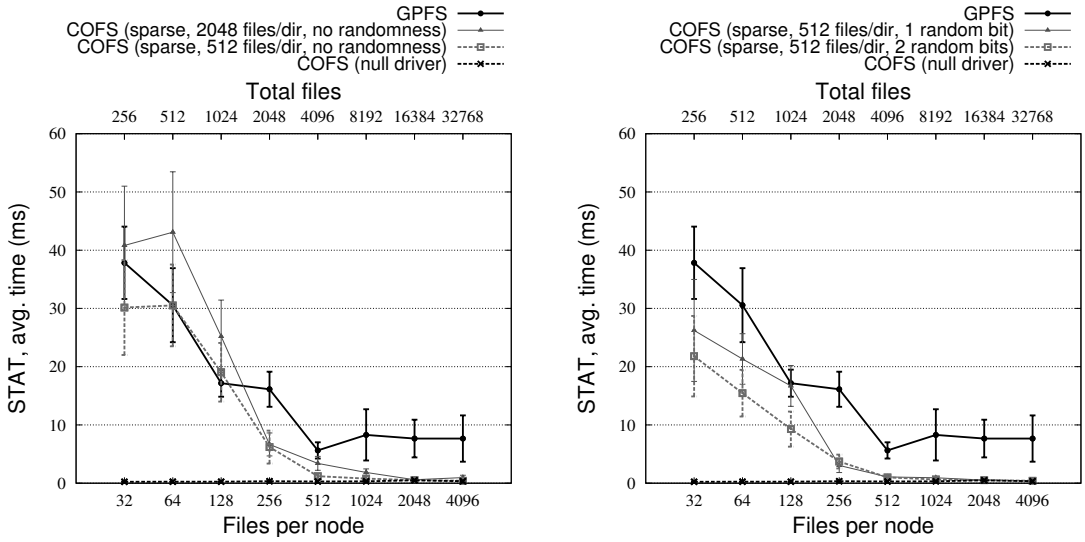

Figure 4.11: Changes in `stat` time in a shared directory for several *COFS* configurations, using 8 nodes and 1 process per node at Nord.

overhead. Even then, results for 16 nodes show a better behaviour than pure *IBM GPFS* (less variability and faster convergence to reduced operation times). That makes us expect that the benefits of *COFS* will be more noticeable for a larger number of nodes (as we will show in the next subsection).

In order to understand thoroughly the causes of *COFS* benefits, we conducted some additional experiments. Figure 4.11 shows the results for 8 nodes: we compared pure *IBM GPFS* behaviour with the results of *COFS* using different sets of parameters for the *Sparse Placement Module* (increasing the number of entries per directory in the underlying file system from 512 to 2,048, and modifying the layout randomness factor); we also executed the tests using the *COFS Null Placement Module* (which diverts all operations to "`/dev/null`") in order to distinguish pure *COFS* behaviour from that related to the underlying file system.

The cost of accessing few files is mostly due to *IBM GPFS* (as the overhead does not appear with the *Null Placement Module*), and this cost is partially related to synchronization: increasing the random bits in the sparse layout distributes effectively the files into a wider number of underlying directories, and we can observe that this helps to mitigate the problem (by reducing the chances of simultaneous colliding accesses from different nodes — as a matter of fact, we are reducing the level of *false-sharing* inside the back-end directories).

The effect of layout randomization is complemented by the limitation of the directory size. When the maximum number of entries is increased up to 2,048 entries, the synchronization overhead is also increased, resulting in larger operation times (to better understand this effect, it is worth noting that the benchmark creates all the files to be from a single node in such a way that files to be accessed by node 1 are created

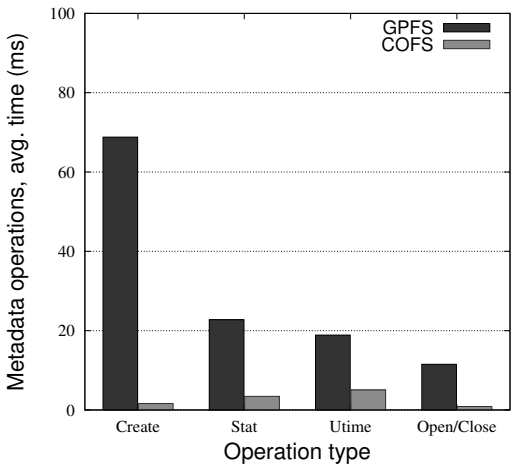

Figure 4.12: Operation time on a shared directory at Nord with 256 files per node, using 64 nodes and 1 process per node.

first, then those for node 2, and so on; beyond the directory size limit, the *COFS* data driver manages to put each node's files into different directories, effectively converting shared accesses into non-shared accesses). This way, access time approaches the *Null Placement Module* results. On the contrary, pure *IBM GPFS* is unable to reduce the operation time.

In summary, our measurements show that using the *Metadata Virtualization Layer* provided by the *COFS* framework can drastically boost our underlying *IBM GPFS* file system for file creations on shared parallel environments (with speed-up factors from 2 to 10, as shown in Figure 4.8). For the rest of metadata operations, performance is also boosted (though the speed-ups are more moderated), and the overhead and variability for parallel access to small-sized directories is reduced.

Scalability

One of our concerns was to check that our proof-of-concept prototype was able to deal with environments larger than our test-bed. So, we conducted some additional experiments to probe possible scalability limits in two different aspects: number of files in the file system and number of nodes in the distributed environment.

In order to check how many entries was able to handle a single *COFS* metadata server, we artificially populated the metadata database with a large directory tree (with a proportion of 10% directories and 90% files), and then re-run the experiments on it. Results show no significant degradation on `create` operations up to 8,000,000 entries (which is one order of magnitude larger than the actual file system in our test-bed). With respect to other metadata operations (`stat`, `utime`, `open/close`), results were stable up to 10,000,000 entries (where we started hitting physical memory limits — the metadata server uses a JS20 blade with 4 GiB RAM). These results were promising, given the limited capacity of the hardware used to run the *Mnesia Metadata Server*; for the sake of comparison, at the time of these measurements, the home file system of the MareNostrum supercomputer was using about 19 million *i-nodes* (which

gives an indication of the number of file system objects). Further scalability should not be a problem, as mechanisms to deal with creations in larger tables in *Mnesia* are discussed in Erlang-related literature [Erl 2013], and node physical limits can be bypassed by using *Mnesia* distribution mechanisms [Nelson 2006]. Also, several works on file system metadata focus on partitioning techniques to distribute metadata and name space information to multiple nodes [Douceur 2006] [Patil 2007] [Weil 2004].

Regarding the number of nodes in the cluster, we had the opportunity to use temporarily a larger partition of the test-bed cluster (with a very similar network configuration and the same hardware components). On this enlarged test-bed, we were able to confirm that the benefits of *Metadata Virtualization Layer* were not only maintained but increased in larger scales. As an example, Figure 4.12 shows the comparison of metadata operation times on 64 nodes, accessing 256 files per node on a shared directory (results with other directory sizes show similar trends).

As we may observe, the base-line given by pure *IBM GPFS* shows considerably high operation times for 64 nodes, because the inter-node conflicts when accessing a shared directory increase with the number of nodes, and they are the main component of the operation cost. On the contrary, *COFS* is able to mitigate such conflicts, allowing *IBM GPFS* to obtain remarkable speed-ups. We expect to observe the same trend for a larger number of nodes.

Impact of COFS infrastructure on data transfer

As shown in the previous subsections, the benefits obtained by our prototype regarding metadata handling were promising, and they effectively mitigate the issues motivating the present work.

Nevertheless, there was a concern about the possible impact on `read` and `write` operations on file contents: theoretically, altering the file hierarchy could lead the underlying file system to modify the actual location of data, impacting negatively on `read/write` bandwidth; additionally, we wanted to be sure that the *COFS* infrastructure was not adding an unacceptable overhead to data transfer operations. Some of the possible causes of overhead have been outlined in Section 4.3 and include caching issues, FUSE double buffer copying and extra kernel-boundary communication, and a higher number of round-trips to the *Metadata Service* due to lack of additional kernel-level information (like the file system *intents* when requesting metadata). In order to rule these possibilities out, we conducted a series of experiments using the *IOR* I/O benchmark [Ior 2013]. This benchmark measures the bandwidth when reading and writing data in parallel; each participating process operates on a series of blocks that may belong to either a shared file or to different files for each process (the "aggregated size" parameter indicates either the size of the single shared file or the sum of the individual files, respectively).

We executed one *IOR* process per node accessing either separate files per node (in the same directory) or a single shared file in parallel. The aggregate data sizes used were 256 MiB, 1 GiB and 4 GiB (either distributed in separate files, or accumulated in

Table 4.6: Impact of *COFS* on data transfers, depending on use pattern

	Separate files per process	**Single shared file**
Sequential read	*COFS* performance comparable to *IBM GPFS*, except for small files (<32 MiB per node), where *COFS* suffers an important slowdown.	*COFS* performance comparable to *IBM GPFS*.
Random read	*COFS* performance comparable to *IBM GPFS*, except for small files (<32 MiB per node), where *COFS* suffers an important slowdown.	*COFS* performance comparable to *IBM GPFS*.
Sequential write	*COFS* performance drawback for a single node, and performance improvements of *COFS* over *IBM GPFS* as the number of nodes is increased.	*COFS* performance drawback for a single node, and comparable performance for multi-node.
Random write	*COFS* performance comparable to *IBM GPFS*, except for small files (<32 MiB per node), where *COFS* suffers form slight slowdown.	*COFS* performance comparable to *IBM GPFS*.

a single file, depending on the execution mode). The block and transfer sizes for *IOR* were 4 KiB (as the stable version of FUSE at the time of the experiments imposed this limit for write data transfers). The bandwidth measures produced by *IOR* include both the data transfer time and the file `open/create` and `close` times.

Table 4.6 summarizes the results of our experiments. Overall, *COFS Metadata Virtualization Layer* using FUSE over *IBM GPFS* is usually able to obtain a performance similar to native *IBM GPFS*. The most remarkable exceptions occur when each node accesses a separate set of relatively small files.

Figure 4.13 shows the *IOR* bandwidth obtained for sequential reads. When all nodes share a single file, performance of *COFS* and native *IBM GPFS* are comparable, without significant speed-ups or penalties. We may assume that the performance of shared access is dominated by the mechanisms used to guarantee a consistent view of the data from the different nodes, because the obtained bandwidth is below the nominal bandwidth expected from the network setup. As pure data transfers do not seem to be the limiting factor, the overhead introduced by FUSE infrastructure is not noticeable. Also, metadata handling mechanisms have little to do in this case.

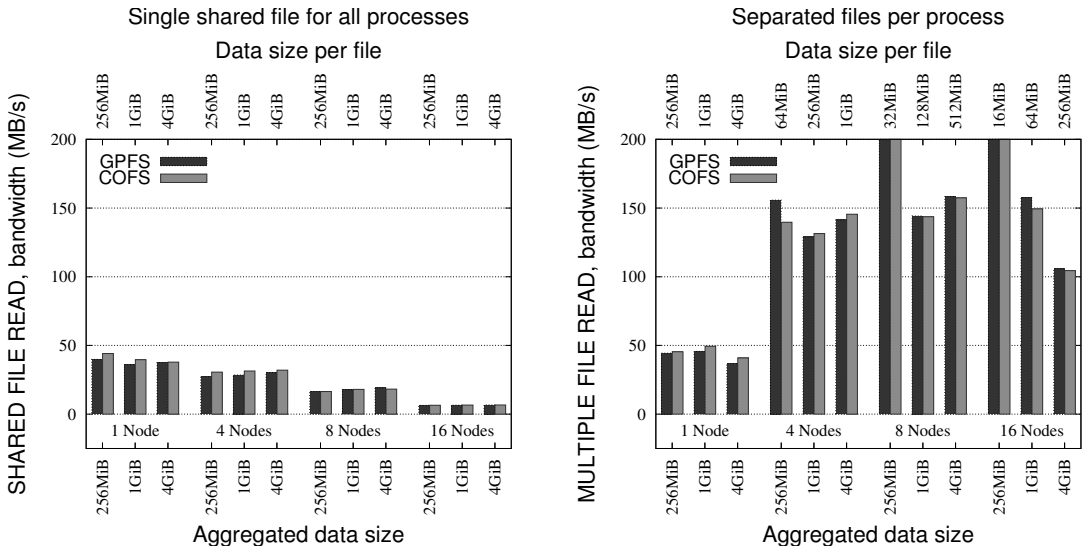

Figure 4.13: Consecutive (serial) `read` performance comparison between *IBM GPFS* and *COFS* over *IBM GPFS* at Nord, using 1 process per node.

When each node works with its own dedicated file, both *COFS* and *IBM GPFS* have a similar behaviour when accessing the same aggregated sizes, and the performance varies around 150 MB/s. Nevertheless, a notable exception occurs for an aggregated data size of 256 MiB and 8 nodes. In this case, the bandwidth values (out of the figure's scale) go up to 1,500 MB/s for *COFS* and up to 4,250 MB/s for *IBM GPFS*. The values for 16 nodes are also high: 1,250 MB/s for *COFS* and 2,500 MB/s for *IBM GPFS*.

In these situations, the individual file size falls below 32 MiB and, having a closer look at the experiments, we saw that the total time for the data transfer time for *IBM GPFS* was about a few milliseconds (which was comparable to the `open` time). It was then clear that *IBM GPFS* was able to keep the file data completely in the local cache (as it was created on the same node) and, therefore, the transfer consisted in a memory copy with negligible cost.

Under these circumstances, *COFS* is paying the cost of its infrastructure, partly due to the FUSE extra data movement (discussed in Section 4.3) and partly due to extra round-trips to the *COFS Metadata Service*. Due to the small amounts of time involved (tens to hundreds of milliseconds), the slight increment in operations time appear as considerable differences in bandwidth ratios. Improvements on *COFS* leasing and caching mechanisms could probably mitigate this difference.

The very same effect appeared also for random reads to separate files per process. The orders of magnitude of the measurements are fully comparable with those for sequential reads to individual files per process (note that data caching minimizes the effect of randomizing the read access for small files).

Regarding random writes to separate files, we have observed a similar pattern at a different scale: there are no remarkable differences between *COFS* and *IBM*

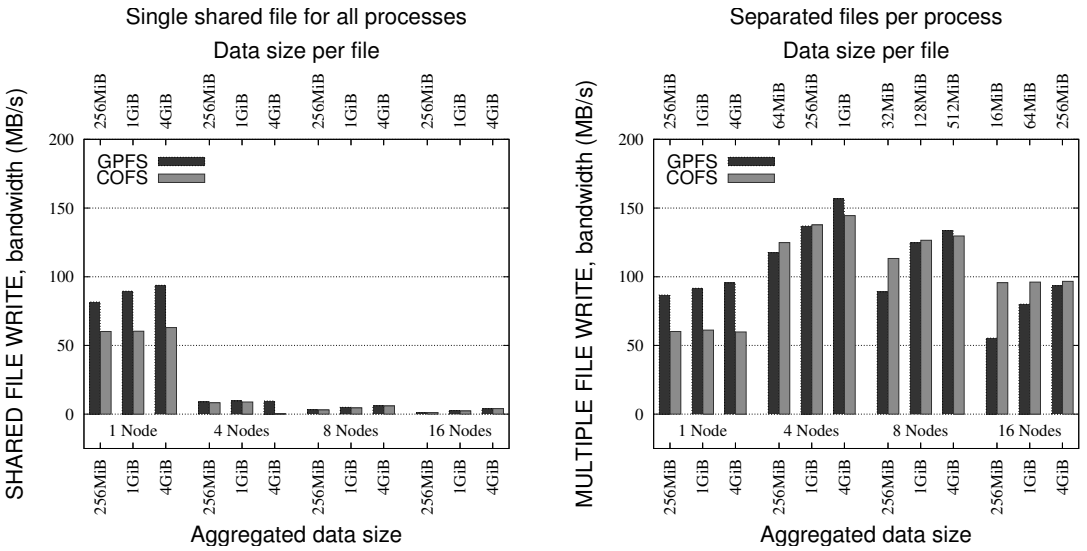

Figure 4.14: Consecutive (serial) `write` performance comparison between *IBM GPFS* and *COFS* over *IBM GPFS* at Nord, using 1 process per node.

GPFS performances, and the bandwidth is between 1.5 MB/s and 2.5 MB/s, except for aggregate sizes of 256 MiB and 8 nodes, where the performance raises up to 90 MB/s.

Nevertheless, the relative difference between *COFS* and *IBM GPFS* for the peak is much smaller (almost negligible). *IBM GPFS* cannot exploit a pure local cache operation (as it was the case for `read`) as we have to eventually send the written data to the file system servers. So, effective transfer time increases, hiding part of the extra cost of the *COFS* infrastructure.

Sequential data writes present quite a different behaviour, as shown in Figure 4.14. On one side, the *COFS* overhead is clearly noticeable when executing *IOR* in a single node. In fact, the 'single file in a single node' situation corresponds to the worst-case scenario for our platform, as there is no room for advantage by reorganizing the location of the (single) file.

Nevertheless, the *IBM GPFS* advantage disappears as we use more nodes. For concurrent writes to a single shared file, performance of both systems is comparable. In this case, the cost is dominated by the mechanism to combine data writes from multiple clients, and metadata reorganization has no significant effect.

On the contrary, when writing several files, we can appreciate that *COFS* is able to maintain a certain level of performance, while *IBM GPFS* behaviour suffers a fast degradation as the number of nodes is increased. This is an example of how improving the metadata behaviour can produce a better use of data transfer bandwidth.

Looking at the *IOR* numbers corresponding to file creation times (obtained from the detailed benchmark logs), we saw that values for *IBM GPFS* were much greater than those for *COFS*: about 20 seconds vs. less than 2 seconds for 16 nodes and 4 GiB

of aggregated data. According to the *IOR* code, this means that *COFS* had all the individual processes writing data (i.e. utilizing the maximum possible bandwidth) 2 seconds after the execution start; on the contrary, the last *IBM GPFS* process did not start writing until 20 seconds after the execution start, so there was a slight serialization of the writes from the different nodes.

