Albano GRP 1.0.0-2015
(Revision 2015-04-08)

Forensics and File Recovery on the Lustre Distributed File System

Sponsor

The Department of Electrical, Computer, Software & Systems Engineering at Embry-Riddle Aeronautical University

Released April 8, 2015

Justin Albano
Master of Software Engineering

Abstract: With the advent of large-scale distributed systems and cloud computing, the file system of networking infrastructure has become a leading performance bottleneck. In order to overcome this challenge, researchers have developed distributed file systems that can provide petabytes of aggregate file I/O and petabytes of cumulative storage; foremost among these, the Lustre file system. Since its advent in 1999 at Carnegie Mellon University, Lustre has gained the interest and financial support of some of the largest technology entities, including Intel, Oracle, Seagate, and Oak Ridge National Laboratories, and is used in over 60% of the TOP100 supercomputers in the world. Despite this ubiquity, little public research has been conducted on the forensics of the Lustre file system. This paper is intended to fill this gap and focuses on the recovery of deleted files in a working Lustre file system. The approach taken by this paper is both practical and theoretical, where the Lustre components are studied in their distributed operating environment, and concurrently, the Lustre source code is inspected in order to find leads that allow an external tool to recover files distributed across a Lustre cluster. Upon completion of this research, it was found that full file recovery is a real possibility, although no definitive answer has been found at this time. Further research must be completed in order to take the findings of this research and put them into action, making the recovery of Lustre files a real possibility in the realm of file system forensics.

Department of Electrical, Computer, Software & Systems Engineering
Graduate Research Project, Spring 2015

Embry-Riddle Aeronautical University
600 S. Clyde Morris Blvd.
Daytona Beach, FL 32114

Revision History

Date	Reason for Change	Version
Mar. 19, 2015	Initial draft of document	0.1.0
Apr. 5, 2015	Completion of conceptual solution description	0.2.0
Apr. 7, 2015	Completion of solution architecture	0.3.0

Signatures & Official Release

The following signatures denote the official release of this document as part of the Graduate Research Project conducted by Justin Albano as part of the requirements for completion of the Masters in Software Engineering degree through the department of Electrical, Computer, Software, and Systems Engineering at Embry-Riddle Aeronautical University in the Spring 2015 semester. Such signatures, apart from the signature of Justin Albano, do not denote any responsibility of the signing party for the content contained within this document and does not serve as an acceptance of the accuracy or veracity of the content contained within this document.

	Name	Signature	Date
Student	Mr. Justin Albano		
GRP Advisor	Dr. Remzi Seker		
MSE Coordinator / Department Chair	Dr. Remzi Seker		

Contents

1. Introduction

With the explosion of large-scale distributed applications and cloud networks, the focus of a large portion of software application development has shifted from geographically local applications to disperse, service-oriented development. At the heart of this movement is the need to develop the underlying infrastructure that can support this evolution. In particular, one of the fields of study that has paid great dividends in both performance and scalability is that of the network file system. While network file systems are not a new invention, the paradigm on which these file systems are based has changed drastically over the past few decades.

The invention of network file systems, foremost among these the Network File System (NFS), dates back to around 1976, when Digital Equipment Corporation (DEC) created the File Access Listener (FAL) [1]. This precursor to the NFS set the stage for moving the technology of software infrastructure from localized machines to geographically disperse collections of network nodes. While inventions such as the NFS have gone a long way in creating the foundation on which our modern world of connectivity is situated, there were many shortcomings with these file systems. Primary among these is the lack of focus on high throughput and large-scale storage.

In order to solve these issues, large-scale, distributed file systems were created. One of the most popular and widely supported is the Lustre File System. In 1999, Peter Braam created the forerunner to the Lustre file system architecture during a research project at Carnegie Mellon University [2]. In 2001, Braam started Cluster File Systems (CFS) and after a series of acquisitions, Open Scalable File Systems (OpenSFS) has become the *de facto* maintainer of the Lustre file system, in consort with companies such as Intel (and Whamcloud, which is now a subsidiary of Intel, Corp.). As of March 2015, Lustre is now on its 2.7 future release version, with plans for exabyte storage capability by 2018 (on behalf of research efforts by the US Department of Energy, DoE) [3].

At the time of writing, the Lustre file system is now used on over 60% of the TOP100 high performance computers (HPCs) in the world, according to [4]. The explosive growth of distributed file systems, such as Lustre, is due in large part to their ability to support aggregate file Input/Output (I/O) rates of terabytes per second, as well as the ability of these file systems to incorporate large, disparate storage options into a single, uniform file system. For example, as an enterprise grows, new storage can easily be added to the file system without changing the interface through which a client of the file system accesses his or her files. Likewise, as file-based distributed technologies, such as Apache Hadoop, continue to grow in popularity, so will file systems that can be support larger I/O rates and larger volumes of persistent storage.

While Lustre continues its substantial growth, research in the corollary fields surrounding the file system have lagged behind. Primary among these deficiencies is research in the area of forensics on large-scale Lustre file systems. In an age of increasing cyberwarfare, cybercrime, and cybersecurity, file system forensics is an essential part of law-enforcement, military intelligence, and security. Apart from these pressing issues, the ability to scan a file system for deleted files or trace a file system to investigate leads of nefarious behavior serves a pragmatic purpose for many enterprises and companies using the Lustre file system.

In order to bridge this gap between Lustre development and Lustre forensic research, the author proposed the topic of Lustre file system forensics, with a focus on the recovery of deleted files, as his Graduate Research Project (GRP) for the Department of Electrical, Computer, Software, and Systems Engineering (ECSSE) at Embry-Riddle Aeronautical University (ERAU)[1]. This proposal was accepted, and advised by Dr. Remzi Seker of the ECSSE department at ERAU in December of 2014.[2] This document serves as the culmination of the research conducted on the proposed research topic, including all background and prerequisite knowledge required to understand the research findings and solutions devised through this research.

---

[1] This GRP was chosen in order to fulfill the required course curriculum for completion of the Master of Software Engineering program in the ECSSE department of ERAU.

[2] This proposal was officially accepted on December 4, 2014. For more information, including the original text of this proposal, see **Appendix A: Original Research Proposal**.

The following section provides an overview of the scope of the research conducted, as well as the constraints and freedoms under which this research was completed. While a singular, definitive solution has not yet been implemented to solve the problem of Lustre file system forensics, it is the opinion of the author that such a solution is possible, from both a theoretical and practical viewpoint. Therefore, it is the hope of the author that this paper, and the research contained within will serve as a jumping-off point for future research into the field of file system forensics on the Lustre file system.

1.1 Scope of Work

Originally, the research proposed by the author for the Lustre file system consisted of research into the proposed architecture and design upgrades detailed in [5], [6], [7], and [8]. Upon further investigation and discussion with Dr. Remzi Seker, a need was discovered for forensic analysis of the Lustre file system. In particular, the recovery of once-deleted files on a large-scale distributed Lustre file system. This need was further corroborated through discussions with a Lustre researcher at Oak Ridge National Laboratory (ORNL). In order to meet this need, original topic of the author's GRP was altered to focus on the forensic analysis of Lustre file systems.

The timeline for this GRP was set to start in January of 2015 (the start of the Spring 2015 semester at ERAU) and complete in accordance with the completion of the Spring 2015 semester in April of 2015. Therefore, work was conducted on this project from January 2015 until mid-April 2015.