The effect is more remarkable for smaller data sizes because the total transfer time is smaller (about 15 seconds for 1 GiB) and the impact of serializing the individual data writes is higher (there are fewer chances of overlapping data transfers from the different nodes and thus, there are fewer chances of fully utilizing the available bandwidth).

4.4.2 Metadata performance on GPFS at MareNostrum

The performance of applications observed in the MareNostrum supercomputer was one of the motivations of the work developed in this thesis. Parallel applications on large scale parallel systems like the MareNostrum expect to be able to perform I/O simultaneously from all the nodes as they would do in a single node (i.e. efficiently, in parallel and keeping the consistency).

In this subsection, we summarize the experiments performed on the MareNostrum supercomputer to validate the virtualization results obtained in the Nord test-bed cluster. As mentioned in Subsection 4.2.3, the measurements for `stat` and `open` in MareNostrum closely resemble the performance of `utime`; therefore, we will focus on presenting the results for `utime` and `create`.

The potential impact of the interception technology (FUSE) on data transfers is equivalent to the one studied for Nord (Subsection 4.4.1); therefore, we will not include data benchmark results (*IOR*) in the discussion, and will focus on the metadata benchmark (*Metarates*).

Virtualization results

We deployed the *COFS* prototype on the *IBM GPFS* file system of the MareNostrum supercomputer to verify that it could mitigate some of the metadata performance issues and that the benefits obtained where significant enough to compensate the cost of adding an extra layer on top of the file system. This deployment was important to check that the trends observed in the Nord test-bed were still valid for a larger-scale system with faster hardware like the MareNostrum.

A difference of this particular deployment of *COFS* is that, at MareNostrum, it uses IP over Myrinet for communicating the *Metadata Driver* with the remote *Metadata Service* and clients. The reason for using Myrinet for *COFS* instead of the 1 Gbps Ethernet used by *IBM GPFS* was to compensate for a network topology favouring the official file system servers. The forty (40) *IBM GPFS* servers (14 of them dedicated to the file system being tested) had dedicated 1 Gbps links to the main switch; on the contrary the *COFS Metadata Service* run on a standard computation node sharing 1 Gbps link

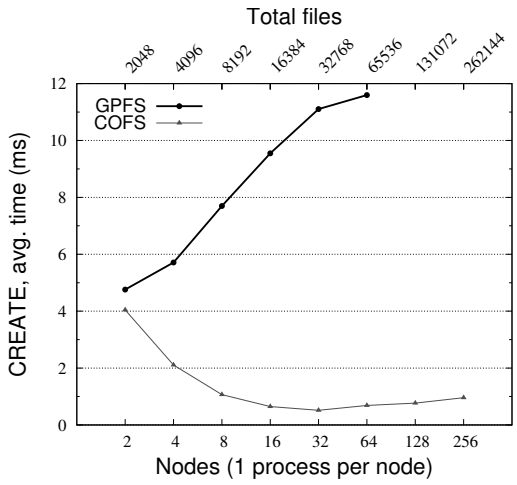

Figure 4.15: Parallel `create` improvements of *COFS* over *IBM GPFS*, using 1,024 files per node in a shared directory at MareNostrum, with 1 process per node.

with other thirteen blades; as the tests were run while the whole cluster was in production executing other applications, we noticed that the results varied significantly depending on the blade being allocated for the metadata server and the activity of its neighbouring blades. On the contrary, the different topology of the Myrinet network mostly prevented this variability. Though the Myrinet protocols are theoretically able to provide less latency and greater bandwidth than the 1 Gbps network, using the IP stack on top of it adds enough overhead to compensate the differences: tests performed with the cluster in dedicated mode revealed that there were no substantial differences between the performance of the 1 Gbps Ethernet network and the IP over Myrinet network regarding metadata traffic.

Figure 4.15 shows the benefits of the *COFS Metadata Virtualization Layer*, exporting a single shared directory virtual view to the application level while maintaining separate directories in the actual layout of files in the underlying *IBM GPFS* file system. By redistributing the entries into smaller low level directories, *COFS* allows *IBM GPFS* to fully exploit its parallel capacity by converting a shared parallel workload into multiple local sections that do not require global synchronization.

The improvement translates into a reduction of the average `create` time from more than 10 ms in 16 nodes for *IBM GPFS* to about 1 ms when using *COFS* over *IBM GPFS*. The numbers for pure *IBM GPFS* were limited to 64 nodes because beyond that point the system was suffering severe performance problems when running the benchmarks (that issue was not present when using *COFS* over *IBM GPFS*).

Figure 4.16 shows the average time for `utime` requests. We can see that, in this case, the overhead introduced by *COFS Metadata Virtualization Layer* is noticeable for a small number of nodes, but it converges with pure *IBM GPFS* results at 16 nodes. Beyond that point, we can observe that *IBM GPFS* rapidly degrades its performance as we increase the number of participating nodes; on the contrary, *COFS* is able to compensate and eliminates the degradation, allowing the system to scale better to larger amounts of nodes.

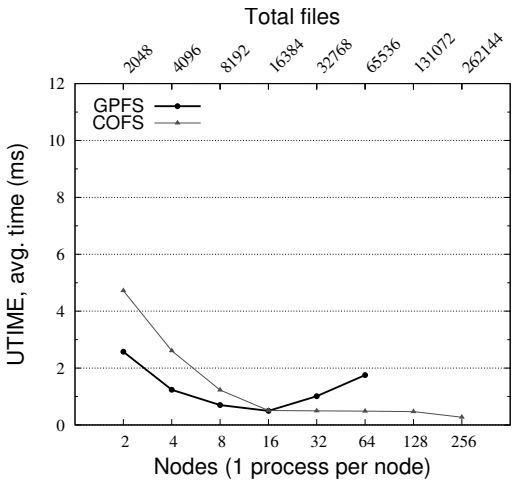

Figure 4.16: Parallel `utime` scalability using *COFS* over *IBM GPFS*, using 1,024 files per node in a shared directories at MareNostrum, with 1 process per node.

COFS takes advantage of the reduced operation time when *COFS* delegation is active by redistributing the files in smaller directories and avoiding parallel accesses to such directories when possible, thus reducing the conflicts between different nodes and allowing *IBM GPFS* to fully exploit the parallelism.

In summary, our measurements show that the *Metadata Virtualization Layer* provided by the *COFS* framework can drastically boost the base *IBM GPFS* on large-scale systems like the MareNostrum supercomputer. *COFS* allows to obtain speed-up factors up to 10 for file creations on parallel environments (as shown in Figure 4.15). For the rest of metadata operations, performance is also boosted for large numbers of nodes, and performance degradation due to conflicting parallel accesses is reduced.

4.4.3 Metadata performance on Lustre at INTI cluster

After studying the metadata behaviour of *IBM GPFS* and how the methods implemented in *COFS* were able to mitigate its performance problems, the next step was checking if the same issues were present in other file systems and, in such case, if the solutions from *COFS* were also able to alleviate them. This study was carried out as part of the task 12.4 of the PRACE-2IP European project, which had as objective to identify potential scalability issues in the *Lustre* file system in very large-scale environments, and to propose solutions in the case such issues were found.

Lustre [Braam 2007] is a parallel file system able to offer a POSIX compliant interface and based on three types of components: the clients (nodes accessing the file system), the storage servers where the data resides (based on OSDs) and a metadata server responsible for name space, access rights and consistency management.

One of the key characteristics of *Lustre* regarding metadata management is that it relies on a single metadata server (possibly replicated for failover replacement) to handle all metadata. This approach simplifies consistency management (compared to a fully distributed locking mechanism for metadata management — as in *IBM GPFS*).

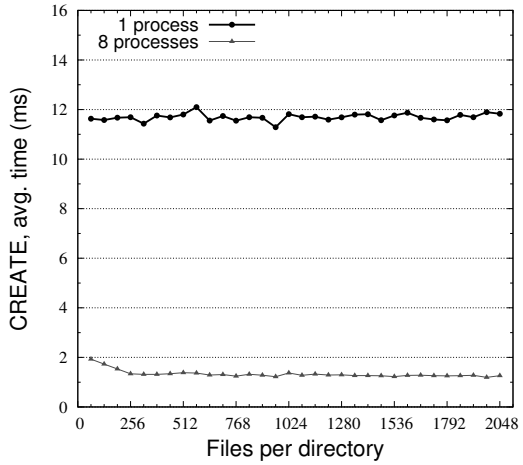

Figure 4.17: Cost of `create` on a shared directory in a single *Lustre* node, using 1 and 8 processors at INTI.

The goal of the evaluation was, on one side, to check if the potential scalability issues in *Lustre* were analogous to those found in *IBM GPFS*, or if they were of a different nature. On the other side, we wanted to verify if the *COFS* framework could be used to mitigate the potential issues, or if alternative techniques were needed.

The test-bed was deployed in the INTI cluster at CEA, consisting of 128 Bullx Inca Nehalem-based blade nodes (with 8 X5560 processors at 2.80 GHz). The InfiniBand network topology was a fat-tree with 9 Mellanox InfiniBand QDR top-switches (with 36 ports each). The storage system (Xyratex Exascale I/O prototype) was a ClusterStor 6000 including 9 SSUs with 18 CS6000 controllers (SandyBridge-based, FDR capable), a Cluster Management Node based on a quad-redundant node with shared, high-performance MDT storage, 2 Mellanox QDR switches (36 ports each) and 1 Gbps Ethernet Network switch (24 ports). The *Lustre* file system was the latest (at the time of the tests) Xyratex 2.1 prototype branch.

The focus of study in the PRACE-2IP file system prototype was the behaviour of metadata; consequently, we used the *Metarates* benchmark to obtain the measurements.

The following subsections summarize the observations of metadata behaviour in the *Lustre* prototype and the results of applying the *COFS* infrastructure on top of it.

Metadata issues in Lustre

The *Metarates* benchmark has been executed in the *Lustre* prototype in order to compare the results with the observations in *IBM GPFS*. One of the goals is, given the different strategies used by both systems, to be able to obtain thorough information about the causes of metadata performance issues and which are the best ways to neutralize them.

The first step in the study of metadata behaviour in *Lustre* was measuring the file system performance on a single node, with the goal of spotting elements affecting

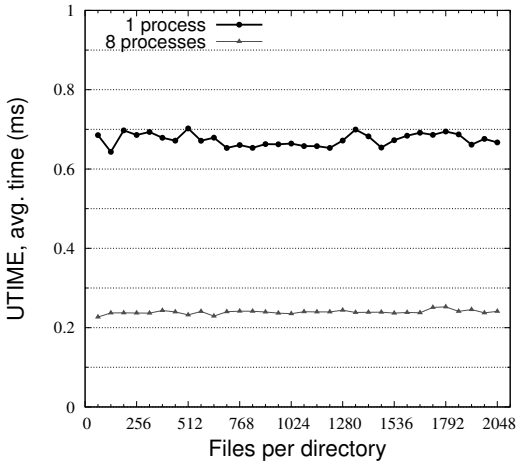

Figure 4.18: Cost of `utime` on a shared directory in a single *Lustre* node, using 1 and 8 processors at INTI.

the performance but unrelated to parallel access patterns. In particular, there was an interest in assessing the impact of the number of files on a single directory (which notably affected *IBM GPFS*).

Figure 4.17 shows the average time spent in a file `create` operation in a *Lustre* client for increasing sizes of the working directory. There are two important things to observe. First, the pattern of the lines is essentially flat (except when using 8 processors in directories smaller than 256 files, where the ratio between overhead and actual work is higher), indicating that the size of a directory is not, per se, an important factor regarding performance. The second is that using multiple threads inside a node effectively increases the performance.

The same behaviour can be observed for the `utime` system call (Figure 4.18), and the results for the `stat` system call are very similar (though slightly faster, there are no differences depending on the number of files in the directory).

It is worth mentioning that this behaviour differs significantly from the observations done in the *IBM GPFS* file system, where small directories used from a single node could benefit from special optimizations, resulting in a highly reduced operation time compared to larger directories (Figure 4.3). Oppositely, there is no equivalent optimization in *Lustre*.

The next step in the study of *Lustre* metadata behaviour is to observe the influence of having multiple client nodes using the file system at the same time. We compared the results in a single node with the results of executing the benchmark on 2 and 4 nodes. For the different series, each node uses the same number of distinct files in order to avoid the effects of different work versus overhead ratios (therefore, the total number of files varies according the number of participating nodes).

The results for the `create` operation obtained using 1 and 8 processes per node are shown in Figure 4.19. For this experiment, all files were created in a single shared directory. The first observation is that *Lustre* seems able to exploit efficiently the internal parallelism, and the *Lustre* metadata server does not seem to behave too

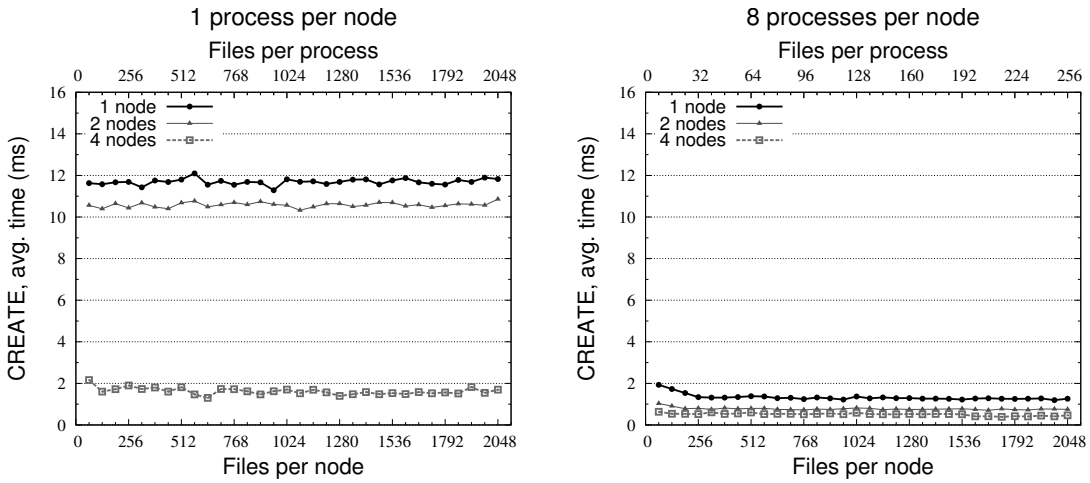

Figure 4.19: Cost of parallel `create` in a shared directory at INTI from multiple *Lustre* nodes, using 1 and 8 processes per node.

differently if parallel requests come from the same node or from different nodes (though increasing the number of nodes seems to be slightly more efficient than simply increasing the number of threads in a node — for example, the performance of using 4 nodes with a single process is not far from using 8 processes in a single node).

Nevertheless, it is worth mentioning the lack of improvement in performance when moving from one to two nodes: *Lustre* seems to be able to exploit effectively the parallelism only beyond 4 processes. A possible explanation for this effect would be that the overhead of maintaining the consistency when multiple clients use the file system is relatively high, and can only be compensated when there is enough parallelism to be exploited.

The same experiments were executed creating the files in separate directories for each process, to avoid intra-directory conflicts. The results obtained showed the same pattern as Figure 4.19, with just slightly better performance for 8 or more processes.

Other metadata operations such as `utime` and `stat` show a different pattern and reveal a potential scalability issue. Figure 4.20 shows the average cost of the `utime` system call using up to 4 nodes with 1 and 8 processes per node.

The first thing to note is that we do not see the lack of performance increase when moving from 1 to 2 processes (which we could observe for file `create` in Figure 4.19). That means that the overhead in `create` is probably due to either the directory entry creation or the actual data file creation itself (tasks that are exclusively related to the file creation operation), rather than accessing or modifying a file's metadata.

The second interesting observation reveals a potential scalability issue. Indeed, we can observe that executing the benchmark with 4 processes offers a slightly better average performance than using 8 processes. This effect is substantially bigger for the

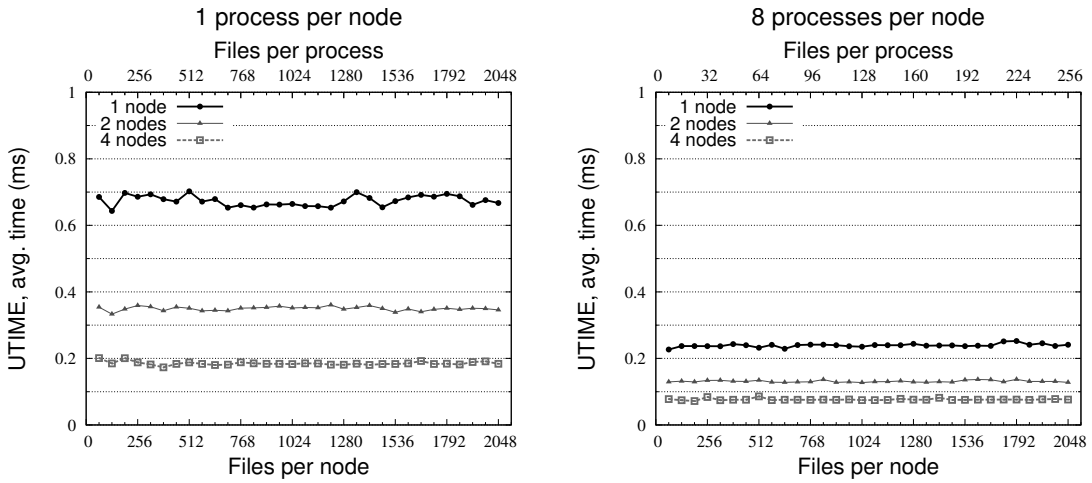

Figure 4.20: Cost of parallel `utime` in a shared directory at INTI from multiple *Lustre* nodes, using 1 and 8 processes per node.

`stat` system call, where the system obtains an average cost of 0.12 ms when using 4 processes, and then the performance drops to an average cost of 0.21 ms when using 8 processes.

After doing additional measurements, we determined that the anomaly was mainly due to two different causes. First, `utime` and `stat` seem to benefit more from multi-node parallelization than from single-node, multi-process parallelization; second, parallel access to a shared directory generates contention, which tends to increase with the number of processes accessing the shared resource.

For `utime`, the average cost of the system call when using 8 processes in a single node is about 0.24 ms, and then it drops to 0.13 ms when using 2 nodes with 4 processes each, and finally reaches 0.1 ms for 8 nodes with a single process each. In the case of `stat`, the average costs of the system call for 8 processes are 0.21 ms, 0.11 ms and 0.06 ms when the processes are distributed into 1, 4 and 8 nodes respectively. A possible reason for this effect is that the actual cost of the operations is very small and, therefore, the impact of network aspects like latency and bandwidth (which are potentially improved by using more nodes) is more significant. Additionally, communication with the *Lustre* developers revealed that potentially parallel `utime` operations can be serialized inside a client node to facilitate metadata journaling procedures. This effect is not significant in the case of `create`, which has an overall cost more than an order of magnitude larger.

It is also expected that the effect of increasing the number of nodes versus intra-node parallelism will only be noticeable for a relatively small number of nodes, until either the network or the server is saturated. This point was confirmed with additional measurements showing that using additional threads in each node had no improvement in performance when using 32 nodes or more.

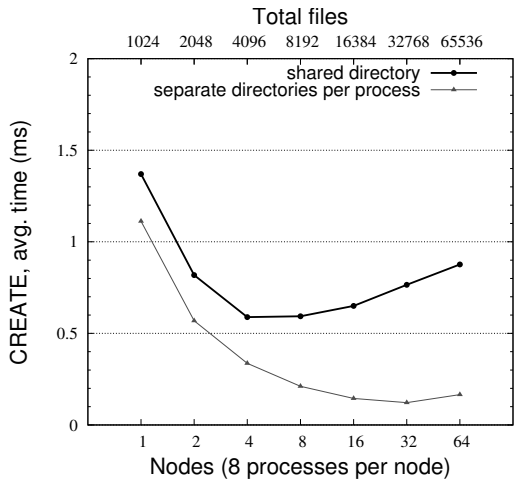

Figure 4.21: Shared directory contention on *Lustre* `create` operations at INTI, creating 1,024 files per node and using 8 processes per node (128 files per process).

The other factor impacting the performance is the contention caused by parallel access to a shared directory. In order to assess this effect, we re-executed the benchmarks making each process work in a separate directory.

Figure 4.21 shows the impact in performance of file `create` operations in a shared directory versus creating the files in unique separate directories, and how it increases with the number of nodes involved (each node uses 8 processes).

On the contrary, the differences between using shared and non-shared directories tend to disappear for `utime` and `stat` operations, and the cost of the operations converge to a base line for 32 nodes and beyond (see Figure 4.22). A possible explanation compatible with this behaviour is that other costs (e.g. network-related costs) become dominant over the actual operation time, so that differences are no longer relevant.

It is interesting to see that the cost of `utime` in separate directories is also close to the shared directory costs. Conversations with *Lustre* developers suggested that this drop in performance for a small number of nodes could be caused by a lock in the *Lustre* clients, used to serialize certain intra-node metadata operations with the objective of guaranteeing proper replayability of the metadata journal.

There is a metadata-related operation which does not seem to be affected by parallel access to shared directory versus non-shared directories: `open`. We have not observed any noticeable differences when files are open by different processes in a single shared directory compared with opening the files in unique directories. This is probably due to the fact that other metadata operations (which do show differences) require some kind of synchronization when accessed in parallel (`utime` modifies the access times and must be kept consistent, `stat` requires synchronization of, at least, data sizes, and `create` has to keep the shared directory consistent); on the contrary, `open` does not require, per se, to perform any type of synchronization (apart from the minimum required to ensure that the file being open exists and can be accessed).

Another important difference between `open` and the previously studied metadata operations is that, while the cost of other operations tends to stabilize as we increase

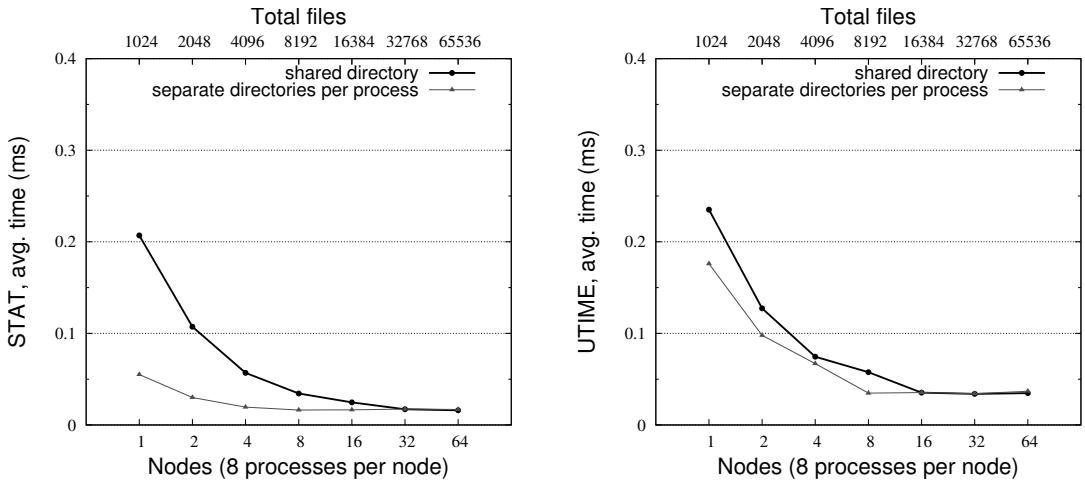

Figure 4.22: Shared directory contention on *Lustre* `stat` and `utime` operations at INTI, accessing 1,024 files per node and using 8 processes per node (128 files per process).

the number of nodes, `open` shows a clear degradation when using more than a few nodes, which is aggravated when using multiple threads per node.

Figure 4.23 shows the loss of performance of the `open` operations when the number of nodes is increased. The figure shows the results for parallel `open` operations in a shared directory; there are no noticeable differences in behaviour and performance when operations are performed in separate directories dedicated to each process.

In summary, we have observed that parallel `create` operations on shared directories suffer from overhead with respect to parallel creations in separate directories per process (though the overhead is less important than in *IBM GPFS*).

Parallel operations in *Lustre* scale quite well up to an optimum value about 16–32 nodes (depending on the operation and conditions) but show hints of performance degradation beyond that point. The goal should be avoiding the performance degradation and even continue the performance improvement with more nodes.

Finally, the sequential behaviour of *Lustre* regarding metadata operations is not significantly affected by the number of files in the directory (unlike *IBM GPFS*), and it does not seem to have specific optimizations to exploit the sequential case (e.g. via control delegation to the client node).

Virtualization results

After studying the behaviour of metadata in *Lustre*, we deployed the *COFS* prototype to check if it could mitigate some of the metadata performance issues we observed, especially the loss of performance when creating files in a shared directory.

Being a drop-in component with only generic dependencies, the prototype initially

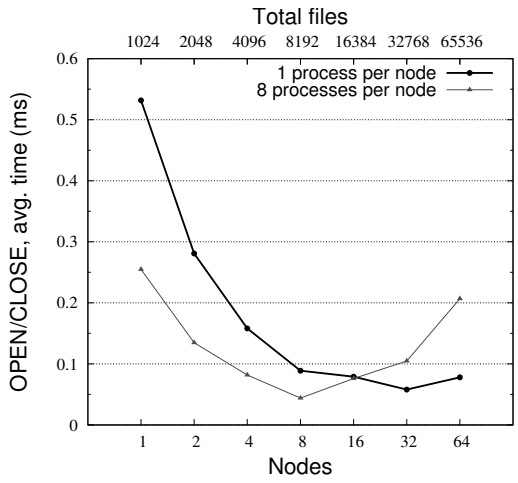

Figure 4.23: *Lustre* `open` operation scalability in a shared directory at INTI with 1,024 files per node, using 1 and 8 processes per node.

developed for *IBM GPFS* required no modifications, apart from usual rebuilding of binaries. At the CEA test-bed, the *COFS Metadata Service* was run in a normal computation node of the cluster. Even if this implied some performance handicaps compared with *Lustre* metadata server setup, we expected to gather enough information to assess the feasibility of *COFS* mechanisms to improve certain performance aspects of *Lustre*.

COFS Metadata Virtualization Layer offers a file system name space decoupled from the underlying file system (*Lustre*, in this case), and it internally reorganizes the location of the files without altering the user view of the directory hierarchy; specifically, *COFS* tries to place files in such a way that the underlying file system can obtain a good performance even if the original file distribution would cause parallel access conflicts and, consequently, performance degradation.

An application working on top of *COFS* will send file system requests to *COFS Metadata Service* which, eventually, may forward them to the underlying *Lustre* file system. This involves an overhead with respect to a native *Lustre* operation, which can be compensated if the gains of the file distribution made by *COFS* are higher than the cost of the extra work performed by the *COFS Metadata Virtualization Layer*.

According to the results of the study of *Lustre* metadata behaviour in the previous subsection, one of the candidate situations where *COFS* can mitigate *Lustre*'s performance degradation is the parallel creation of files in a shared directory.

Figure 4.24 shows the results of using *COFS* over *Lustre* for parallel file `create` operations on a shared directory. We can see that, for a small number of nodes, the overhead of *COFS* is higher than the benefits of file reorganization, but beyond 8 nodes, the gains compensate the overhead, preventing further degradation of the performance.

It is worth mentioning that, in the case of *IBM GPFS*, *COFS* hid the overhead by exploiting the cases for which *IBM GPFS* was highly optimized (e.g. small directories only used locally). According to the observations in the previous subsection, *Lustre* does not appear to have such cases with drastic optimizations and, on the contrary,

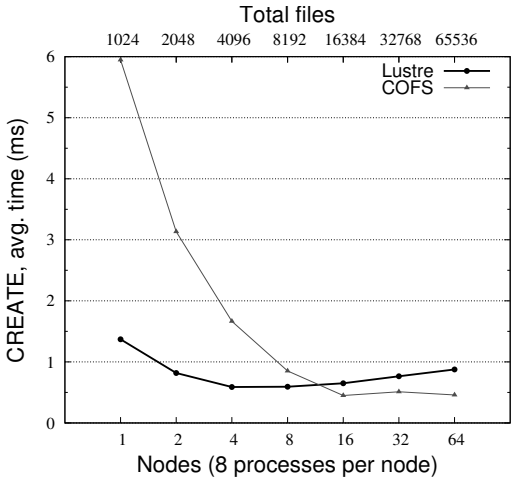

Figure 4.24: Parallel `create` performance on shared directories using *COFS* over *Lustre* at INTI, with 1,024 files per node and 8 processes per node (128 files per process).

performance changes show smooth and homogeneous trends. Therefore, there is less room to hide the virtualization overhead.

Even if the benefits of *COFS* in the case of parallel creation are smaller than they were for *IBM GPFS*, there is still an indication that the way *COFS* manages the directories is adequate for large-scale parallel access, and it could be considered as a possible candidate to be integrated inside *Lustre* itself in order to avoid parallel creation degradation, without the need to pay the overhead of an external layer.

Another situation in which we have observed an improvement due to using *COFS* over *Lustre* is file `open` operations. Again, native *Lustre* showed performance degradation for large numbers of clients, though it was not related to accessing a shared directory, as it also appeared for non-shared directories.

Figure 4.25 shows the `open` behaviour of *COFS* over *Lustre* on shared directories. Again, the overhead introduced by *COFS* is higher than the benefits, for small amounts of nodes, but it is compensated when the number of nodes increases, mitigating the *Lustre* degradation. The low overhead of *COFS* for the single-node case is due to some specific optimizations (essentially based on aggressive caching) introduced to be competitive with *IBM GPFS* control delegation mechanisms for isolated, single-node, file system operations.

The `open` case is interesting because it is less depending on the underlying file system metadata than other metadata operations like `utime` or `stat`, and it is almost completely handled at *COFS* level. This explains why *COFS* can show performance improvements, even when there was no obvious optimal situation observed in the study of *Lustre*.

The execution of the benchmarks for the remaining metadata operations (namely `utime` and `stat`) showed no further improvements derived from the use of *COFS* because either *Lustre* was already showing scalable good performance, or because the margins between optimal and non-optimal *Lustre* behaviour were too small to compensate for *COFS* overhead.

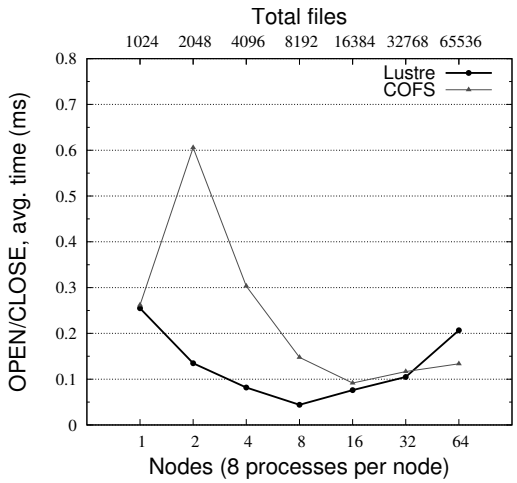

Figure 4.25: Parallel `open` performance on shared directories using *COFS* over *Lustre* at INTI, with 1,024 files per node and 8 processes per node (128 files per process).

Nevertheless, the results also indicate that *Lustre* may obtain benefits from integrating some of the directory management techniques used by *COFS* in order to improve situations with degraded performance, without paying the cost of an external layer.

4.4.4 Summary

From the experiments, we have learned some facts about file system behaviour. We have evaluated the effects that the number of files and clients have on the performance of metadata operations in both *IBM GPFS* and *Lustre*, specially when requests are made from multiple nodes to shared directories.

We have observed that parallel creations on shared directories suffer from overhead with respect to parallel creations in separate directories per process. The overhead is less significant in *Lustre* than in *IBM GPFS*, but still substantial. The behaviour of the other metadata operations varies depending on the file system: *IBM GPFS* shows a progressive degradation on shared directory behaviour when the number of participating nodes is increased, while *Lustre* performance seems to be stable. Nevertheless, *Lustre* reveals an important loss of performance of the `open` operation when increasing the number of nodes.

The number of entries in a directory affects the performance of *IBM GPFS*. The reason is that the system is highly optimized for small directories, reverting to 'normal' performance values for large directories. *Lustre* does not seem to have equivalent optimizations, and show a consistent behaviour across different directory sizes.

Decoupling the name space from the low-level file system layout is the mechanism used by *COFS* to mitigate metadata issues. The *COFS Metadata Virtualization Layer* offers large shared directory views while internally splitting them to take advantage of non-shared operations.

Using *COFS* over *IBM GPFS* drastically improves the performance of file `create` operations on shared directories, and prevents performance degradation of other

metadata operations at large scale. Its use over *Lustre* also results in improvements for both parallel `create` and `open` operations, preventing performance degradation for large numbers of nodes.

Regarding the rest of studied *Lustre* metadata operations (`stat` and `utime`), the overhead introduced by *COFS* appears to be too high to be used effectively. Nevertheless, as the techniques to handle directories and metadata seem to adequate for large scale systems, it could be considered to integrate them inside *Lustre* to take advantage of the benefits, without having to pay the cost introduced by the external virtualization layer.

4.5 Conclusion

In this chapter, we have discussed the issues of large-scale file systems used in HPC environments related to metadata management. We have studied the behaviour of such systems in actual large-scale computing clusters and implemented a prototype that uses file system metadata and name space virtualization to mitigate their issues. The evaluation has shown that an extra *Metadata Virtualization Layer* is an effective tool to solve some of the metadata-related performance issues on parallel file systems, and hints techniques that could be directly integrated into the file systems themselves.

After evaluating the behaviour of *IBM GPFS* and *Lustre* file systems, we have detected that parallel file creation on shared directories suffers from severe performance penalties on both systems beyond a few dozens of nodes. The use of the *COFS Metadata Virtualization Layer* shows that decoupling the name space view from the actual layout in the underlying file system allows to avoid contention bottlenecks, improving the performance and scalability of the system.

COFS operates by detaching the user view from the underlying layout of the file system. By doing so, it is able to transparently convert non-optimized cases (from the file system point of view) into optimized ones, thus getting all the benefits that the underlying file system can offer without asking users to change their behaviour.

The experiments also show that the centralized *Metadata Service* approach used by *COFS* does not introduce substantial scalability issues, due to the fact that metadata is relatively small and metadata operations are usually short.

The scale at which file system metadata should be managed in a distributed way has to be carefully considered, in order to avoid the cost of distribution and synchronization being higher than the potential *Metadata Service* contention it tries to solve. In fact, we could observe that *Lustre* metadata management (which uses a centralized approach) appeared to scale better than *IBM GPFS* (which aggressively distributes metadata across multiple servers).

In those situations where the overhead of the additional layer introduced by *COFS* is too high to obtain significant benefits, the directory and metadata management algorithms used by *COFS* could be integrated inside the target file system (e.g. *IBM GPFS* or *Lustre*) to obtain the benefits without the overhead of an additional layer.

CHAPTER 5

Name Space Virtualization to Improve Usability

Contents

5.1 Introduction

Cloud-based storage services are widely available. Nowadays, these services are provided to final users by means of a number of commercial products (*Box* [Box 2013], *Dropbox* [Dro 2013], *JustCloud* [Jus 2013] or *SugarSync* [Sug 2013] are just a few examples). Some popular features of such products are data synchronization among devices, data sharing among users, and backup management.

From a technical perspective, some of the challenges that these applications face are keeping the integrity and consistency of data, and optimizing the transfer of data between the client system and the cloud. Nevertheless, for the final user, such technical aspects are things that happen behind the scenes and, once a certain quality in the service is achieved, there is another aspect that becomes important: usability.

In terms of usability, cloud-based backup and synchronization services show a lack of flexibility. The selection of files to consign to the cloud is usually rigid and coarse-grained. For example, it is typically not possible to select a single file in an arbitrary location of the file system for synchronization or sharing: most systems (such as the examples mentioned above) force the synchronization of the whole directory where the file resides (some of them can generate a special url that allows obtaining a copy of a 'shared' file, but this is not a truly collaborative solution because the copy obtained is detached from the original). The result is that the user is forced to select a

limited number of directories to put in the files that will be under the cloud control and, at most, select some specific files in them. Moving a file outside one of such directories may cause the file to fall out the cloud.