In general, few constraints were imposed on the research contained in this document. The main goal of this research is to devise a means or method of recovering deleted files from a Lustre file system. While not expressly stated, it is the goal of the author to create a tool or similar automated method to recover files from a live Lustre file system (a Lustre file system that was online and still in use in its intended environment of operation, as opposed to a file system that is brought offline for static, forensic analysis). Due mainly to the time constraints of the time period in which this research was conducted, such a tool was not created. Instead, this envisioned tool was used as a mechanism to drive the research found within this document. Therefore, it is important to bear in mind that all findings and research contained within this document were gathered as a means of creating such a tool.

1.2 Prerequisite Knowledge

This document is written with the intent of explaining the research conducted for the aforementioned GRP to both technical and non-technical readers. Although seemingly incompatible, this paper contains a great deal of technical detail, but it also written in a manner that a non-technical reader can consume and still understand the basic and fundamental concepts behind the research presented. The purpose of this approach is to allow the non-technical reader to grasp the fundamentals of this research, while concurrently allowing a technical reader to further understand the research and appreciate the material to a greater depth, based on his or her technical ability.

Some prerequisites that are assumed for a technical reader of this paper include

- A working knowledge of the C programming language, including its constructs and the common manner in which C source code is written on large scale projects. The Lustre code base is overwhelming written in C and therefore, any examples of Lustre code or examples of Linux source code that interacts with the Lustre file system, will be presented to the reader in its native, C language. Specific examples include function pointers, void pointers, structures (`struct`), and pointer manipulation.
- An understanding of the class hierarchy constructs in Object-Oriented Programming (OOP). These constructs include interfaces, abstract classes, and concrete implementations of abstract base classes (ABCs), as well as how these constructs can be used to present a coherent interface to a client, while at the same time, abstracting the implementation details from the client. Specific examples include accessing an object using its interface, allowing the implementation class to change without affecting the logic of the accessor (also referred to in this paper as indirection).
- A working knowledge of telecommunications and network concepts. For example, it is understood that the reader knows what an Internet Protocol (IP) address is and thus, when an IP is assigned to a node in a network, no further explanation is presented about the details of IP or its address assignment.

- A working knowledge of Linux. A large portion of the installation procedures presented in this paper leverage a working knowledge of Linux and the Command Line Interface (CLI) of Linux. Simple commands may not be explained in the procedures presented; for example, the procedure may simply state that the user should "change directory," rather than instruct the user to execute the `cd` command.

1.3 Structure of Paper

Section 2 of this paper presents the background information required to understand both the problem statement as well as the proposed solution. This foundational information includes a description of the terminology used throughout the Lustre file system, and a brief introduction to the Linux kernel structures and algorithms used by the Lustre file system, as well as the solution presented in this paper. Section 3 describes the process of creating a Lustre cluster to replicate the findings in this paper. This description includes information about setting up the virtualization environment required to host the client and server components of a Lustre file system, as well as the procedures required to configure these Lustre components to communicate with one another.

Section 4 provides the reader with a description of the problem being solved by this paper. This information includes the fundamental abstraction required to characterize both the problem, as well as provide a basis on which the solution is presented. Section 5 provides a detailed description of the solution design suggested by this paper. This section also describes various perspective surrounding the solution, including the differing viewpoints and contexts in which the solution can be viewed. This section heavily relies on an understanding of the fundamental concepts presented in section 2, as well as the basis provided by section 4. Section 6 provides concluding comments about both the solution and the Lustre file system in general, including opinions and closing comments by the author.

The glossary of this document provides brief definitions for the terms commonly used throughout this document. These definitions are not intended to be exhaustive, and the references throughout this document should be used where further, more detailed information about a term or concept are required. The acronyms and abbreviations section provides a complete enumeration of all acronyms, abbreviations, and other shorthand terms used throughout this document. This list is alphabetically order and include any parenthetically defined shorthand terms used throughout this document. The references section contains a complete listing of all references used throughout this document, ordered by used in the text of this paper (where the first source referenced appears first in the list and the last source referenced appears last in the list). Any supplemental comments about the references are also included in this section.

Appendix A presents the reader with the original proposal presented to Dr. Remzi Seker of the ECSSE department at ERAU in order to initiate the research contained in this paper. Appendix B contains a description of solutions that were investigated prior to the completion of the solution presented in section 5. The intent of this section is to provide the reader a view into the thought process of the author while developing the solution contained in this paper, as well as provide information to the reader about solutions that may be infeasible (letting the reader know about solutions that are ultimately dead-ends) or solutions that can be developed through future research. Appendix C describes various pitfalls commonly found throughout researching the Lustre file system, with respect to the Lustre documentation, source code, and client and server software. Appendix D presents the reader with a detailed description of the major issues encountered during research in hopes that the reader may avoid the same issues, or else, overcome these issues with a fresh perspective not afforded to the author.

It is the hope of the author that the culmination of these sections will provide the reader with both breadth and depth of understanding about the Lustre file system, as well as the challenges and hurdled learned by the author during the ensuing research. While many of these sections are not considered essential in a research document, it is the intent of the author to provide the reader with the greatest level of detail and allow the reader to learn from the mistakes of the author, eventually leading to a solid foundation from which future research in the area of Lustre file system forensics (and the Lustre file system in general) can be conducted.

1.4 Notes to the Reader

I take full responsibility for all the information contained within this document. All material and information obtained from external references has been cited accordingly, and all effort has been given to provide the original

author with proper acknowledgement for his or her work. All errors contained within this document are my fault alone, and I take full responsibility for each of them.

1.5 Acknowledgements

I would like to personally thank Dr. Seker of the ECSEE department at ERAU. Without his advisement, patience in answering my questions, and his guidance in both the GRP process, as well as throughout my time at ERAU, this research would not have been possible. I would also like to thank Dr. Oral H. Sarp at ORNL for his patience in answering my questions about Lustre. Although our correspondence was limited, the information you have provided to me during my GRP is invaluable and has gone a long way in aiding my understanding of Lustre and the completion of the research contained in this document.

Joshua 24:15

—J.A.

2. Background

This section contains an overview of the conceptual and technical information required as prerequisite knowledge for understanding the research findings contained within this document. This information includes

- A timeline showing the creation and development of the Lustre file system, as well as the supporting technologies that were integral to the development of Lustre
- A conceptual overview of the Lustre file system, including a brief description of Lustre, the terminology used in the Lustre community, the vernacular used when describing the Lustre file system, the components that make up the file system, and the interactions among these components
- A description of the Linux file system architecture, including a summary of how Linux interacts with the Lustre file system and how the Lustre file system interacts with the standard interface provided by the Linux operating system
- A round-trip description of how files exist on the Lustre file system, including where entities of interest (in the context of the purpose of the research contained within this document) reside and the data of interest passed between the components of the Lustre file system during normal operation
- A brief description of the MapReduce architecture, which is required as a foundation for understanding the solution architecture presented in this paper

While the summaries contained within this section are as detailed as possible, the purpose of this section is not to focus on the technologies and concepts themselves, but rather, to provide the knowledge necessary to familiarize the reader with the terms and concepts required to understand the research contained within this document. Therefore, this section is not exhaustive in its descriptions. When possible, citations are made to sources where further information can be found and where more detailed descriptions of the concepts contained within can be found.

All code citations within this section are included as footnotes with the following accompanying information:

- The revision Secure Hash Algorithm (SHA) for the commit from which the code snippet is referenced; this revision hash is in reference to the Git repository for Lustre, found at [9]
- The file name of the referenced file, relative to the root directory of the Lustre Git repository found at [9]
- The line number, or range of line numbers, which the citation references

Note that the information presented in these citations may not be accurate for any revision other than the one denoted. As development continues on the Lustre file system continues, these snippets may be removed from the Lustre code base. Therefore, the reader should be aware that the precise location of the snippet may be inaccurate for the Lustre code base at the time of reading.