There is a reason behind the rigidity of file organization for cloud-based storage services. The interaction between a local system and the cloud-based storage service is handled by local components: these components are programs or daemons that run in the local computer and act as local agents that communicate with the remote cloud service. These local components usually need to determine if a file has been modified and changes should be synchronized; for that, the local components have to monitor and track changes in the local file system. Modern file systems offer event-driven interfaces to obtain this information without incurring in the high overhead that a continuous polling from applications would cause. However, these interfaces vary in terms of granularity and the details provided: the minimum common service just indicates if changes have occurred in a directory, but it does not necessarily inform about which files have been modified, and it is up to the application to collect and store the necessary state to identify the changes.

For example, the *FSEvents* API in *Apple Mac OS X* [FSE 2012] sends a notification when files in a selected directory (and subdirectories) have changed, but it does not indicate which ones; the mswindows API function `ReadDirectoryChangesW` informs about specific changes in a given directory (and subdirectories), though information may be dropped if the notification buffer overflows [MSDN 2013]; finally, the *Linux inotify* interface provides details about changes related to an arbitrary file or directory (but not to subdirectories) [Love 2005]. In all cases, the burden of maintaining handles to track the changes and filtering out the uninteresting events falls on the monitoring application.

As a consequence, the local components of cloud-based services use individual directories as the basic unit of operation, and try to limit their number as a way to minimize the state required for effectively tracking changes. Unfortunately, this also reduces the ability of the user to organize the files for the cloud in a flexible way, having to abide by the constraints imposed by the cloud service.

One of the contributions in this thesis consists in increasing the flexibility in the organization of files to be incorporated to the cloud. To this end, we propose decoupling the view of the file system name space from the actual underlying directory hierarchy by means of the virtualization techniques discussed in this work. This virtualization of the name space allows offering different simultaneous views to different users or applications. A direct application of this mechanism permits a local component of a cloud-based service to keep relevant files concentrated in a small, easy to track set of directories while, at the same time, the user sees files organized according to her preferences and needs.

One of the advantages of using a *Metadata Virtualization Layer* to offer specialized name spaces is that it does not require changes to either the file system change tracking interfaces or the local components of the cloud-based storage services. Yet, it provides the user with increased flexibility.

5.2 Approach

In this section, we present some concepts related to the virtualization of the name space, and then we discuss the details of the structure of the specific *Metadata Virtualization Layer* needed for the current purposes, as well as some of the design decisions.

5.2.1 Decoupling virtual name space from physical layout

Cloud-based storage systems rely on interfaces provided by the underlying file system to track relevant changes in monitored local directories; but, as mentioned in Section 5.1, the available mechanisms make tracking the whole directory tree impractical; consequently, relevant files are usually confined into a specific set of folders.

The name space virtualization model allows to overcome these limitations without changing the way cloud components work. Particularly, we can let the local components of the cloud-based service interact directly with the native file system, organizing the relevant files in the usual way: i.e. concentrating them in a set of selected folders. We will refer to these folders as 'cloudified' directories. In order to simplify the examples, the cases that we will discuss will use a single cloudified directory; nevertheless, all techniques described are equally valid for a set of them.

On the other hand, the user (and, by extension, the rest of applications running in the local computer) should be able to organize the files without the restrictions imposed by cloud-based services. This means that we need to provide an alternate view of the file system layout which may differ, at least in some points, from the physical name space (for example, synchronized files will appear anywhere in the directory hierarchy instead of being concentrated in the 'cloudified' physical directory). We will provide this alternate view through a virtual name space.

The selection of files to be stored in the cloud and the additional information associated with them (e.g. if they can be shared and, if so, with whom) can be set by different mechanisms. For instance, the user may introduce her preferences as extended attributes of the file (it is possible to transparently hook functions into the extended attribute interface, and trigger a communication with the cloud service components to set particular configuration options for a file whenever a specially formatted value is set for a particular attribute); the advantage of this method is that it is integrated in the standard file system interface. Alternatively, it is also possible to use an ad-hoc application to indicate the desired setup to the cloud service.

Auxiliary data about the directory hierarchy in a particular name space may also be stored in the cloud. This includes, for example, the relation between a particular data file in the cloud and its position in a virtual name space, so that cloudified files can be represented according to a virtual view when accessed from outside the local system (e.g. via a web service).

In our basic model, the virtual name space will expose part of the physical name space enriched with virtual files. We will use the expression 'anchor directory' to

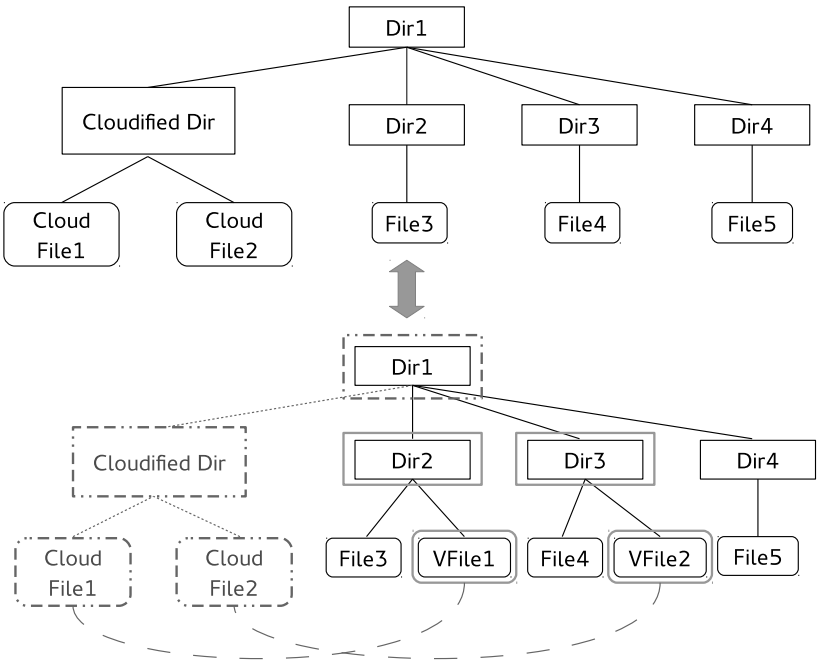

Figure 5.1: Relation between a physical (top) and a virtual (bottom) name space.

indicate a directory in the virtual name space exposing a modified view of the contents of the physical directory with the same path, adding virtual files associated to that directory and hiding physical files that should not be available in the virtual view.

Figure 5.1 shows the relationship between the physical name space and a virtual name space. The physical name space has a dedicated cloudified directory containing two files to be managed by a cloud-based service; this is the name space that the local components of the cloud service will see. On the other hand, the rest of applications will see the virtual name space at the bottom of the figure. Directories "`Dir2`" and "`Dir3`" act as anchor directories and they expose the contents of the corresponding physical directories plus two virtual files ("`VFile1`" and "`VFile2`") which correspond to physical files "`CloudFile1`" and "`CloudFile2`". Regarding "`Dir4`", there are no differences between the physical and virtual name spaces.

Optionally, the cloudified directory in Figure 5.1 could be hidden from the virtual name space view, preventing applications from directly accessing and manipulating the cloud-based storage service data. This adds an extra level of protection to the cloudified directory and allows using it to store administrative information for the name space virtualization mechanism (for example, the cloud service itself could be used to synchronize information about the mapping of the cloudified physical files into the virtual name space for a particular device, as mentioned before).

The binding between the virtual entries and the physical files is similar to hard links in POSIX file systems, allowing multiple entries to reference the same data file;

nevertheless, their semantics differ in two aspects. First, when a user removes a file in the virtual name space, the corresponding entry is also to be removed from the cloudified directory in the physical name space - and vice versa; so, the removal of an entry in a name space requires the removal of the entries referring to the same file from the other name spaces (conventional hard links must be removed individually, and finding all hard links to a given file is a costly operation). Second, the attributes associated to entries in different name spaces may be different, even when they refer to the same data file: for example, the file access permissions could allow only read-write access for the owner in the virtual name space and permit read-only access for a synchronization application accessing the physical name space (oppositely, hard link entries would share all attributes).

The relationship between entries in the physical and virtual name spaces is transparent to applications: the *Metadata Virtualization Layer* is responsible for keeping the necessary information to track the usage of name spaces and for maintaining their consistency.

5.2.2 Structure of the virtualization layer

Applications perceive the file system name space through path resolution (e.g. when checking if a file exists in a given path and can be open) and directory listings. Inside the file system, the path resolution mechanism converts a given path into a reference to the actual piece of information; then, that reference can be used in subsequent operations to access and manipulate the data. Additionally, directory listings also collaborate to expose a picture of the name space by providing the names of the objects located in a particular node of the directory hierarchy. Consequently, our mechanism to provide a coherent virtual name space to applications focuses on providing support for those two operations, while relying on the underlying file system for the rest of storage management aspects (such as storing and retrieving file contents).

Figure 5.2 outlines the architecture of our *Metadata Virtualization Layer*, which closely follows the outline depicted in Section 3.3. Applications interact with the storage system using the standard file system interface. This way, they are unaware of the presence of the *Metadata Virtualization Layer*: usual access operations such as `open`, `read`, `write` and `close` files are available, no matter if the target is a physical or a virtual object. Likewise, operations more closely related to the name space organization (such as `create`, `rename` or `unlink`) are identical on both anchor and physical directories.

When a request from an application crosses the file system interface, it is captured by the *Interception Engine*; then, the *Metadata Manager* decides what to do with the request. Essentially, the *Name Space Manager* detects whether the request is related to a virtual entry; if so, the system uses information stored as virtual attributes to fulfil the petition. These virtual attributes, handled by the *Attributes Manager*, include pieces of data such as the corresponding path in the physical name space or the access permissions to be applied when accessing the target object through a particular virtual

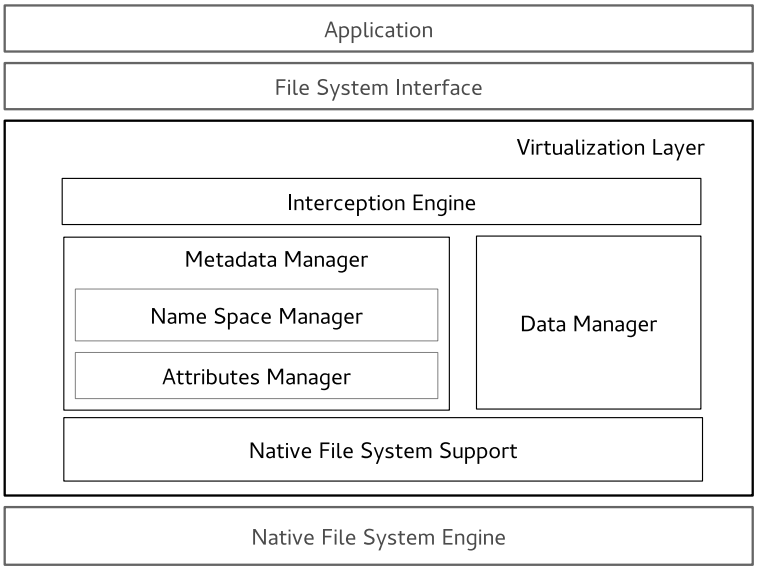

Figure 5.2: Architecture of the *Metadata Virtualization Layer*.

name space. On the contrary, if the original target is a physical object, the request is forwarded to the underlying file system.

The *Data Manager* is responsible for organizing the storage of data in the underlying file system (for instance, when a new virtual file is created, this module determines where the corresponding physical file should be located according to its cloud-related properties — e.g. if it has to be synchronized or shared). Additionally, the *Data Manager* may monitor the cloudified physical directory in order to detect new files and, if necessary, to incorporate them to the virtual name space (this situation occurs, for example, when a synchronized file is created in a remote device).

The interaction of the *Metadata Virtualization Layer* with the underlying file system is done through the *Native File System Support* module. The mission of this module is to deal with the intrinsics of a particular file system and maintain the necessary housekeeping information to allow a correct and consistent access to the actual data.

5.2.3 Design decisions

Kernel vs. user-space

Following the reasoning in Subsection 3.5.9 and the experience of the implementations in Chapter 4, we have decided to make the interception mechanism kernel-based. Apart from facilitating a transparent deployment of the *Metadata Virtualization Layer* from the application's perspective, the kernel also protects the interception mechanism, preventing applications from bypassing it.

A consequence of intercepting requests at the kernel level is that the interface with the underlying file system is a low level, system-dependent one; this means that each

different platform needs a specific implementation of the *Native File System Support* module. Nevertheless, the greater control and the availability of detailed information about the operation compensate the difficulties of handling the interception and the low level file system interface at the kernel level.

Oppositely, the *Metadata Manager* and the *Data Manager* are implemented in user-space. Here, the main reason was the ease of implementation: standard libraries and tools that can be used at user-space are not always available at the kernel level, and encapsulating the complexities of the name space virtualization logic in a standard process also protects and improves the stability and security of the underlying operating system.

Metadata representation

The attributes of virtual objects (files and directories) are stored in database tables. Each object has a unique identifier which is independent of its path. The tables use that identifier as a key, and store relevant information about the object. In particular, for a virtual file, one of the tables stores the path of the corresponding physical file. Ancillary tables contain other pieces of information depending on the functionalities required (for example, additional access permissions information).

The *Name Space Manager* also uses a table to store virtual directory entries. A virtual entry record contains the parent directory identifier and the name inside the directory as the key, as well as the identifier of the virtual object itself and its type as values. This allows the representation of virtual directory hierarchies.

Cross-name space semantics

Another relevant design decision involves defining the semantics of cross-name space operations, such as the behaviour of a `rename` request when the source and the target paths may belong to different name spaces. For the implementation of the prototypes, we tried to take a 'least-surprise approach' from the user's expectations.

First, we consider that the target of a virtual file `rename` is always a virtual file. This fulfils one of the main purposes of the name space virtualization mechanism: to improve the flexibility of cloud-based storage services. In our context, a virtual file represents a file under the control of the cloud (e.g. a synchronized file); therefore, even if we move that file around, we still want that file to be in the cloud (which in practice means that it has to be a virtual file with its physical counterpart stored in the cloudified physical directory). In the simple case, both the parent directories of the source and the target entries are anchor directories: then, `rename` just consists in adjusting the virtual name space information in the *Metadata Manager* (the mapping to the physical file in the cloudified directory remains unmodified). When the target's parent directory is a physical directory, it is automatically transformed into an anchor directory before the actual move takes place (as virtual files are always associated to anchor directories).

When renaming a physical file, the target will be converted to a virtual file if the target's parent directory has been configured to do so (i.e. if the user has selected the

whole directory to be synchronized or backed-up in the cloud). The file will also be virtualized if a virtual name space is being used and a virtual file with the same target path already exists (in this case, renaming means replacing the target file's contents with the data from the source file, and if the target were in the cloud, it is expected to remain there).

5.3 Implementation aspects

In this section, we explain some details about the implementation of prototypes of *Metadata Virtualization Layer*. We have chosen two different targets for the prototypes: the first is a *Microsoft Windows*-based system (*Microsoft Windows* 7 Enterprise Edition, using *Microsoft NTFS* as file system); the second is a *Linux*-based system (Ubuntu-Server 12.04 LTS distribution, using a POSIX file system — namely *Ext4*). With this target selection, we demonstrate that virtual name spaces are implementable in widely different platforms.

5.3.1 Common implementation aspects

Despite the semantics of POSIX and *Microsoft NTFS* being different, we have tried to make both implementations as similar as possible in order to facilitate code reuse across platforms. Naturally, the kernel-level code (including interception and the low-level file system support) must be system-specific but, as mentioned in Subsection 5.2.3, the majority of the name space virtualization logic is implemented in a user-level service, and most of it can share a common structure across platforms. From now on, we will refer to the set of components of the *Metadata Virtualization Layer* running in the user-space as the User-Level Service (or *ULS*, for short).

The *ULS* is divided into two layers: a platform-dependent layer transforms the requests captured by the system-dependent *Interception Engine* into system-independent requests; then, a platform-agnostic layer handles the high level logic and the data structures needed to provide the file system functionality.

The data structures representing the virtual name space and the objects in it (virtual files and anchor directories) are the same in the *Microsoft Windows* and *Linux* prototypes. Each virtual object is designated by an unique identifier, which is used as a key to access the database tables containing the necessary information to manage the object. The stored data includes file attributes (owner, permissions), internal management information (such as the actual path of the physical file corresponding to a virtual file), and specific support data for the cloud-based service (e.g. flags indicating if a virtual file or an anchor directory is being synchronized, shared or backed-up, as well as additional data related to this functionality). The virtual objects are linked together to form the virtual name space by means of an entry table, where the key is the composition of the parent directory's identifier and the name of the object inside that directory.

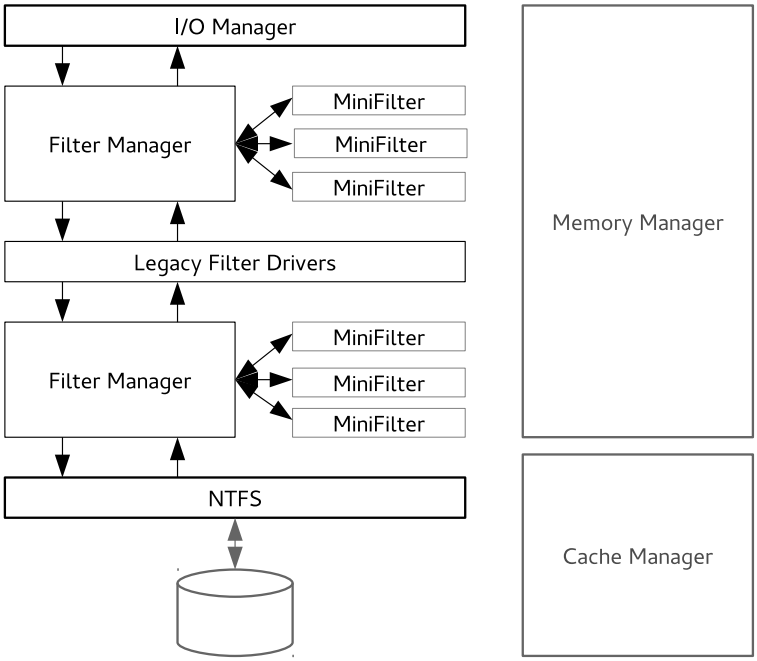

Figure 5.3: *Microsoft Windows* I/O subsystem components.