2.1 A Brief History of the Lustre File System

The history of the Lustre file system is a story that reflects many of the major advancements and historical changes that have occurred in the technology sector over the past two decades. Apart from the releases of feature version of the Lustre file system, this history also includes influence from many of the top technology entities throughout the past sixteen years, including Carnegie Mellon University, Sun Microsystem, Oracle, Intel, and Seagate. While this history is not directly related to problem and solutions presented in this paper, the rich story of Lustre since its inception provides the reader with a more holistic view of the file system and deepens the understanding of the environment in which Lustre was conceived and from which it has evolved.

A timeline representing the major events in the history of Lustre, including major version releases and important logistical milestones, is illustrated below in **Figure 1**. Note that the solid lines on the timeline represent official releases (and are adorned by the version number above the description), while dashed lines represent logistical changes that have occurred over the life of Lustre, including changes in ownership and purchasing of intellectual property or other property associated with the development of the Lustre file system.

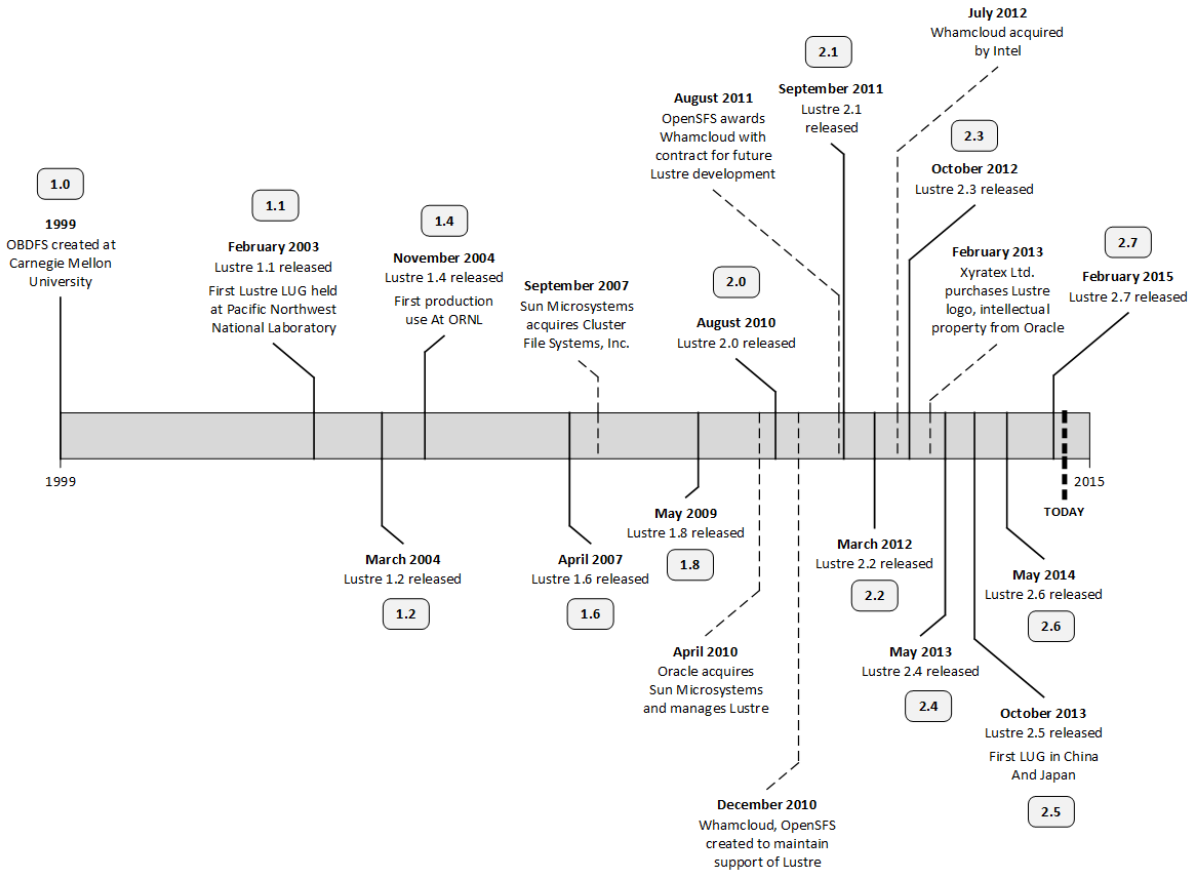

Figure 1. The history of Lustre is not only the story of the Lustre file system itself, but also the story of the technology sector over the past two decades [10], [11], [12].

The Lustre file system, originally called the Object-Based File System (ODBFS), was conceived at Carnegie Mellon University in 1999 and was later acquired by Dr. Peter Braam under his company, CFS [13]. In September of 2007, Sun Microsystems bought CFS, and in turn, acquired the Lustre File System. At the time, this purchase was made in order to better position Sun Microsystems in the HPC market [14]. When Oracle acquired Sun Microsystems in April of 2010, it also become responsible for the development and maintenance of the Lustre file system and all future releases of Lustre [15].

After this acquisition, Oracle announced in December of 2010 that long-term support for Lustre would be ceased [16]. In response to this discontinuation of Lustre support, organizations such as Whamcloud and OpenSFS were created to take on task of supporting future releases of the Lustre file system [17]. The following August, OpenSFS awarded Whamcloud with a contact for future development and feature releases of Lustre, ensuring that Lustre would continue its life for many years to come [18].

In July of 2012, Intel purchased Whamcloud, taking on responsibility for the development of the Lustre file system [19]. In accordance with this transfer of responsibility, OpenSFS transferred its contracts with Whamcloud directly to Intel, who continued to invest in development of Lustre. Following the trend of acquiring Lustre-related resources from Oracle after its failure to continue Lustre development, Xyratex Ltd. acquired the intellectual property and logo for Lustre in February of 2013 [20]. Xyratex, a Seagate affiliate, of whom Dr. Braam joined in late 2010, announced in April of 2014 that it would donate the `lustre.org` domain name to the open source community, adding to the assets maintained by the Lustre community.

As of today, OpenSFS is largely responsible for the distribution of Lustre contracts (of which the original proposal for this paper used as a jumping-off point) while Intel and Seagate share a large portion of the development workload and are responsible for many of the recent feature releases of the file system. While the Lustre code base, along with many of the supplemental assets of Lustre, are maintained by the open source community, Intel and other enterprises have made paid services available for customers; many of these service and deployment plans are a result of increased interest in cloud computing technologies, such as Apache Hadoop.

During this period of drastic ownership changes and possible discontinuation, Lustre published a total of fourteen releases, the earliest at its interception in 1999 and the most recent in March of 2015. At the time of writing, Lustre 1.8 is considered to be one of the most widely used version of Lustre and has served the purpose of bridging the gap between the 1.x and 2.x releases of Lustre.

2.2 An Overview of the Lustre File System

A Lustre file system is a highly distributed cluster of nodes, consisting of two distinct portions: (1) the server portion and (2) the client portion. The server portion of a Lustre cluster is responsible for persisting and storing the files within the file system, managing and controlling the file system, and presenting the client portion of the file system with an interface through which to interact with the files on the Lustre file system. The client portion is conversely responsible for using the server portion interface and providing end-users with access to the Lustre file system. Using this scheme, it is important that the client portion of a Lustre file system appear as though it were a local file system, ensuring that the distribution of the file system is abstracted from the end-user.