The differences in the file system interfaces are addressed either by decomposing the native requests into simpler operations with common functionality or, alternatively, by coalescing multiple low-level requests into a single *ULS* operation. For example, *Microsoft Windows* operations that take a full path argument are transformed into a series of individual path component look-ups to determine the internal target object, followed by the invocation of the platform-independent operation on such object; this makes it compatible with the *Linux* VFS interface, which separates the path resolution from the operation itself. Oppositely, a single conceptual operation such as a file removal consists of a sequence of three low-level requests in *Windows*: opening the file, setting a deletion flag, and then closing the file and eventually deleting the entry; in this case, the platform-dependent layer in the *ULS* keeps information to track the request sequence and invokes a single platform-independent removal operation when necessary.

5.3.2 Windows-specific implementation details

Infrastructure

Figure 5.3 represents the *Microsoft Windows* I/O stack. The *Microsoft Windows* I/O subsystem is packet-based: a request from an application to the I/O service is translated into a series of IRPs (Input/Output Request Packet) by the *I/O Manager* [Russinovich 2009]. IRPs are sent down the I/O stack for a particular device; such stack usually consists of a series of filter drivers, which may alter the informa-

tion contained in the IRP (and even generate new IRPs). The stack ends up in the file system driver, which communicates with the physical device driver and responds to the IRPs. The components in the I/O stack also interact closely with the *Memory Manager* and the *Cache Manager* [Nagar 2006].

Recent versions incorporate a special type of filter driver called *Filter Manager*. The *Filter Manager* acts as a place holder to plug *minifilters* [MSDN 2012]. The main difference between a *minifilter* and a filter driver is that a *minifilter* does not need to implement all IRP operations: it may register to receive a subset of the IRPs, and the missing functionality is provided by the *Filter Manager* itself. Several instances of the *Filter Manager* may appear at different depths of the I/O stack.

A *minifilter* may register both a 'pre-operation' and a 'post-operation' callbacks. The 'pre-operation' callback is invoked when the IRP is going down the I/O stack; the 'post-operation' callback is invoked when the response to the IRP is travelling up to the application, once the request has been processed by the lower levels. Additionally, a *minifilter* can declare the processing of an IRP completed (effectively preventing it from reaching the lower levels and the file system) or, on the contrary, it may generate additional IRPs. Special care has to be taken when using these options, since failing to receive expected IRPs, or receiving unexpected ones, may cause inconsistencies in lower level filters or the underlying file system itself.

The *Windows* prototype of our *Metadata Virtualization Layer* uses a *minifilter* to intercept IRPs related to file system operations. Due to the fact that a high-level operation can consist of a series of IRPs, this *minifilter* is responsible for aggregating information and keeping the necessary kernel state to build a service request for the *ULS*. The *minifilter* may also cache data from the *ULS* to be reused for several IRPs.

Name space mapping

One of the most important IRPs for the name space virtualization is `IRP_MJ_CREATE`. This IRP either creates a new file or directory, or opens it if the target already exists. It receives the path of the target object in the arguments; so, handling it correctly is critical in order to generate a consistent virtual view of the file system.

The interception *minifilter* collects the `IRP_MJ_CREATE` arguments and related information (such as the issuer process, security context) and sends them to the *ULS* during the 'pre-operation' step. The *ULS* then discriminates the name space under which the request must be processed. The simple case corresponds to the physical name space: the *ULS* does no additional processing, and the *minifilter* just forwards the IRP to the lower levels without alteration.

The use of the physical name space is activated when a request comes from the *ULS* itself or a process explicitly entitled to use it (e.g. the local components of a cloud-based storage service). For convenience, when a new process is created, it inherits the name space settings of its parent. The rest of processes also fall back to the physical name space when there is no virtual object in the requested path (that would be the case, for example, of a process accessing "`Dir1/Dir4/File5`" in the virtual name space of Figure 5.1).

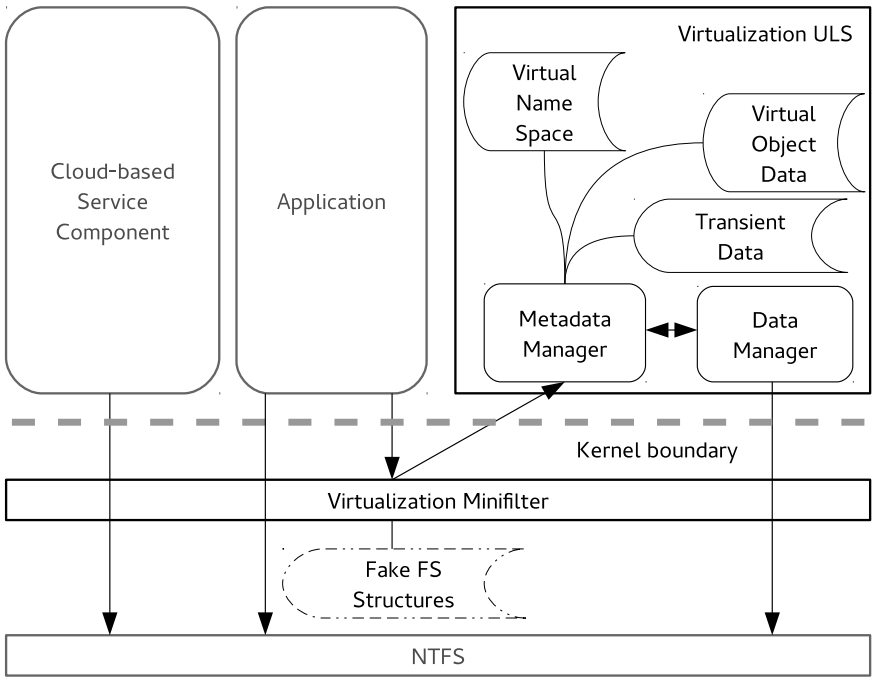

Figure 5.4: Virtualization path in a *Microsoft Windows* environment.

If the target of `IRP_MJ_CREATE` is a virtual entry, then some additional processing is required. Upon reception of the request, the *ULS* resolves the path against the virtual name space. If the target does not exist, the *Data Manager* in the *ULS* decides the physical path where the file should reside and issues a `create` request using the standard, user-level file system interface; then, the virtual name space is updated with the new entry, and the link between the new virtual object and the actual physical path is stored by the *ULS Metadata Manager*. On the contrary, if the target already existed, the mapping to an actual physical path would be retrieved by the *Metadata Manager* and fed into the *Data Manager* to open the underlying file.

In both cases, the *ULS Data Manager* actions result in issuing a standard file system operation which, in turn, will generate a new IRP going down the I/O stack. As mentioned before, the virtualization *minifilter* will let pass this IRP unaltered, and the IRP will get an open handle for the physical file (namely a user-level file descriptor).

The last step consists in binding the obtained handle on the physical file with the original IRP still pending in the virtualization *minifilter*. For that, the *ULS Metadata Manager* stores the handle and other relevant information in a transient data area, and sends a reference to that information back to the minifilter. Then, the minifilter creates fake structures to represent the virtual file and associates the *ULS* reference to them, so that the *Metadata Virtualization Layer* can retrieve the proper information when further operations are performed on the virtual object.

A representation of the IRP processing can be seen in Figure 5.4. It is important to remark that, when dealing with an IRP related to a virtual file, the processing ends

in the virtualization minifilter, and the IRP is not forwarded to the lower levels of the I/O stack (and, in particular, it never reaches the underlying file system). This adds two responsibilities to the minifilter: first, it must create the kernel structures that are usually created by the file system driver to represent a file system object (and fill them with consistent values); second, it must detect all references to these structures in the I/O stack and divert them to the virtualization *ULS*, effectively preventing them from reaching the lower levels of the I/O stack and the underlying file system itself — failure to do so could result in file system inconsistencies.

Once the `IRP_MJ_CREATE` has been successfully processed, further IRPs related to the same file will provide a reference to the kernel structures representing the file. The minifilter can easily distinguish native file system structures from fake ones (corresponding to virtual objects) by checking the associated *ULS* information that was set up during the `IRP_MJ_CREATE` processing. In general, if the IRP corresponds to a virtual file, it will be diverted to the *ULS* for processing; nevertheless, it may be possible to fulfil the request completely at the kernel level if all the necessary information is available (a particular case are `read` and `write` operations: the minifilter can interact with the *Cache Manager* to complete the request if the required data is being cached).

Another essential operation for name space virtualization is the directory listing. This operation is translated into the I/O stack as an initial `IRP_MJ_CREATE` followed by a series of IRPs of type `IRP_MN_QUERY_DIRECTORY`, which provide buffers to fill-in with directory entry information. As with the `IRP_MJ_CREATE` request, IRPs processed under the physical name space are forwarded to the lower I/O stack levels, and those corresponding to a virtual name space are sent to the *ULS*. In the latter case, the *ULS* will access the underlying physical directory, combine its entries with the entries from the corresponding directory in the virtual name space, and eventually hide any entries not to be available through the virtual name space.

Semantics

Apart from taking care of the different IRPs, the *Metadata Virtualization Layer* needs to provide the adequate file system semantics for virtual objects. Indeed, file system operations results are affected by the context (for example, a delete operation may fail if the target is open by another process), and some concurrent requests affecting the very same data must be serialized to avoid race conditions. These specifications define what an application may expect from the file system in each situation.

In the *Windows* I/O model, the *I/O Manager* issues IRPs without restrictions regarding file system semantics: it is the task of the underlying file system driver to take care of those aspects. However, the *Metadata Virtualization Layer* prevents requests related to virtual objects to reach the file system driver: therefore, it must take the responsibility of guaranteeing a behaviour compatible with the underlying file system.

The *Metadata Virtualization Layer* is able to provide the right semantics for virtual object operations through two elements: first, both the virtualization minifilter and the *Metadata Manager* in the *ULS* keep transient data to track context (including on-

going operations and information about previous related requests); second, the *ULS* incorporates a concurrency management mechanism, so that requests referring to the same pieces of data are adequately serialized to avoid consistency issues.

5.3.3 Linux-specific implementation details

Infrastructure

The implementation of the *Metadata Virtualization Layer* for *Linux* follows the same principles as the implementation based on *Windows*: the interception of file system request is based on kernel grounds, while the bulk of processing (the *Metadata Manager* and the *Data Manager* from Figure 5.2) is implemented at user-space. In this section, we will focus on the differences with respect to the *Windows*-based implementation.

The VFS (Virtual Filesystem Switch [Bovet 2008]) mechanism inside the *Linux* kernel was initially designed to facilitate the addition of file systems [Zadok 2006]; it provides an interface for registering the code that should be executed to fulfil each low level file system operation (such as name look-ups, getting and setting file attributes or reading and writing data). This set of callbacks also provides the necessary support to capture requests related to the file system name space.

The *Interception Engine* of our *Metadata Virtualization Layer* is based on the *Linux* VFS callbacks, but we use them via FUSE [Szeredi 2005] instead of registering our code directly. FUSE (Filesystem in USErspace) provides a kernel module that hooks into *Linux* VFS and exports the callbacks to a user-space application. FUSE is a standard component of current *Linux* kernels and provides a stable and widely used platform for implementing file system features in user-space. The decision to use FUSE was driven by the ease of development as well as its proven robustness. On the other hand, it hides some kernel-level information and has limited ability to interact with other kernel components, though this does not substantially hinder the implementation of the prototype.

FUSE provides two different interfaces to the user-level service implementing the VFS-like calls: a 'high-level' interface and a 'low-level' interface. We chose the 'low-level' interface because it allows a slightly greater control of the behaviour of the file system operations (at the cost of having to maintain and take care of some additional information); in particular, the 'high-level' interface is based on paths and automatically caches directory entry data into the kernel, which is inappropriate for our purposes because the *Metadata Virtualization Layer* must be able to change the mapping between directory entries and objects dynamically, depending on the name space being used. On the contrary, the 'low-level' interface is based on *i-node numbers* and allows full control of directory entry mapping and its caching at the kernel level.

One of the main differences between the implementation of the *Interception Engines* in *Microsoft Windows* and *Linux* is that the combination of VFS and FUSE does not allow file system requests to pass through the *Metadata Virtualization Layer* and reach the underlying file system: FUSE diverts all requests to the User-Level Service

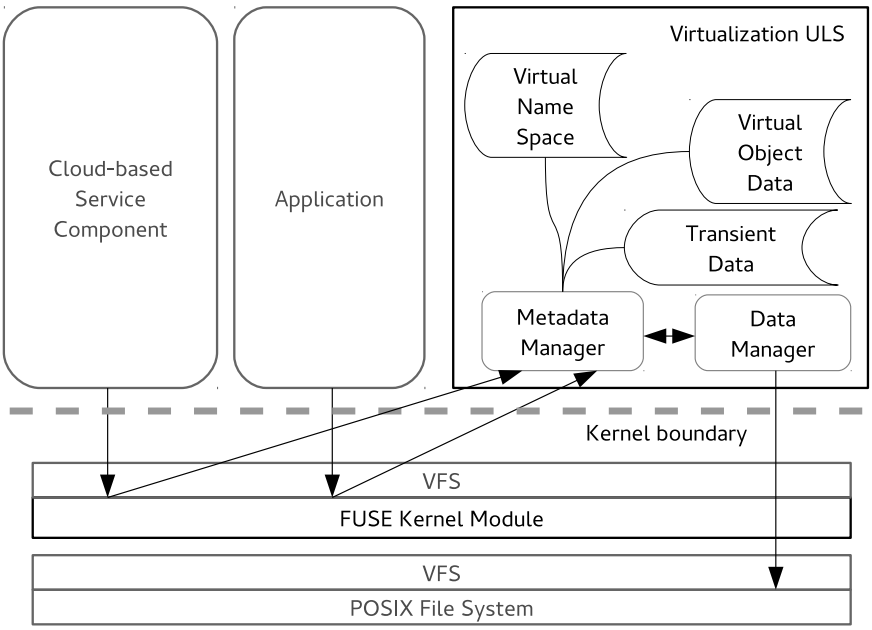

Figure 5.5: Virtualization path in a *Linux* environment.

(*ULS*). In particular, this includes both requests coming from the cloud-based service components (which should access the physical name space) and from the rest of the applications (which should access the virtual name space). Therefore, it is the *ULS* who must explicitly forward to the underlying file system the operations related to the physical name space.

Figure 5.5 illustrates the behaviour of the *Metadata Virtualization Layer* in *Linux*. The *Metadata Virtualization Layer* attaches to VFS via the FUSE kernel module and appears as a conventional file system which is mounted on top of the physical name space (i.e. the underlying POSIX file system). When an application performs a file system operation, this operation is converted into one or more VFS requests, which are intercepted by the FUSE module and diverted to the *ULS*; if the request is to be processed under the physical name space, the *ULS* generates the corresponding series of operations for the underlying file system.

In order to bypass the VFS layer and be able to perform operations directly on the underlying file system, the *ULS* acquires a handle of the root directory of the underlying file system before the VFS layer is mounted on top of it. Then, whenever a physical object is to be accessed, the physical path is resolved after this handle, instead of using the conventional path resolution mechanism; in the prototype, this is enabled by using the `openat` system call and its relatives.

Name space mapping

As discussed in subsection Section 5.2, path resolution is one of the key elements to provide the perception of a virtual name space to applications. In the case of

Microsoft Windows, the requirement to resolve a path was embedded as part of the IRPs representing the different operations. In the *Linux* implementation, path resolution is performed separately from the actual operation via explicit VFS+FUSE `lookup` callbacks; the path resolution procedure returns an internal identifier which is then used to invoke the actual operation (`create`, `mkdir`, `unlink`, etc).

Apart from the formal difference, the treatment of `lookup` in the *Linux*-based prototype is equivalent to the *Microsoft Windows*-based prototype: if the request refers to the physical name space, the *ULS* generates a new request for the underlying file system; otherwise, the *ULS* tries to resolve it against the virtual name space data, and may fall back to check the underlying file system if the entry is not found.

The processing of directory listings is also equivalent to the procedure followed in the *Microsoft Windows*-based prototype, with the sequence of `IRP_MJ_CREATE` and `IRP_MN_QUERY_DIRECTORY` IRPs translated into an `opendir` followed by a series of `readdir` callbacks, with essentially the same semantics as their *Windows* counterparts.

Tracking directory changes

User notification of file system changes in the *Linux* prototype deserves a specific comment. In the *Linux* kernel, the change notification mechanism (usually *inotify* [Love 2005]) is implemented inside the VFS and reacts to requests coming from the user-space, without requiring any action from the underlying file system. Specifically, FUSE delegates all change notification handling to the *Linux* VFS. Unfortunately, this means that FUSE does not provide any means to notify the user applications that changes have occurred behind the scenes.

This limitation would affect a component of the cloud-based service that wished to track the changes in the cloudified directory through the VFS layer. When the user application acts on an arbitrary directory in the virtual name space, the *ULS* could directly update the cloudified directory in the underlying file system; but then, the VFS associated to the *Metadata Virtualization Layer* could not relate the virtual directory with the physical cloudified directory and the cloud service component would not be notified.

The initial solution used in the *Linux*-based prototype makes the *Data Manager* of the *ULS* redirect updates to the cloudified directory through the VFS layer, instead of writing directly to the underlying file system; that makes the VFS aware of the changes, so that notifications work properly. An alternative solution to the extra indirection level consists in modifying FUSE to allow informing the VFS about the changes made by the *ULS*. FUSE already provides a mechanism for sending requests from the user-space to the kernel module (e.g. to invalidate data cached by the kernel); so, the same feature can be used to push change notification events into the kernel and, upon reception, the FUSE kernel module can invoke the proper change notification kernel interface.