To fulfill this responsibility, the Lustre file system divides the server portion into three pairs of components: (1) object storage components, (2) management components, and (3) metadata components. These pairs are further divided into two components classifications that make up each pair: (1) servers, responsible for providing services to the file system and (2) targets, responsible for storing persistent data used by its accompanying server. This results in a total of six component classifications:

1. Object Storage Servers (OSSs)
2. Object Storage Targets (OSTs)
3. Management Servers (MGSs)
4. Management Targets (MGTs)
5. Metadata Servers (MDSs)
6. Metadata Targets (MDTs)

A simple, example Lustre topology is illustrated below in **Figure 2**.

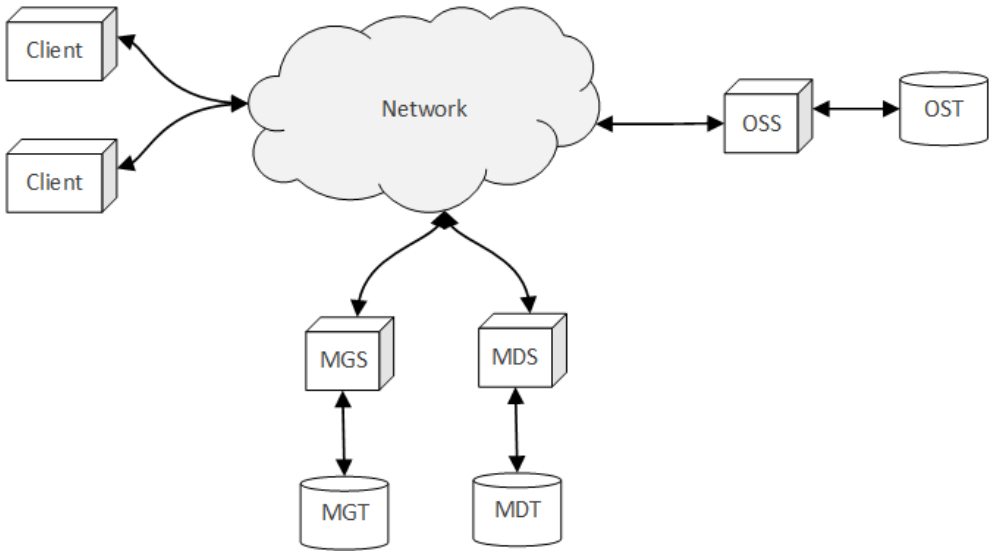

Figure 2. Lustre file systems consist of client nodes and server nodes (composed of OSSs, OSTs, MGSs, MGTs, MDSs, and MDTs) which work in concert to provide the end-user with a seemingly local interface to the distributed file system.

An OSS in the Lustre file system is responsible for providing the services needed by the Lustre cluster to access the persistent storage of its OSTs. Each OSS typically manages and serves two to eight OSTs, with each OST having a maximum of 16 TB of storage [21][1]. Thus, the OSS, paired with its multiple OSTs, make up the backbone of the storage capability of a Lustre file system: During operation, files are broken up into segments, and these segments ultimately reside on the OSTs.

A MDS in a Lustre file system is responsible for managing the metadata about the file system. Lustre is a metadata-based file system, as opposed to a Controlled Replication under Scalable Hashing (CRUSH) file system; thus, the data about the files in the file system are stored in a centralized metadata server (or set of servers). In order for a client in the Lustre file system to find the location of the segments that make up a file, the client must first contact the metadata server to obtain the location of these segments. Once these locations have been obtained, the client can directly contact the nodes containing these segments, and upon gathering the required segments, can reconstruct the file. This process is illustrated below in **Figure 3**.

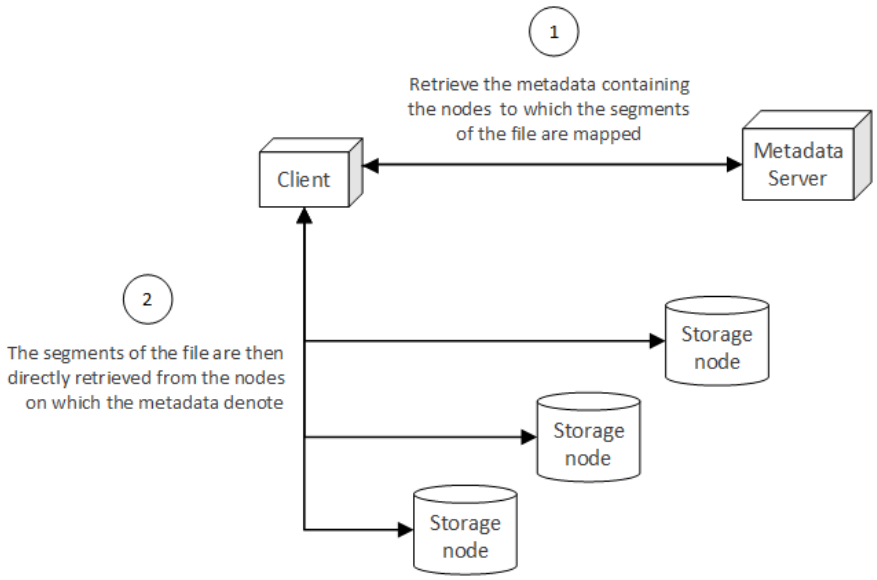

Figure 3. Once the metadata associated with a file has been retrieved from the metadata server, the client then directly accesses the storage nodes of the file system to obtain the segments of a file, ultimately reconstructing the file from these segments.

In the context of a Lustre file system, the MDS acts as the metadata server, storing the persistent metadata information on its accompanying MDT. In Lustre, the locations of file segments amount to a mapping from a segment to a specific OST. Therefore, in order to construct a file distributed across a Lustre file system, a client requests the metadata, or mappings from segment to OST, from the MDS. Once the metadata has been gathered, the client then directly contacts the OSTs and obtains the segments of the file. Upon receiving all of the segments for a file, the client then reconstructs the file and presents the file to the end-user [21], [22].

A CRUSH file system, on the other hand, does not maintain a centralized metadata server. Rather, the location of a file segment in the file system is computed through a deterministic algorithm, as a function of the topology of the network and a rule set [23]. The advantage of a CRUSH file system is that the location of the segments of a file can be calculated by any node in the cluster, removing the need for a centralized metadata service and ultimately resulting in a truly distributed file system. While this is a desirable end-goal for a file system, a more practical

technique (and more widely adopted by many of the most commonly used distributed file systems) is to use a centralized metadata server, as is the case with the Lustre file system[3].

An MGS in a Lustre file system is responsible for maintaining the configuration data associated with the file system. In a working Lustre file system, the target nodes of the cluster will contact the MGS, providing it with relevant configuration data and information about the target, and clients will contact the MGS in order to obtain this data [21]. Similar to the OST and MDT, the MGT acts as the persistent storage device used by the MGS to persist the configuration and usage information passed to it.

While the Lustre cluster depicted in **Figure 2** is a simple example of the server and client portions of Lustre, enterprise Lustre file systems contain many more nodes than illustrated. As of Lustre 2.4, multiple MDTs can be supported by a single file system, but only a single MDS can interface with the MDT on behalf of a client at a given time [21]. Therefore, the metadata portion of the Lustre file system can be configured using a fail-over setup, where a failure in one MDS results in another MDS providing clients with access to the MDTs previously associated with the now-failed MDS. Theoretically, a Lustre file system can support the following [21]:

- 4,096 MDTs
- 8,150 OSTs

While this is a theoretical upper bound, the following has been defined as the practical range for a Lustre file system (not yet tested in a practical environment) [21]:

- 1 primary MDT with 1 MDT backup
- 500 OSSs with a total of 4,000 OSTs[4]

Likewise, the connections between the components in a Lustre cluster are more complex than those presented in **Figure 2**. In reality, the nodes in a Lustre file system are connected through the Lustre Network (LNET). LNET is a networking Application Programming Interface (API) specific to Lustre that provides many of the capabilities required by Lustre to perform large-scale I/O aggregation and transfers over disparate networks.

For example, if some of the nodes are connected to the file system using an InfiniBand network, while others are connected using an Ethernet network, LNET abstracts these differences and provides a consistent, aggregate network interface over which each of the components in the file system interact. This abstract network also provides routers to connect disparate networks to one another, in much the same way as a router in a physical telecommunication network operates. While LNET is a complex interface, the details of its implementation and configuration are generally abstract enough that a simple Lustre file system will not require a great deal of LNET configuration. For more information on LNET, see [21].

With an understanding of the topology and components of a Lustre cluster, it is important to likewise understand how and where files on a Lustre file system exist. The following section delves into the details of the representation of files in a Lustre file system, including the mechanisms used to store the files on OSTs.

2.3 Object Storage & Striping

Lustre is an object-based distributed file system, and therefore divides a file into numerous objects and stores these objects on OSTs through the cluster. In general, an object can be thought of as a logical portion of a file and is analogous to a file segment, as described in the previous section. The main advantage to storing the objects that make up a file on separate OSTs is that the I/O required to retrieve a file from the file system becomes parallelized: Instead of a client reading the objects in a serial fashion from a single storage device (as is the case on a local file

[3] For more information on CRUSH file systems, see [23] and [24].
[4] Within a single Lustre file system, not per OSS

system), the client can simultaneously read the objects from different OSTs in the cluster. Thus, the distribution of objects in the file system closely mimics the behavior of a Redundant Array of Independent Disks (RAID) setup for the disks of a local file system.

Moreover, a Lustre file system can distribute the objects of a file over topologically diverse portions of a network (or an aggregate network, as in the case of LNET), ensuring that the I/O induced by the transfer of the objects is spread over different portions of a network, rather than concentrated or focused on a singular region in a network. In this sense, the distribution of objects not only provides for the parallelization of object retrieval, but also aids in mitigating possible bottlenecks induced in a network by the transfer of a large number of objects.

Using this approach, a Lustre file system matches a RAID 0 arrangement, where data is sequentially distributed, or striped, across a collection of disks [25]. While there is currently work being done on the Lustre file system to support other types of RAID arrangements, particularly RAID 0+1 and RAID 5/6 (logical arrangements, in the same sense that the currently arrangement mimics that of a RAID 0 arrangement for physical disks), Lustre currently supports RAID 0 and is focused on the increased I/O performance of this setup, rather than the redundancy benefits of alternative types of RAID setups[5]. Instead, it is assumed that the disks of a single OST can be configured to use a RAID selection that supports redundancy, ensuring that data on this OST is less likely to be lost, in turn reducing the likelihood of file loss over the entirety of the Lustre file system [21].

In a Lustre file system, striping occurs in a round-robin fashion, where strips are written according to a defined stripe size. When a file is stored, bits are written to an object on an OST until the stripe size is reached, whereupon bits are written to an object on the next OST. This process continues until the entirety of the file has been written to the selected OSTs. If a file is large enough that the stripes of the file have been written to the object on each of the OSTs once, the next stripe is written after the first stripe within the object of the first OST [21]. This process is illustrated below in **Figure 4**. Note that the OSTs on which objects are written need not be adjacent.

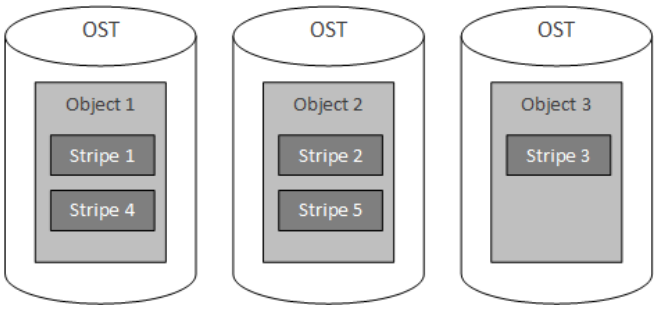

Figure 4. Parts of a file exist on an OST as objects, which contain one or more stripes that are written to the object in a round-robin fashion.

This round-robin scheme is parameterized by two factors: (1) the stripe count and (2) the stripe size. The stripe count denotes the number of objects over which the file will be written (this can be logically thought of as the number of OSTs used to store the file)[6]. In **Figure 4**, the stripe count is set to 3. The stripe size is the number of bits written for each stripe. By default, the stripe count for a Lustre file system is 1, while the stripe size is 1 MB [21].

Note that multiple objects, storing data for different files, exist on a single OST, each with possibly varying stripe counts and stripe sizes. **Figure 5** illustrates a more complex storage configuration, where three files are stored with

[5] For more information on the future plans for supporting other types of logical RAID configurations, refer to Lustre Contract SFS-DEV-003. See also [5], [6], [7], and [8].
[6] Note that the stripe count of a file is *not* the number of stripes that make up the file, but rather, the number of objects that make up the file [21].

different stripe counts and strip sizes. In this example, file A is written with a stripe count of 3 over each of the OSTs, while file B is written with a stripe count of 2 (and larger stripe size than file A and file C) over only two of the OSTs, starting with the right-most OST. Lastly, file C is written with a stripe count of 1, and therefore only exists on a single OST.

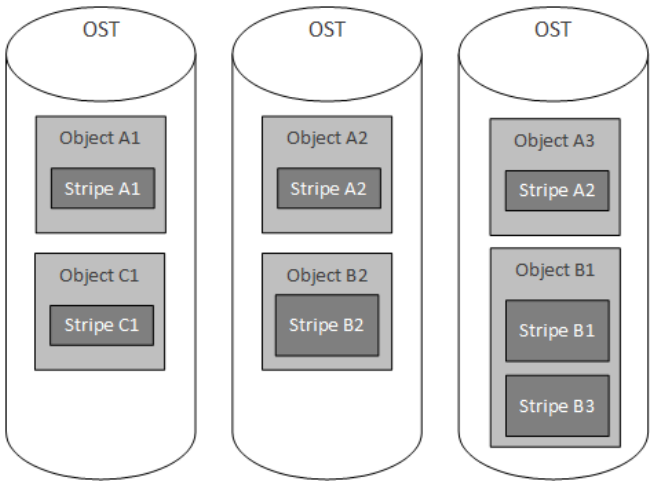

Figure 5. Various objects, representing parts of files, concurrently exist on a single OST, each with possibly varying stripe count and strip size.

With this understanding, it is possible to reconstruct a file given the location of the objects and the stipe size (note that the stripe count is superfluous information, as the number of OSTs found from the locations of the objects is equal to the stripe count). For example, if the objects, object 1 and object 2, for a file are known to exist on OST 1 and OST 2, respectively, and the stripe size is known, then object 1 and object 2 can be retrieved by a client (note that the stripe count is implicitly known). Upon retrieving the objects, object 1 is read for n bits, where n is equal to the stripe size. Once n bits are read from the first object, starting at bit 0 (and it known that there are more bits in the file, such as with knowledge of the file size), then n bits are read from the second object, starting at bit n. If there are more bits to read, n bits are read from first object, starting at bit $2n$, and this process continues until the entire file is read.

Note that the last read from an object may not be equal to n bits. Instead, the last number of bits, m, to read from the object is bound by $[1, n]$ and is equal to

$$m = f - \left\lfloor \frac{f}{sn} \right\rfloor$$

where f is equal to size of the file in bits, s is the number of stripes read so far, n is the stripe size, and $\lfloor x \rfloor$ is the integer floor of x (truncation of the decimal value for a given number x). The resulting process of obtaining the object locations from the MDS and reconstructing a file is illustrated in **Figure 6**. This figure is an expansion of **Figure 3**, and includes the details described in this section.