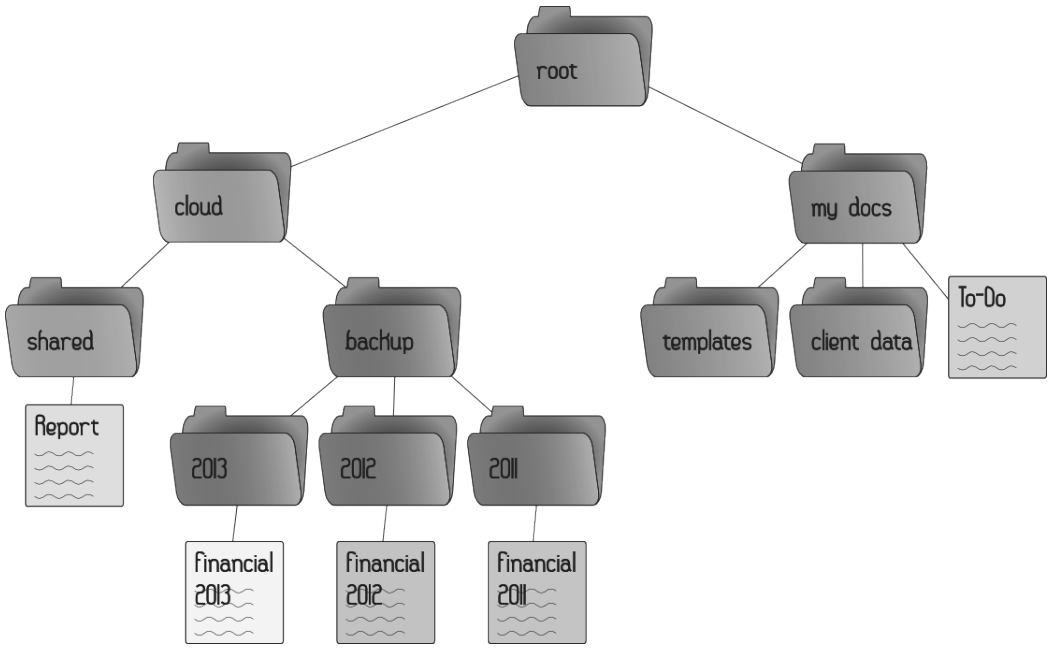

Figure 5.6: Example of a traditional file system organization.

5.4 Outcome

In this chapter, the goal was using the metadata virtualization framework to improve the usability of a diversity of file systems from the final user's perspective. The technical aspects discussed in Section 5.3 provide the insights necessary to be able to interact with existing operating systems, enabling the possibility to fulfil the requirements of the metadata virtualization framework.

The result of the techniques described above is the addition of new functionalities to the existing file system. These new functionalities provide the final user with new ways to handle the storage system that, previously, were either unavailable or impractical.

Figure 5.6 shows an example of a traditional file system organization. The example represents a production environment where an employee has to prepare a public report based on financial data for the current year and, occasionally, has to check data from previous years. We can see that the necessary files are scattered across different branches of the directory hierarchy; the reason is that each branch has specific features that we want applied to specific files. For instance, the public report is stored in a "shared" directory attached to a cloud storage service, so that other people has access to it; the financial data corresponding to each year is also located in a cloud-based backup service, with a rigid structure to allow the easy identification and retrieval of data from a certain year; finally, the user also holds some files (such as a "to-do" list and some useful report templates) in his own documents folder in the local disc.

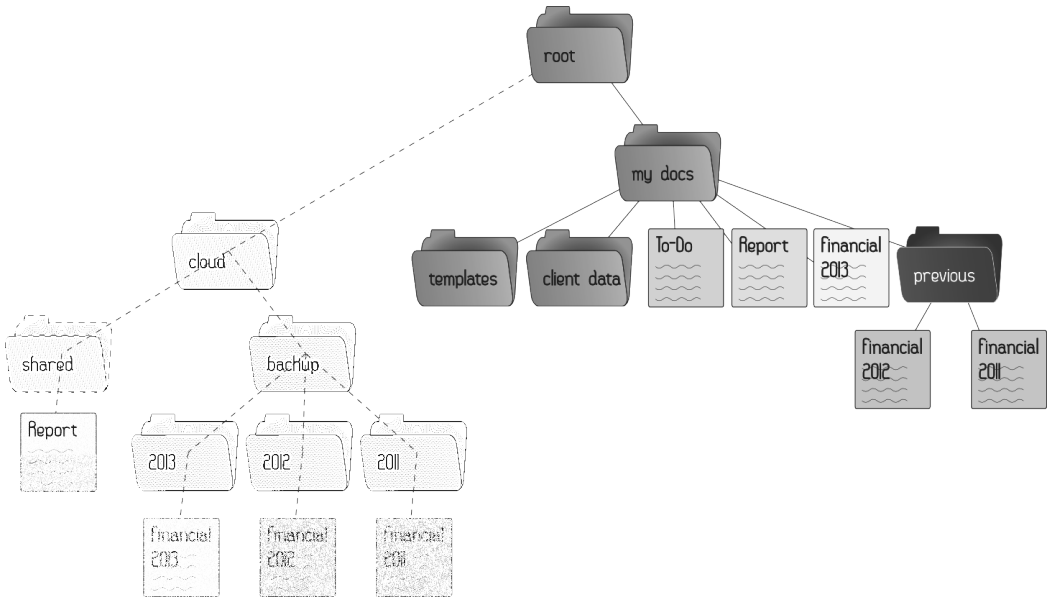

Figure 5.7: Example of a flexible organization enabled by the metadata virtualization framework.

Despite having all the needed files with the required features, this scenario is uncomfortable for the final user because all files are scattered, instead of being held together in a single place to be used in a combined way. The options provided by a traditional system are not specially appealing: the user can either copy all files to a single place temporarily and copy them back after the work is done, or try to use links or shortcuts to have some appearance of the files being together.

In the first case, the copying procedure is prone to failure: a wrong path can result in the unduly overwrite of valid data, or in the failure to update a file with old contents. Moreover, the copies would miss the features provided by the specific branches of the directory tree (in the example from Figure 5.6, any updates to financial data files copied into the local documents folder would not be backed up until they are copied back to their original places, and any modifications of a local copy of the public report would not be visible by other users). Alternatively, the use of links also has issues: first, they have to be maintained manually (a cumbersome procedure if there are many files to handle) and, second, the semantics of the operations on links is not identical to the equivalent operations on files, which could cause unexpected results (for instance, the removal of a shortcut does not involve the removal of the original file containing the actual data; and a 'move' operation over a symbolic link causes the link to be severed, instead of replacing the contents of the original file).

The use of the techniques described in this chapter allows the final user to have a comfortable organization (having all the necessary files together), but without the issues mentioned in the previous paragraphs: no manual maintenance, no missing features and no changes in semantics.

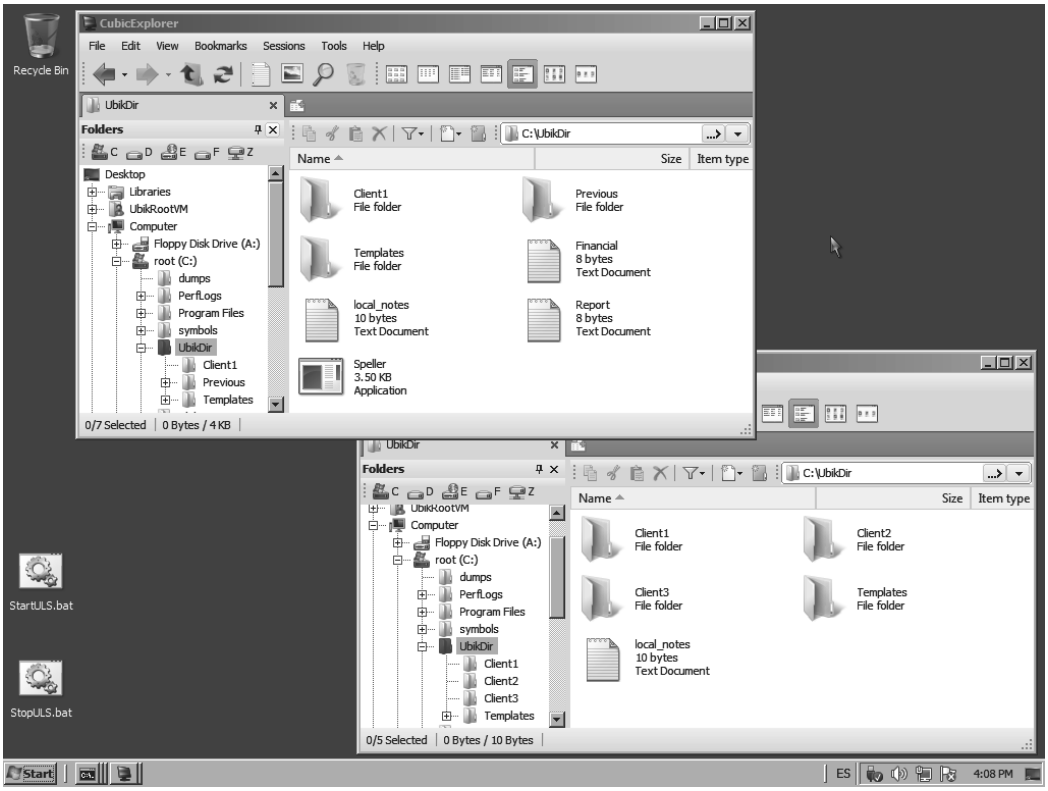

Figure 5.8: Screenshot from the *Windows* prototype of the metadata virtualization framework.

Figure 5.7 shows the result of applying the methods discussed in this chapter to the example from Figure 5.6. The metadata virtualization framework provides a name space where the user can combine the local files (such as the "`to-do`" list and the templates) with files from the branches attached to a remote cloud storage service (such as the shared report document and the financial data for the current year). The user can even create virtual directories to group physically disperse files (like the "`previous`" directory containing the financial data for previous years).

The binding between the virtual files and the actual physical files scattered across the cloud-related directories is hidden from the user and, once defined, it is internally handled by the metadata virtualization framework without further user intervention; in the example of Figure 5.7, the complete cloud-related subtree is even suppressed from the user view. Nevertheless, the logic in the *ULS* component of the metadata virtualization framework still takes care of properly mapping the operations on the virtual view to the actual files in the cloud-based subtree.

Summarizing, the result of the presented techniques is the possibility of simultaneously having multiple flexible organizations of the file system, adapted to specific needs, and which existing applications can use transparently without needing any modifications.

As an example, Figure 5.8 presents a screenshot from the *Windows* prototype of the metadata virtualization framework showing two applications working on two different views of the same file system, similar to the examle presented in Figure 5.6 and Figure 5.7. The bottom right browser presents a folder containing a file ("`local_notes`") and a few subfolders ("`Templates`", "`Client1`", "`Client2`" and "`Client3`"); the top right browser presents the same folder with a different view, containing the "`local_notes`" file and the "`Templates`" and "`Client1`" directories from the other application, but also some additional virtual objects from a remote location (virtual files "`Report`" and "`Financial`", a "`Speller`" application, and a virtual directory "`Previous`".

Additionally, the view in the top left browser hides two of the directories in the alternative view ("`Client2`" and "`Client3`". This ability to selectively hide portions of the file system is another example of a functionality that cannot be achieved with the means provided by traditional file systems.

Both views from Figure 5.8 are fully functional: applications can be executed, and files can be opened with the usual applications, regardless of them being virtual or not. The construction of this prototype allowed us to verify that the metadata virtualization framework was a feasible mechanism to provide the flexibility demanded by modern file system environments.

5.5 Extensions

The model of having a physical and a virtual name space can be easily extended to multiple, simultaneous, virtual name spaces. In fact, the prototypes described so far permit such multiplicity, and the only addition needed is the logic to select the appropriate name space to resolve a given path.

Using multiple virtual name spaces enables presenting specialized views of the file system to a particular application or group of applications. Following the case described in the paper, we could have, for example, multiple cloud-based storage services using incompatible directory layouts coexisting in the same system.

Additional features can also be added, such as hiding part of the files to certain services, quarantining the changes from certain applications or selectively encrypting sensitive data.

In general, the mechanisms explained in this chapter allow any feature specific of a given file system to be made globally available to the whole file system name space, without having to limit its use to specific directories or *mount points*.

5.6 Conclusion

Cloud-based storage services provide means to replicate local data and keep it synchronized across different platforms. To keep this data up to date, local components

of the cloud services need to track changes in the local file systems. However, current mechanisms to perform this tracking require rigid file organizations in order to be effective.

The contribution in this chapter consists in using our metadata virtualization framework to improve the flexibility and ease of use of cloud-based storage systems. We remove the need to use strict directory hierarchies by providing multiple simultaneous views of the file system organization: the *Metadata Virtualization Layer* allows user applications to access a view of the file system without organizational restrictions, while cloud components maintain a view of the file system structured according their needs.

The prototypes presented in this chapter show how virtual name spaces can be implemented on top of both *Windows* and *Linux* platforms, demonstrating the feasibility of the approach in actual systems.

CHAPTER 6

Related Work

Contents

6.1 Related work

There is a lot of previous work that analysed the limitations of file systems and addressed them from different perspectives, many of which have influenced the development of the techniques used by the *COFS Metadata Virtualization Layer*. This section summarizes some of the developments related to the present work.

6.1.1 Metadata management

Some advances in metadata handling are related to where and how metadata should be stored. While it was initially considered a good thing to mix data and metadata to improve cache efficacy, some research pointed out that access patterns differed greatly for data and metadata and, therefore, they should be treated separately.

Systems like *MFS* [Muller 1991] followed the proposal of having different structures and caching policies for data and metadata, physically separating them in different storage systems when possible, in order to increase performance and avoid interferences with data accesses.

DualFS [Piernas 2002] is also separates data and metadata, but it does not require placing them in different devices. Metadata is organized as a log file system (*LFS* [Rosenblum 1992]) to improve write performance and crash recovery, using metadata dynamic relocation and caching to adapt to changing access patterns. Similarly, *hFS* [Zhang 2007] uses the same approach, but intermixing very small data files with their related metadata to improve access time.

The systems mentioned above demonstrate the advantages of clearly separating the file system metadata from the bulk of file contents. Nevertheless, they still amal-

gamate the metadata, focusing on extracting the maximum efficacy from the storage device transfers. Therefore, additional techniques are required to improve both the adequacy to multiple simultaneous accesses and the flexibility in the file system organization.

Directories are usually represented as special files containing collections of entries; these collections were originally structured as simple lists, but their internal organization has evolved to deal with large amounts of entries (for example, extendible hashes [Tang 2003] are commonly used to handle directory entries). As a consequence, manipulating a directory involves making sure that the underlying data structure supporting it is kept consistent.

The cost of handling large sets of loosely related attributes or directory entries together and keeping them consistent is becoming important in modern file systems and, in particular, the effects of excessive synchronization traffic are much noticeable for parallel file systems [Cope 2005].

Some mechanisms have been proposed to mitigate this effect and increase metadata update performance. The *soft updates* mechanism [Ganger 1994] tracks dependencies between directory entries and *i-node* modifications to be able to commit partial changes to metadata, without being tied to block granularity. The *Zettabyte File System* [Bonwick 2003] implements a transactional mechanism to keep track of unrelated modifications and to avoid unnecessary locking (though it is not designed to support parallel multi-node accesses).

Specifically focusing on directory issues, some systems use special techniques to be able to manipulate different portions of a directory independently, reducing the *false-sharing* effect. A common approach is to use some kind of hash-based partitioning allowing to diversify the target structures of the operations and, thus, minimize potential collisions. For example, *OBFS* [Wang 2004] does a heavy use of hashes to handle a flat name space inside an object storage device (OSD) as if it were a single directory, and *GANESHA* [Deniel 2007] uses a multi-level combination of hash functions and red-black trees to deal with large directories and metadata sets while reducing the risk of conflicts during parallel access.

An alternative approach to break the traditional granularity of the metadata consists in leveraging database technologies and use tables to store metadata and, in some cases, also file contents.

The use of databases presents additional advantages such as easily allowing the extension of metadata with extra information (e.g. user-defined attributes) and enabling the recovery of data by attribute query in addition to path-based access [Olson 1993]. Conceptually, this would bring flexibility to file system organizations, allowing mutable name spaces based on file attributes.

On the downside, fully-fledged databases are claimed to be slow for heavily loaded file systems (though new developments may show otherwise), and some POSIX operations like directory streaming, and extended operations like per-attribute searches, may be difficult to implement efficiently [Ames 2005].

Despite the performance concerns, several metadata management services have been successfully based on databases. For example, *Chimera* [Mkrtchyan 2006] provides the name space service for the *dCache* distributed system [Fuhrmann 2006] by means of a database. The *Logic File System* [Padioleau 2003] also uses a database to allow file access by attribute selection. The *Linking File System* [Ames 2005] uses a database to enrich the metadata with information about navigable relationships among files, though it relies on non-volatile main memory technologies for performance. The *Trove* storage subsystem for *PVFSv2* [PVF 2003] also may use a database for metadata storage.

6.1.2 Parallel file systems

Currently, platforms for high-performance computing have shifted from classic big single-node supercomputers to large clusters of smaller nodes linked by low-latency interconnect networks. In 2005, 60% of systems in the Top500 supercomputer list were considered clusters [Boulton 2005] and by June 2006, the ratio reached 73% [Stein 2007]. Just to mention an example, the MareNostrum supercomputer at BSC has more that 10,000 processors distributed in more than 2,500 nodes. Parallel applications running on such systems expect to be able to perform I/O simultaneously from all the nodes as they would do in a single node (i.e. efficiently, in parallel and keeping the consistency).

So, together with the rise of clusters, parallel file systems have appeared to fill the gap and offer a consistent single view of the storage system across all nodes in the clusters, trying to handle simultaneous parallel reads and writes accesses to many files in many directories, many files in a single directory or even a single file. As of this writing, three of such systems have reached a state mature enough to dominate the production-grade arena: *Lustre* [Braam 2007], *IBM GPFS* [Schmuck 2002] and *PVFSv2* [Ross 2000] [PVF 2003].

Lustre claims to offer a fully compliant POSIX interface and is based on three types of components: the clients (nodes accessing the file system), the storage servers (based on object storage devices — OSDs) where the data is kept, and metadata servers, responsible for keeping the name space, access rights, and consistency management. Current versions have unique metadata server (with a failover replacement) to simplify consistency management. Current scalability expectations are about a thousand of client nodes and a few hundred of storage nodes. Communications between the components is done via an optimized network stack [Brightwell 2002]. *Lustre* has been used to provide the root file system for a cluster of disk-less nodes [Boggs 2006].

IBM GPFS also offers a POSIX interface, while having the possibility to use non-POSIX advanced features for increased performance (e.g. for MPI-IO). Unlike *Lustre*, *IBM GPFS* uses block-based storage. A typical configuration consists of clients which access to a set of file servers connected to the storage devices via SAN. Metadata is also distributed, and its consistency is guaranteed by distributed locking, with the possibility of delegation to increase performance in case of exclusive access. Current in-

stallations such as the ones at BSC reach a few thousand of nodes, and continent-wide multi-cluster configurations have been also tested in production-like environments as part of the DEISA infrastructure [DEI 2008].

PVFS [Ross 2000] also offers a POSIX interface, though the system provides a native API mostly designed to support advanced parallel interfaces such as MPI-IO. Like *Lustre*, metadata is managed by a single server, which provides data locations to the clients which, then, access directly to storage servers. Both data and metadata are stored as regular files using the local file systems present in the nodes where the server daemons are run. The file system was mostly rewritten for its version 2 [PVF 2003] in order to have a modular design for the networking and storage subsystems and to support flexible and extensible data distribution policies. There is also support for limited metadata distribution, data replication and fault tolerance. The counterpart is that the system has evolved towards providing high-performance for advanced parallel interfaces (such MPI-IO), and POSIX semantics has been relaxed.

The performance of these systems has been analysed in [Cope 2005] and [Hedges 2010]. All of them achieve good performance for reading and writing data in parallel in a moderate-size cluster, but *PVFSv2* is heavily penalized when being used by a single node. The differences in metadata handling between *Lustre* and *IBM GPFS* are also significant: the unique metadata server in *Lustre* seems to take advantage over *IBM GPFS* in case of simultaneous access to same directories, while *IBM GPFS*'s distributed locking appears to be much better for scattered accesses. This means that the way users organize their file hierarchy may have a great impact on systems performance. It is worth mentioning that the high specialization of *PVFSv2* in parallel data reads and writes has a price in terms of poor metadata performance [Cope 2005].

6.1.3 Metadata consistency

How to handle metadata is an important factor that must be considered carefully when dealing with parallel and distributed file systems. This also applies to the *COFS* framework, as the separation of metadata and name space handling from the actual data layout is a key feature to obtain performance benefits.

IBM GPFS [Schmuck 2002] can distribute the name space and metadata across several nodes, integrated in the same servers used to store data (in fact, directories are essentially treated as special files)

On the contrary, other file systems, such as *Lustre* [Braam 2007], *Ceph* [Weil 2006] or *PVFSv2* [PVF 2003] decouple metadata and name space management from data accesses, having specialized services and distinct storage mechanisms for name space handling. In all these cases, the metadata also includes information to reach the physical location of file contents. This is also true for the decoupled architecture of *pNFS* [Hildebrand 2005] (where the information returned by the metadata server also contains information about data distribution) and the object manager in *Ursa Minor* [Abd-El-Malek 2005].

COFS Metadata Service separates name space and metadata management from the

file data handling and does not keep any information about how and where file data is physically stored: data handling is a responsibility of the underlying file system. This allows us to have a lighter server with a reduced load (e.g. there is no need to contact the *COFS Metadata Service* if a file is resized or has new data appended).

Mechanisms to maintain name space coherence also vary with file systems. For example, *IBM GPFS* uses distributed locking techniques, with the possibility of delegating data management to the client under controlled circumstances. These techniques aim to prevent bottlenecks and to facilitate parallel access to disjoint sets of information; the drawback is that distributed locking and leasing techniques are complex and also require so sophisticated fault detection and recovery mechanisms.

In a completely opposite approach, *PVFSv2* prevents consistency problems by distributing metadata with a "no shared data, no cached data" policy [Sebepou 2008]; and *Lustre* uses a single metadata server to avoid conflicts. *Panasas* [Welch 2008] (closely related to *Lustre*) organizes the directory tree into separate 'volumes', having a single independent metadata servers for each volume.

COFS approach for metadata handling is oriented towards the centralized service model, combined with a limited client caching mechanism to speed-up cases with low risk of conflicts.

6.1.4 File organization versus low-level file system layout

Users like to organize their files in such a way that related things are close in the directory tree. Traditional file systems assume that 'related' things are to be stored together (and possibly to be treated in similar ways).

The idea of using directories as a way to group things that are handled together is implicit in the concept of *mount point* (a directory that is used to traverse file system boundaries): creating a file beyond one of these directories determines where it will be stored.

This concept is still very alive in modern file systems. For example, *Panasas* makes the volumes visible as directories in the name space root. *ONTAP GX* [Eisler 2007] is also based on volumes; they tend to be small and the *mount point* directories (called *junctions*) can be located anywhere in the name space; physical re-location and replication, when needed, is applied to the whole volume.

An alternative to the "group by directory" philosophy is based on stackable file systems. Stackable file systems are a well-known technology that can be used to extend the functionality of a file system [Zadok 2006]. In particular, fan-out stackable file systems (or *union file systems*) can combine the contents of several directories (possibly in different file systems) into a unified virtual view [Pike 1992] [Pendry 1995] [Wright 2006]. Essentially, each operation is forwarded to the corresponding objects in all the underlying file systems and the result is a sensible composition of the operation results for each layer, including file attributes and other metadata.

Typical applications of *union file systems* are 'live' installation environments based on read-only media (e.g. a CD providing a read-only file system with the base operating environment is combined with a RAM-based file system supporting site-specific configuration files and temporary read-write storage), development environments for building the code on top of read-only sources like *Diverge FS* [Lin 2004], or automatically versioning files using copy-on-write techniques [Yamamoto 2000]. All these examples exploit the same concept: having an immutable base (possibly the combination of several sources), and a single writeable layer that allows modification (either temporary or permanent).

Recently, systems like *Blutopia* [Oliveira 2007] went a step further, combining the stackable file system principles with distributed environments to provide a mechanism for centralized cluster management. Essentially, each different service (mail, web servers...) and even each different configuration for the service is set up as a file system layer stored in central servers. Then, each client node in the cluster can combine the remotely exported layers to offer the necessary services. Client re-provisioning can be done by simply changing the stacked layers, and maintenance and upgrade tasks can be carried out safely in a test system and then propagated to all clients by changing the exported images. The idea behind this mechanism is similar to data center management systems based on virtual server farms, such as *Parallax* [Warfield 2005], but without imposing the need of a virtual machine to support each service.

RAIF [Joukov 2007] (Redundant Array of Independent Filesystems) exploits stackable file systems in quite a different way. Instead of limiting the writeable layers to the topmost, *RAIF* replicates the directory tree and distributes files across several layers. A set of rules specifies the distribution policy (in which layer a file must reside, where to replicate it and even how to stripe it across layers). It is assumed that each layer is backed by a file system with different features, and the goal is to optimize system behaviour by placing the files in the most adequate file system according its probable use (for example, temporary files could be placed in local or even temporary RAM disks while large media files can be stripped across file systems optimized for streaming access). Nevertheless, one of its limitations is that the directory tree has to be replicated in all branches, making the mechanism difficult to scale to many underlying file systems.

Ursa Minor [Abd-El-Malek 2005] also allows specifying completely independent storage policies on a per-object basis (in fact, there are no directories in *Ursa Minor*: data objects have an identifier in a flat name space and are accessed by non-standard protocols — unless accessed via an *NFS* front-end).

COFS also treats directory contents as a potentially unrelated objects. The main difference from *RAIF* is that metadata and name space handling are decoupled from the underlying file systems. Regarding *Ursa Minor*, *COFS* differs in the separation of pure metadata from data distribution information (as mentioned in previous subsection) and the fact that *COFS* does provide a standard conventional interface, so that any standard application can make direct use of it.

6.1.5 Virtualization

The notion of virtualization is not new to file systems; on the contrary, it is commonly used at different levels to hide complexities and extend functionalities.

Acting on the lowest device level, *Parallax* [Warfield 2005] uses a virtual machine to offer a block-based interface hiding a distributed storage environment. In a similar way, *Peabody* [Morrey III 2003] uses a software-based iSCSI target to provide virtual disk images that a local file system can then use as a backing store.

At operating system level, mechanisms such as the *Linux* VFS provide a common layer aimed to facilitate the integration of different file systems into the OS internal structure [Zadok 2006].

At the infrastructure level, *ONTAP GX* [Eisler 2007] virtualizes the file system servers and network interfaces used to access the file system data, offering a single virtual large server with multiple interfaces and allowing transparent re-provisioning of physical servers. A similar approach is used by *Slice* [Anderson 2000].

The same transparent redirection techniques are used for server off-loading to achieve load balancing (*Cuckoo* [Klosterman 2002]) or efficient power management in data centres [Narayanan 2008].

Name space virtualization has been used to provide a unified virtual view of multiple file systems: well-known examples of this technique are *union file systems* [Pendry 1995] [Wright 2006]. Fan-out stackable file systems [Zadok 2006] are one of the mechanisms allowing the construction of unified virtual name spaces; for example, *RAIF* [Joukov 2007] uses this technique to combine several file systems with the same directory hierarchy under a single virtual name space, with the goal of diverting files to the most appropriate underlying file system according to each file's characteristics. Our approach uses a similar technology with the opposite goal: instead of unifying several file systems under a single virtual name space, we provide multiple virtual name spaces for a single underlying file system.

ONTAP GX [Eisler 2007] also provides a virtual layer allowing volumes (directory sub-trees) to be transparently re-located or replicated. On the other hand, *RAIF* [Joukov 2007] uses virtualization to divert the target file system for individual files, though the directory tree itself is not virtualized, but directly mapped into the underlying file systems.

The SDSC Storage Resource Broker [Baru 1998] is also able to provide a uniform interface to a variety of heterogeneous storage resources. Several physical storage resources can be combined into logical storage resources as means of transparently providing replication support. The *SRB* is also able to split a large file from the user view and scatter it across several storage resources and to provide compound resources that combine features from different physical storage systems, such as combinations of tapes and cache disks [Wan 2003].

COFS exploits virtualization at the name space level, using single file granularity and virtualizing also the directory hierarchy. User-space mechanisms have been pre-

ferred to kernel integration because of the ease of development and deployment in production-grade environments.

6.1.6 Issues on large-scale file systems

Potential performance risks for parallel file systems have already been spotted. For example, the performance drawbacks caused by mismatches between the workload characteristics and the file system configuration are commented in [Abd-El-Malek 2005].

Some studies tried to determine experimentally the strong and weak points of different parallel file systems and how they behave under different workloads [Cope 2005] [Sebepou 2008] [Hedges 2010]. Another evaluation, focused on *PVFSv2*, aimed to isolate the different causes of performance losses and determine their impact [Kunkel 2007].

Efforts are already being directed to mitigate some of the issues and limitations, either by means of file system-specific modifications to improve certain metadata operations [Devulapalli 2007], or by means of a middleware to be used by certain classes of applications [Yu 2007].

The usual path of action consists in adding optimizations at some level so that the file system is able to deal with the spotted issue in a reasonable way. Additionally, *COFS* allow for a complementary approach: automatically reorganize and avoid situations where specific patterns are harmful for the overall performance of the file system.

The issues caused by multiple nodes writing into shared structures and, specifically, by large-scale application check-pointing, were one of the motivations for the development of *PLFS* [Bent 2009]. *PLFS* uses an approach similar to *COFS* and use an interposition layer to change the underlying file system layout in order to improve the behaviour of parallel I/O; nevertheless, they focus on parallel writes to shared files and data transfer bandwidth, rather than the metadata issues originated by the parallel creation of files in shared directories, which are addressed by *COFS*.

A different approach to solve the metadata overhead of creating large collections of files is represented by the *SIONlib* [Frings 2009]. This approach consists in mapping an entire collection of files into a single physical file (or a small number of them), avoiding contention at metadata servers. Contrary to *COFS*, *SIONlib* is not transparent, and applications must explicitly call its API to make use of it; being designed for high-performance applications, it assumes certain restrictions in the characteristics of the files and the way they are used (e.g. their size must be known in advance, and all files must be created from the same parallel application). *COFS* solution may introduce more overhead, but it offers greater flexibility and does not have such limitations.

6.1.7 Cloud storage

Nowadays there are multiple commercial products providing cloud-based storage with added value services, such as data synchronization across devices or collabo-

rative sharing. Many of these services are integrated in the form of one or more selected directories in the existing file system (like *Box* [Box 2013], *Dropbox* [Dro 2013], *JustCloud* [Jus 2013] or *SugarSync* [Sug 2013], to name just a few).

In all these products, the use of the cloud is activated by moving the files into specific places (either drives or directories); on the contrary, our proposal enables the cloud services support independently of a particular file's location.

The combination of cloud storage and local storage also has similarities to the compound digital entities handled by the *SRB* data grid [Wan 2003]. The *SRB* can manage logical compound resources that combine different storage resources (such as tapes and disk-based caches). One of the differences with our framework is that we handle the composition at the name space level, while *SRB* does it at the storage resource level, providing a low-level logical storage device.

6.1.8 Implementation techniques

Regarding technical aspects, the implementation of the *minifilter*-based interception engine for the *Windows* prototype borrows ideas from the 'isolation driver' concept from OSR [OSR 2010] [OSR 2011].

The *Linux*-based prototypes rely on FUSE [Szeredi 2005] to interface with the underlying file system.

CHAPTER 7

Conclusion

Contents

7.1 Conclusion

In this thesis, we have presented a mechanism to decouple file system metadata management from the underlying file system organization. This technique allows to present multiple virtual views of the name space and the file system object attributes that can be adapted to specific application needs without altering the underlying storage configuration.

In the first contribution, we have introduced the design of a metadata virtualization framework that makes possible such decoupling. This framework incorporates mechanisms to overcome the limitations of traditional file systems, imposed by the dependence of metadata on their internal structures. First, we have replaced coarse-grained block-based metadata amalgamations by fine-grained database entries that enable more agile metadata access and manipulation. Second, we have empowered the static data structure representation of file system objects with active objects embodied as light-weight processes, making them more suitable for parallel independent management and scalability. Finally, we have proposed a technique to deal with large directories by using a stateless partitioning method, fulfilling the standard directory listing semantics without requiring locks being held for long times.

The second contribution consists in a method to improve file system performance by using the metadata virtualization framework. The proposed implementation of the framework is able to transparently modify application requests before they reach the underlying file system. Unlike usual file system optimizations, which introduce specialized features to improve the response in very specific situations, our method allows for a complementary approach: to automatically reorganize requests to avoid situations that are harmful for the overall performance of the file system. The framework prototype has proved this approach feasible, showing that the theoretical overhead can be largely compensated by its benefits. In particular, the prototype has been successfully used to boost file system performance for shared parallel workloads. More specifically, we are able to obtain a speed-up factor of 10 for parallel file creations on *IBM GPFS* and we eliminate the parallel metadata performance degradation when using 32 nodes or more; on *Lustre*, we have been able to mitigate the performance

issues and reduce the parallel overhead of `create` and `open` operations, reaching a speed-up factor of nearly 2 on 64 node configurations.

The third contribution of this thesis consists in a technique to improve the usability of cloud-based storage systems in personal computing devices. We have analysed the reasons behind the coarse-grained nature of cloud-based storage support and presented a proposal consisting in the use of virtual name spaces to provide fine-grained, ubiquitous support for cloud-based storage. We have shown that our metadata virtualization framework can be adapted to this end, and have presented the means to integrate it with two widely used operating systems: *Microsoft Windows* and *Linux*. The result is an environment that seamlessly integrates both local storage and cloud storage, where the actual location of the data is determined by their attributes and not its position in the directory tree.

7.2 Open research areas

The work in this thesis also revealed areas that may be worth of further exploration:

- The use of a metadata virtualization framework allows to reorganize files in the underlying file system to avoid performance penalties. The high-performance oriented prototype developed in this thesis is based on the study of the target file system in order to identify its weak points and prevent applications from hitting them. An alternative approach would consist in having several underlying file systems with different behaviours and an intelligent component of the virtualization framework able to divert the files to the most appropriate file system according to its use. In order to be effective, this could require the use of machine learning techniques in order to determine the needs of particular applications.

- The metadata virtualization framework used to make cloud-based files independent from their location in the name space could also be used to enable other file-specific features anywhere in the file system. Some specially interesting features in modern environments are encryption, off-line availability of remote files, and life-cycle management. Although these features are available by means of specific external applications, a metadata virtualization framework would offer a clean way to integrate them seamlessly under a conventional file system interface, making them readily available to all applications.

- Traditionally, the file system object attributes have been grouped and stored together, regardless of the meaning and pattern of use of their values. During this work, we have observed that this grouping may generate consistency-related issues and jeopardizes the effectiveness of performance-critical mechanisms such as caches. It would be interesting to analyse how the different attributes are used, which is their volatility, and when precise values are needed and when they can be used just as hints. The result of such study would lead to more efficient ways

to group and store attribute values and should inspire the design of better file system interfaces able to specify the information that is really needed.

- In this work we have used lightweight processes to represent file system objects in the virtual name space. These active objects fulfilled the requirements to implement the behaviour of traditional files. Nevertheless, the possibilities of using active objects are much broader, and could be used to implement storage objects with more functionalities, such as being able to generate replicas of themselves and keep them updated, establish relationships with other objects and react to their changes, or even be able to execute arbitrary self-processing algorithms as, for example, extracting summary information from their contents.

Bibliography

[Abd-El-Malek 2005] Michael Abd-El-Malek, William V Courtright II, Chuck Cranor, Gregory R Ganger, James Hendricks, Andrew J Klosterman, Michael P Mesnier, Manish Prasad, Brandon Salmon, Raja R Sambasivan*et al. Ursa Minor: Versatile Cluster-based Storage.* In FAST, volume 5, page 163, 2005. (Cited on pages 124, 126 and 128.)

[Adya 2002] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer and Roger P Wattenhofer. *FARSITE: Federated, available, and reliable storage for an incompletely trusted environment*. ACM SIGOPS Operating Systems Review, vol. 36, no. SI, pages 1–14, 2002. (Cited on page 12.)

[Aiken 2003] Stephen Aiken, Dirk Grunwald, Andrew R Pleszkun and Jesse Willeke. *A performance analysis of the iSCSI protocol*. In Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on, pages 123–134. IEEE, 2003. (Cited on page 11.)

[Ames 2005] Alexander Ames, Carlos Maltzahn, Nikhil Bobb, Ethan L Miller, Scott A Brandt, Alisa Neeman, Adam Hiatt and Deepa Tuteja. *Richer file system metadata using links and attributes*. In Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on, pages 49–60. IEEE, 2005. (Cited on pages 122 and 123.)