With a foundational understanding of how a file is stored and reconstructed in a Lustre file system, it is important to understand how clients present a seamless interface to the end-user. This discussion includes the layered architecture of a Lustre file system, a description of the Linux Virtual File System (VFS) and the corresponding Lustre implementation of the VFS, and the internal intricacies of the client, MDS, and OSTs that make up a Lustre cluster.

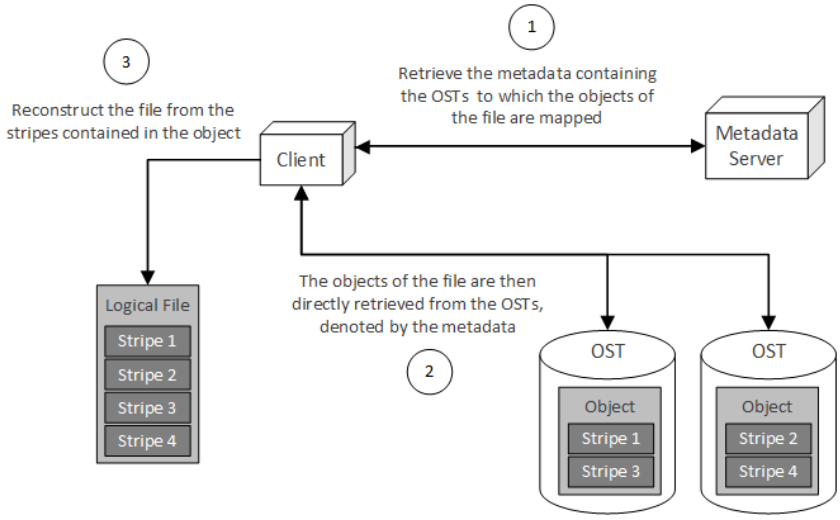

Figure 6. After obtaining the location of the objects from the MDS, the client then retrieves the objects from the OSTs and reconstructs a logical file from the stripes.

2.4 Client Interface to a Lustre File System

Based on an understanding of the scheme presented in **Figure 6**, a Lustre client is responsible for interacting with two main components of the Lustre file system: (1) the MDS, which is used to obtain the metadata about a file, and (2) the OSS, which in turn allows the client to interact with the OST to obtain the objects of which a file is composed. Similar to many distributed systems and communication-focused systems, Lustre uses a layered architecture in order to fulfill these responsibilities. This architecture is illustrated below in **Figure 7**.

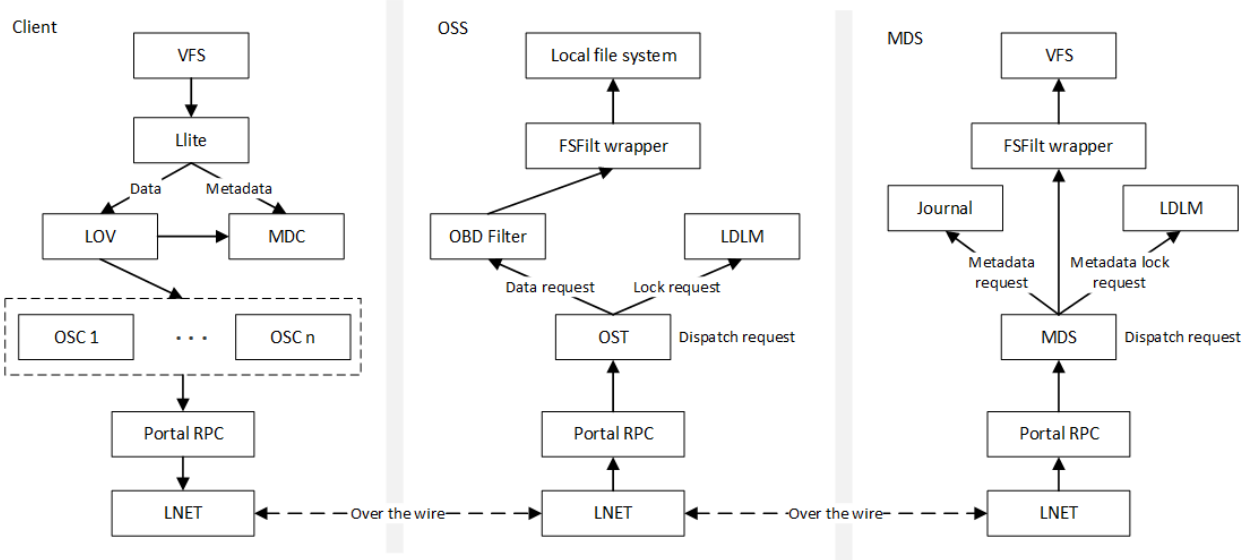

Figure 7. Lustre uses a layered architecture to communicate between clients, OSSs, and MDSs [22].

Starting from the top of the client stack, the Linux VFS interacts with the Lustre implementation of the VFS: Lustre lite (llite). The llite layer then interacts with the Logical Object Volume (LOV) when data needs to be accessed (as in the case of retrieving an object from an OST) or the Metadata Client (MDC) when metadata needs to be accessed (as in the case of retrieving the metadata for a file on the MDS). When the LOV must access metadata, the LOV directly interacts with the MDC in order to obtain the needed metadata. When data is needed, the LOV contacts a series of Object Storage Clients (OSCs). Each client has one OSC per OST in the file system, where each OSC is paired with its corresponding OST [22]. A logical illustration of the MDC and OSCs of a client are depicted in **Figure 8**.

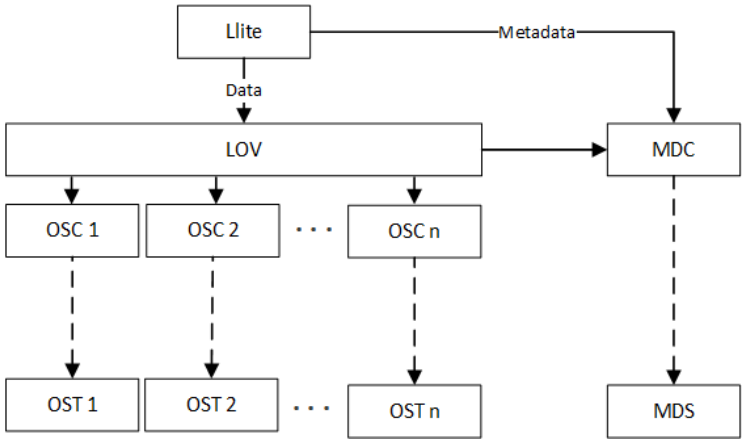

Figure 8. The OSCs of the client stack pair with each of the OSTs in the Lustre file system and the MDC pairs with the MDS running in the Lustre file system.

From the OSC, messages are sent to the other server-side entities in the Lustre cluster through the Portal Remote Procedure Call (RPC) layer. The Portal RPC layer takes in file system-based requests, translates these requests into the appropriate RPC, and sends the resulting RPC over the wire to OSTs and MDSs using LNET [22].

For data requests, these messages are read by LNET layer of the OSSs and transformed back into the RPC made by the client. This RPC is then translated into the appropriate file system request in the Portal RPC layer. From here, requests are divided into two types: (1) data requests dealing with objects existing on the OSTs and (2) lock requests dealing with file locking. In the case of data requests, the request is moved to the Object Based Disk (OBD) Filter, which in turn accesses the File System Filter (FSFilt) wrapper. The FSFilt wrapper is essentially an interface that abstracts the backend file system on the OST[7] from OBD Filter accessing the file system. This wrapper was developed in order to allow multiple types of file systems to be used as the backend storage on each OST without changing the OBD Filter layer to accommodate each backend file system type[8].