[Anderson 2000] Darrell C Anderson, Jeffery S Chase and Amin M Vahdat. *Interposed request routing for scalable network storage*. In Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4, pages 18–18. USENIX Association, 2000. (Cited on page 127.)

[Artiaga 2010] E. Artiaga and T. Cortes. *Using filesystem virtualization to avoid metadata bottlenecks*. In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, pages 562–567, 2010. (Cited on page 4.)

[Artiaga 2011] Ernest Artiaga, Toni Cortes and Jonathan Martí. *Virtual Shell (PCT/EP2011/053410)*, 2011. (Cited on page 3.)

[Artiaga 2013a] Ernest Artiaga and Toni Cortes. *Lessons Learned about Metadata Performance in the PRACE File System Prototype*. PRACE White Paper, May 2013. (Cited on page 4.)

[Artiaga 2013b] Ernest Artiaga, Jonathan Martí and Toni Cortes. *Better Cloud Storage Usability Through Name Space Virtualization*. In Proceedings of the 6th IEEE/ACM International Conference on Utility and Cloud Computing, Dresden, Germany, December 2013. (Cited on page 5.)

[Baru 1998] Chaitanya Baru, Reagan Moore, Arcot Rajasekar and Michael Wan. *The SDSC storage resource broker*. In Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, page 5. IBM Press, 1998. (Cited on page 127.)

[Bent 2009] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte and Meghan Wingate. *PLFS: a checkpoint filesystem for parallel applications*. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, page 21. ACM, 2009. (Cited on pages 52 and 128.)

[Boggs 2006] Adam Boggs, Jason Cope, Sean McCreary, Michael Oberg, Henry M Tufo, Theron Voran and Matthew Woitaszek. *Improving cluster management with scalable filesystems*. In Proceedings of the 7th LCI International Conference on Linux Clusters: The HPC Revolution, Norman, Oklahoma, 2006. (Cited on page 123.)

[Bonwick 2003] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee and Mark Shellenbaum. *The zettabyte file system*. In Proc. of the 2nd Usenix Conference on File and Storage Technologies, 2003. (Cited on page 122.)

[Borthakur 2007] Dhruba Borthakur. *The hadoop distributed file system: Architecture and design*. Web document: `http://hadoop.apache.org/docs/r0.18.0/hdfs_design.html`, 2007. (Cited on page 14.)

[Boulton 2005] Clint Boulton. *IBM Dominates Supercomputer List*. Web document: `http://www.serverwatch.com/news/article.php/3515476`, June 2005. (Cited on page 123.)

[Bovet 2008] Daniel P Bovet and Marco Cesati. Understanding the linux kernel, chapitre 12, The Virtual Filesystem. O'Reilly Media, 2008. (Cited on pages 35 and 113.)

[Box 2013] *Box*. Web page: `http://www.box.com/`, 2013. (Cited on pages 101 and 129.)

[Braam 2007] Peter J. Braam. *Lustre file system: High-performance storage architecture and scalable file system (white paper)*. Rapport technique, Sun Microsystems, Inc., 2007. (Cited on pages 12, 14, 89, 123 and 124.)

[Brightwell 2002] Ron Brightwell, William Lawry, Arthur B MacCabe and Rolf Riesen. *Portals 3.0: Protocol Building Blocks for Low Overhead Communication.* In ipdps, volume 2, page 268, 2002. (Cited on page 123.)

[Cope 2005] Jason Cope, Michael Oberg, Henry M Tufo and Matthew Woitaszek. *Shared parallel filesystems in heterogeneous linux multi-cluster environments*. In Proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution, 2005. (Cited on pages 122, 124 and 128.)

[Dawidek 2008] Pawel Jakub Dawidek. *POSIX file system test suite*. Web page: `http://www.tuxera.com/community/posix-test-suite/`, August 2008. (Cited on page 60.)

[DEI 2008] *DEISA Infrastructure: global file system*. Web document: `http://www.deisa.eu/services/infrastructure#globalfs`, 2008. (Cited on page 124.)

[Deniel 2007] Philippe Deniel, Thomas Leibovici and Jacques-Charles Lafoucrière. *GANESHA, a multi-usage with large cache NFSv4 server*. In Linux Symposium, page 113, 2007. (Cited on page 122.)

[Devulapalli 2007] Ananth Devulapalli and PW Ohio. *File creation strategies in a distributed metadata file system*. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–10. IEEE, 2007. (Cited on pages 51 and 128.)

[Douceur 2006] John R Douceur and Jon Howell. *Distributed directory service in the Farsite file system*. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 321–334. USENIX Association, 2006. (Cited on pages 14 and 83.)

[Dro 2013] *Dropbox*. Web page: `http://www.dropbox.com`, 2013. (Cited on pages 101 and 129.)

[Eisler 2007] Michael Eisler, Peter Corbett, Michael Kazar, Daniel S Nydick and J Christopher Wagner. *Data ONTAP GX: A Scalable Storage Cluster.* In FAST, volume 7, pages 23–23, 2007. (Cited on pages 125 and 127.)

[Erl 2013] *Erlang/OTP*. Web page: `http://www.erlang.org`, 2013. (Cited on pages 25, 61, 73 and 83.)

[Factor 2005] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh and Julian Satran. *Object storage: The future building block for storage systems*. In Local to Global Data Interoperability-Challenges and Technologies, 2005, pages 119–123. IEEE, 2005. (Cited on page 12.)

[Frings 2009] Wolfgang Frings, Felix Wolf and Ventsislav Petkov. *Scalable massively parallel I/O to task-local files*. In High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on, pages 1–11. IEEE, 2009. (Cited on pages 52 and 128.)

[FSE 2012] *File System Events Programming Guide*. Web page: `https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/FSEvents_ProgGuide/`, December 2012. (Cited on page 102.)

[Fuhrmann 2006] Patrick Fuhrmann and Volker Gülzow. *dCache, storage system for the future*. In Euro-Par 2006 Parallel Processing, pages 1106–1113. Springer, 2006. (Cited on page 123.)

[Ganger 1994] Gregory R Ganger and Yale N Patt. *Metadata update performance in file systems*. In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, page 5. USENIX Association, 1994. (Cited on page 122.)

[Ghemawat 2003] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. *The Google file system*. In ACM SIGOPS Operating Systems Review, volume 37, pages 29–43. ACM, 2003. (Cited on pages 13 and 14.)

[Hedges 2010] Richard Hedges, Keith Fitzgerald, Mark Gary and D Marc Stearman. *Comparison of leading parallel NAS file systems on commodity hardware*. In Petascale Data Storage Workshop, volume 2010, 2010. (Cited on pages 124 and 128.)

[Hildebrand 2005] Dean Hildebrand and Peter Honeyman. *Exporting storage systems in a scalable manner with pNFS*. In Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on, pages 18–27. IEEE, 2005. (Cited on page 124.)

[Ior 2013] *IOR HPC Benchmark*. Web site: `http://sourceforge.net/projects/ior-sio/`, 2013. (Cited on pages 78 and 83.)

[Joukov 2007] Nikolai Joukov, Arun M Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger and Erez Zadok. *Raif: Redundant array of independent filesystems*. In Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on, pages 199–214. IEEE, 2007. (Cited on pages 126 and 127.)

[Jus 2013] *JustCloud*. Web page: `http://www.justcloud.com`, 2013. (Cited on pages 101 and 129.)

[Klosterman 2002] Andrew J Klosterman and Gregory Ganger. *Cuckoo: layered clustering for NFS*. Rapport technique, DTIC Document, 2002. (Cited on page 127.)

[Kunkel 2007] Julian M Kunkel and Thomas Ludwig. *Performance evaluation of the PVFS2 architecture*. In Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on, pages 509–516. IEEE, 2007. (Cited on page 128.)

[Lab 2007] *Personal communication with Jesús Labarta, Director of the Computer Science Department at BSC*, 2007. (Cited on page 52.)

[Lin 2004] Chao-Kuo Lin. *DivergeFS file system for Plan 9*. Web page: `http://www.cs.bell-labs.com/wiki/plan9/divergefs/`, 2004. (Cited on page 126.)

[Love 2005] Robert Love. *Kernel korner: Intro to inotify*. Linux Journal, vol. 2005, no. 139, page 8, 2005. (Cited on pages 102 and 115.)

[Mesnier 2003] Mike Mesnier, Gregory R Ganger and Erik Riedel. *Object-based storage*. Communications Magazine, IEEE, vol. 41, no. 8, pages 84–90, 2003. (Cited on page 11.)

[Met 2004] *Metarates*. Web page: http://www.cisl.ucar.edu/css/software/metarates/, 2004. (Cited on pages 53 and 78.)

[Mkrtchyan 2006] Tigran Mkrtchyan, Patrick Fuhrmann and Martin Gasthuber. *Chimera-a new, fast, extensible and Grid enabled namespace service*. In Proceedings of CHEP 2006, February 2006. (Cited on page 123.)

[Morrey III 2003] Charles B Morrey III and Dirk Grunwald. *Peabody: The time travelling disk*. In Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on, pages 241–253. IEEE, 2003. (Cited on pages 11 and 127.)

[MSDN 2012] MSDN. *File System Minifilter Drivers*. Web document: http://msdn.microsoft.com/en-us/library/windows/hardware/ff540402.aspx, 2012. (Cited on pages 25 and 110.)

[MSDN 2013] MSDN. *Desktop App Development Documentation: Directory Management Functions*. Online document: http://msdn.microsoft.com/en-us/library/windows/desktop/aa363946.aspx, 2013. (Cited on page 102.)

[Muller 1991] Keith Muller and Joseph Pasquale. *A high performance multi-structured file system design*. In ACM SIGOPS Operating Systems Review, volume 25, pages 56–67. ACM, 1991. (Cited on page 121.)

[Nagar 2006] Rajeev Nagar and Tony Mason. Windows NT file system internals (OSR classic reprints). Osr Press, 2006. (Cited on page 110.)

[Narayanan 2008] Dushyanth Narayanan, Austin Donnelly and Antony Rowstron. *Write off-loading: Practical power management for enterprise storage*. ACM Transactions on Storage (TOS), vol. 4, no. 3, page 10, 2008. (Cited on page 127.)

[Nelson 2006] Jay Nelson. *Concurrent caching*. In Proceedings of the 2006 ACM SIGPLAN workshop on Erlang, pages 32–38. ACM, 2006. (Cited on pages 73 and 83.)

[Oliveira 2007] Fabio Oliveira, Gorka Guardiola, Jay A Patel and Eric V Hensbergen. *Blutopia: Stackable storage for cluster management*. In Cluster Computing, 2007 IEEE International Conference on, pages 293–302. IEEE, 2007. (Cited on page 126.)

[Olson 1993] Michael A Olson. *The Design and Implementation of the Inversion File System.* In USENIX Winter, pages 205–218, 1993. (Cited on page 122.)

[OSR 2010] *Getting Away From It All: The Isolation Driver (Part I)*, August 2010. (Cited on page 129.)

[OSR 2011] *Getting Away From It All: The Isolation Driver (Part II)*, January 2011. (Cited on page 129.)

[Padioleau 2003] Yoann Padioleau and Olivier Ridoux. *A logic file system*. In Proceedings of the USENIX 2003 Annual Technical Conference, pages 99–112, June 2003. (Cited on page 123.)

[Patil 2007] Swapnil V Patil, Garth A Gibson, Sam Lang and Milo Polte. *Giga+: scalable directories for shared file systems*. In Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07, pages 26–29. ACM, 2007. (Cited on page 83.)

[Pawlowski 1994] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel and Dave Hitz. *NFS Version 3: Design and Implementation*. In USENIX Summer, pages 137–152. Boston, MA, 1994. (Cited on page 12.)

[Pawlowski 2000] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson and Robert Thurlow. *The NFS version 4 protocol*. In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000), 2000. (Cited on pages 12 and 75.)

[Pendry 1995] Jan-Simon Pendry and Marshall Kirk McKusick. *Union mounts in 4.4 BSD-lite*. In Proceedings of the 1995 USENIX Technical Conference (TCON95), 1995. (Cited on pages 15, 125 and 127.)

[Piernas 2002] Juan Piernas, Toni Cortes and José M García. *DualFS: a new journaling file system without meta-data duplication*. In Proceedings of the 16th international conference on Supercomputing, pages 137–146. ACM, 2002. (Cited on page 121.)

[Pike 1990] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey*et al. Plan 9 from bell labs*. In Proceedings of the summer 1990 UKUUG Conference, pages 1–9, 1990. (Cited on page 15.)

[Pike 1992] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey and Phil Winterbottom. *The use of name spaces in Plan 9*. In Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring, pages 1–5. ACM, 1992. (Cited on pages 15 and 125.)

[PVF 2003] *Parallel Virtual File System Version 2*. Web document: `http://www.pvfs.org/cvs/pvfs-2-8-branch-docs/doc//pvfs2-guide.pdf`, September 2003. (Cited on pages 13, 14, 123 and 124.)

[Rodeh 2003] Ohad Rodeh and Avi Teperman. *zFS-a scalable distributed file system using object disks*. In Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on, pages 207–218. IEEE, 2003. (Cited on page 12.)

[Rosenblum 1992] Mendel Rosenblum and John K Ousterhout. *The design and implementation of a log-structured file system*. ACM Transactions on Computer Systems (TOCS), vol. 10, no. 1, pages 26–52, 1992. (Cited on page 121.)

[Ross 2000] Robert B Ross, Rajeev Thakur*et al.* *PVFS: A parallel file system for Linux clusters*. In in Proceedings of the 4th Annual Linux Showcase and Conference, pages 391–430, 2000. (Cited on pages 123 and 124.)

[Russinovich 2009] Mark E Russinovich, David A Solomon and Alex Ionescu. I/O system, chapitre 7. O'Reilly, 2009. (Cited on page 109.)

[Schmuck 2002] Frank B Schmuck and Roger L Haskin. *GPFS: A Shared-Disk File System for Large Computing Clusters.* In FAST, volume 2, page 19, 2002. (Cited on pages 14, 52, 53, 55, 123 and 124.)

[Sebepou 2008] Zoe Sebepou, Kostas Magoutis, Manolis Marazakis and Angelos Bilas. *A Comparative Experimental Study of Parallel File Systems for Large-Scale Data Processing*. LASCO, vol. 8, pages 1–10, 2008. (Cited on pages 74, 75, 125 and 128.)

[Stein 2007] Lex Stein, David Holland, Margo Seltzer and Zheng Zhang. *Can a file system virtualize processors?* In Proceedings of the 1st ACM EuroSys Workshop on System-level Virtualization, 2007. (Cited on page 123.)

[Sug 2013] *SugarSync*. Web page: `http://www.sugarsync.com`, 2013. (Cited on pages 101 and 129.)

[Szeredi 2005] Miklos Szeredi. *FUSE: Filesystem in USErspace*. Web page: `http://www.fuse.org`, 2005. (Cited on pages 25, 61, 64, 77, 113 and 129.)

[Tang 2003] Rongfeng Tang, Dan Meng and Sining Wu. *Optimized implementation of extendible hashing to support large file system directory*. In Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on, pages 452–455. IEEE, 2003. (Cited on page 122.)

[Thain 2005] Douglas Thain, Sander Klous, Justin Wozniak, Paul Brenner, Aaron Striegel and Jesus Izaguirre. *Separating abstractions from resources in a tactical storage system*. In Proceedings of the 2005 ACM/IEEE conference on Supercomputing, page 55. IEEE Computer Society, 2005. (Cited on page 13.)

[Wan 2003] Michael Wan, Arcot Rajasekar, Reagan Moore and Phil Andrews. *A simple mass storage system for the SRB data grid*. In Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on, pages 20–25. IEEE, 2003. (Cited on pages 127 and 129.)

[Wang 1993] Randolph Y Wang and Thomas E Anderson. *xFS: A wide area mass storage file system*. In Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on, pages 71–78. IEEE, 1993. (Cited on page 14.)

[Wang 2004] Feng Wang, Scott A Brandt, Ethan L Miller and Darrell DE Long. *OBFS: A File System for Object-Based Storage Devices.* In MSST, volume 4, pages 283–300. Citeseer, 2004. (Cited on page 122.)

[Warfield 2005] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach and Steven Hand. *Parallax: Managing Storage for a Million Machines.* In HotOS, 2005. (Cited on pages 126 and 127.)

[Weil 2004] Sage A Weil, Kristal T Pollack, Scott A Brandt and Ethan L Miller. *Dynamic metadata management for petabyte-scale file systems*. In Proceedings of the 2004 ACM/IEEE conference on Supercomputing, page 4. IEEE Computer Society, 2004. (Cited on page 83.)

[Weil 2006] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long and Carlos Maltzahn. *Ceph: A scalable, high-performance distributed file system*. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 307–320. USENIX Association, 2006. (Cited on pages 12 and 124.)

[Welch 2008] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka and Bin Zhou. *Scalable Performance of the Panasas Parallel File System.* In FAST, volume 8, pages 1–17, 2008. (Cited on page 125.)

[Wright 2006] Charles P Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P Quigley, Erez Zadok and Mohammad Nayyer Zubair. *Versatility and unix semantics in namespace unification*. ACM Transactions on Storage (TOS), vol. 2, no. 1, pages 74–105, 2006. (Cited on pages 15, 125 and 127.)

[Yamamoto 2000] Tetsuo Yamamoto, Makoto Matsushita and Katsuro Inoue. *Accumulative versioning file system Moraine and its application to metrics environment MAME*. ACM SIGSOFT Software Engineering Notes, vol. 25, no. 6, pages 80–87, 2000. (Cited on page 126.)

[Yu 2007] Weikuan Yu, Jeffrey Vetter, R Shane Canon and Song Jiang. *Exploiting lustre file joining for effective collective io*. In Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on, pages 267–274. IEEE, 2007. (Cited on pages 51 and 128.)

[Zadok 2006] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu and Charles P Wright. *On incremental file system development*. ACM Transactions on Storage (TOS), vol. 2, no. 2, pages 161–196, 2006. (Cited on pages 16, 113, 125 and 127.)

[Zhang 2007] Zhihui Zhang and Kanad Ghose. *hFS: A hybrid file system prototype for improving small file and metadata performance*. In ACM SIGOPS Operating Systems Review, volume 41, pages 175–187. ACM, 2007. (Cited on page 121.)

Index