In the case of lock requests, the request is forwarded to the Lustre Distributed Lock Manager (LDLM). The LDLM is responsible for maintaining global consistency for file locking among all of the nodes in a Lustre File System. This module of the Lustre file system is explored in greater detail in **Chapter 4** of [22].

[7] Each OST has its own backend file system, which stores the objects of the Lustre file system internally on the OST. This backend file system is separate from the Lustre File System and is often an existing file system, such as the Z File System (ZFS).
[8] This information was published in 2009, and at the time of writing of [22], [22] mentions that future development will likely alter or remove the FSFilt layer as the Lustre Disk File System (ldiskfs) is replaced by other file systems, such as ZFS. The transition of ZFS (and corresponding deprecation of ldiskfs) is corroborated by both [21] and [22], as well as through personal communication with a researcher at ORNL on March 6, 2015.

The MDS stack is similar to the OSS stack, but serves a different purpose. The MDS stacks includes a journaling capability that is used in conjunction with the VFS on the MDS to allow for transaction-based operations. The journaling aspect of the MDS, as well as the MDS in general, is explored in greater detail in **Chapter 6** of [22].

With an understanding of the Lustre file system stack and how the client interacts with the MDS and the OSSs of a Lustre cluster, it is important to delve into how the client presents a Portable Operating System Interface (POSIX) compliant file system interface to the end user. This discussion includes details about the VFS in general, as well as llite in particular. The discussion of the VFS is not exhaustive, but rather, focuses on those aspects of the VFS that are particularly useful in understanding the Lustre file system.

2.5 Linux VFS & Lustre

The Linux VFS[9] is an abstraction of the Linux file system that provides a level of abstraction and indirection for file systems mounted on a Linux machine. In essence, the VFS provides an interface that each file system must implement, ensuring that developers can make system-level file I/O calls, such as `read()` or `write()`, without knowing the internal details of the file system on which the calls are being made. This abstraction is illustrated in **Figure 9**.

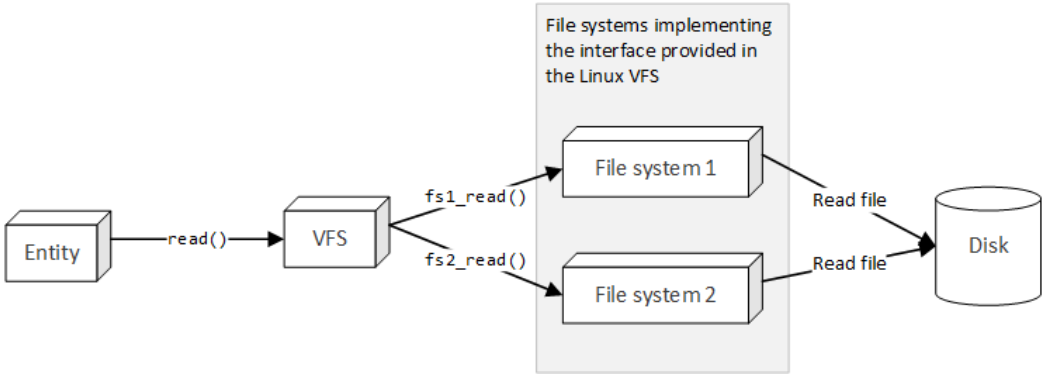

Figure 9. The Linux VFS provides a level of indirection, ensuring that an entity can interact with a mounted file system without knowing the internal details of the file system.

This scheme is very similar to the use of software interfaces in OOP to abstract the implementation details of a class, and as will seen shortly, the VFS is implemented in a pseudo-OOP manner (the VFS is written in a non-OOP language, but uses many of the techniques common in OOP to provide interfaces and structures for implementing file systems to use). In general, a file system implementing the VFS interface can be thought of as a tree of VFS structures. The four primary VFS structures are ([26])

1. Superblock: the root of the tree, representing a mounted file system
2. Inode: represents a single file in the file system
3. Dentry: represents a single directory entry, or a component in a path
4. File: represents an open file associated with a process in the Linux kernel[10]

[9] The Linux VFS is sometimes defined as the Linux Virtual File Switch, rather than the Linux Virtual File System [26].

[10] While the nomenclature can be confusing, the Linux kernel refers to the unopened files of a file system as inodes, while open files associated with an executing process are referred to as files. Additionally, directories are simply treated as files, and therefore, inodes are used to represent directories in the VFS. Likewise, a dentry structure does not represent a file, but rather, a component in a path [26].

In order to provide an abstraction of the operations that each file system is expected to perform, the superblock, inode, dentry, and file data structures contain a pointer to a structure, specifically superblock operations, inode operations, dentry operations, and file operations, respectively, containing pointers to the functions that implement these operations. This indirection scheme is illustrated below in **Figure 10**.

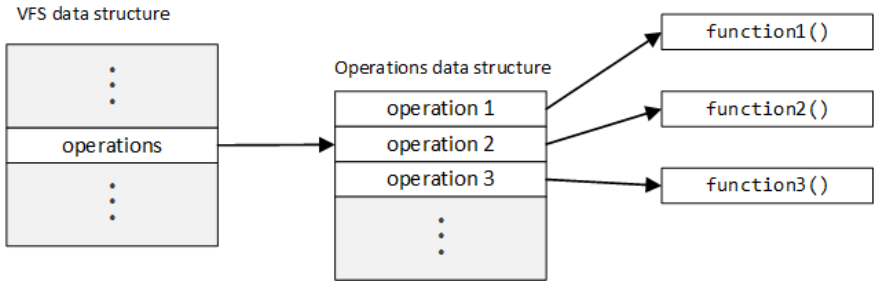

Figure 10. Operation indirection is achieved by storing the pointers to the functions implementing the operations in an operations table data structure.

The operations performed by on the VFS data structures by a file system implementation can be changed by simply creating an operations data structure specific to the file system and setting the pointers within this operations data structure table to functions that are specific to the file system. This variation of file system operations through the operations data structure is illustrated below in **Figure 11**.

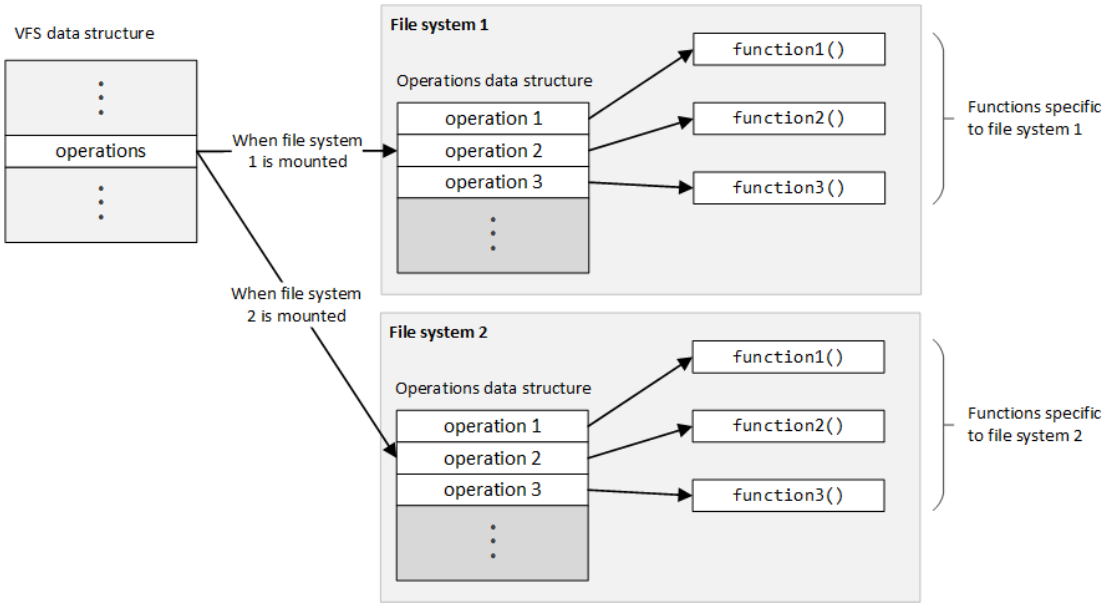

Figure 11. Depending on which file system is mounted, the operations table will vary, ensuring that the functions specific to the mounted file system will be called.

In order to call an operation for the VFS data structure, a client (in the sense of an external entity accessing the data structure, not a client node in a Lustre file system) can simply make a call in the following manner,

```
vfs_data_struct->operations->operation1();
```

where `vfs_data_structure` is a pointer to the VFS data structure, `operations` is a pointer to the operations table for the mounted file system, and `operation1()` is a function specific to the mounted file system pointed to in the operations table (as illustrated in **Figure 11**). Note that a client making this call does not know which function will be called; the function call simply depends on the operations block pointed to through the `operations` pointer, established prior to calling the function. This is the main advantage of the VFS: Clients can operate on the file system in a generic manner, completely blind to the implementation details of the file system and likewise blind to the mechanisms used to call the function (which operation table is mounted at the time of the call).

In order for the function to perform an operation on the data structure making the call, the data structure is passed as an argument into the function:

```
vfs_data_struct->operations->operation1(vfs_data_struct);
```

Passing the data structure as an argument to the function ensures that the function has access to the state of the object on which it is operating. This is directly analogous to OOP, where the state of an object (in the form of the attributes of that object) are directly accessible to the methods of the object through the `this` reference (in a fully qualified call or attribute access; the attributes are also implicitly accessible without the use of the `this` reference) [27]. A class diagram representing the OOP equivalent is illustrated below in **Figure 12**.

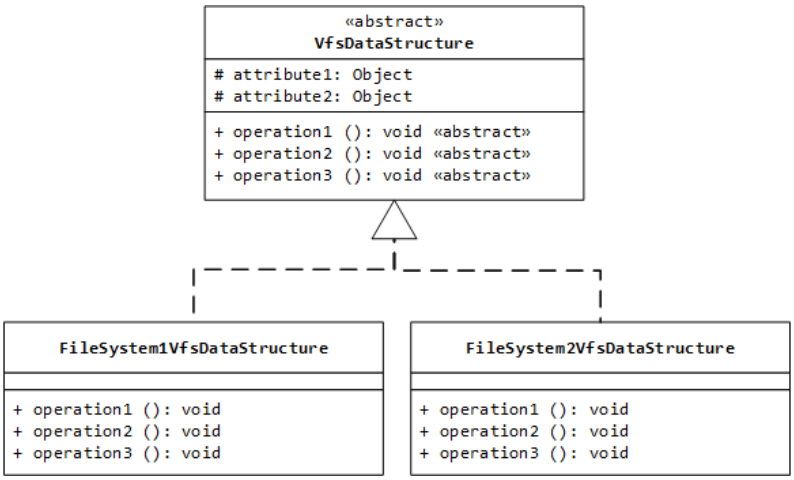

Figure 12. The VFS data structures and operation tables are analogous to class inheritance and method overriding in OOP.

In order to interact with the subclasses in an indirect manner, the following could be performed:

```
// Obtain a VfsDataStructure object from the factory
VfsDataStructure vfs_struct = fileSystemFactory.getVfsDataStructure();

// Execute the operation on the VFS data structure
vfs_struct.operation1();
```

Depending on the type of the object returned from the factory method, the operation will either be performed on the `FileSystem1VfsDataStructure` or the `FileSystem2VfsDataStructure` (in much the say way as the operation called in the non-OOP scheme depends on which operations table is referenced when the VSF data structure is constructed). With this understanding, these concepts can be applied directly to the data structures used in the VFS. The connections between the VFS data structures are illustrated below in **Figure 13**.

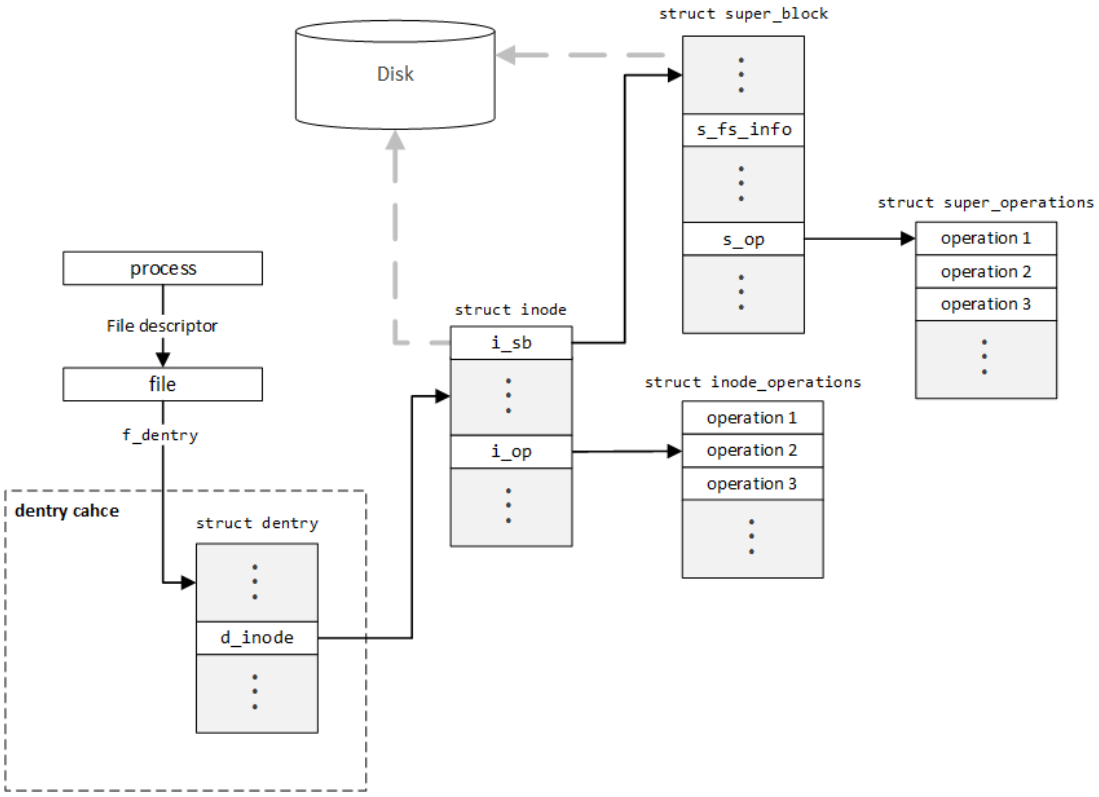

Figure 13. The VFS data structures point to one another, creating a tree structure that represents the file system in memory [22], [28], [29].

As previously stated, the data structure associated with a process in the Linux kernel contains a list of open files (using file descriptors). Each of these file structures points to a dentry structure, which represents the name of the file. This dentry structure then points to an inode, which stores the data required to reconstruct the file. This inode then references the superblock, which represents the root of the file system. This superblock also contains a reference to each of the inodes created for the file system (active inodes). Note that only inodes and the superblock exist in persistent storage (disk): When the system shuts down, all other VFS structures are lost and must be recreated when the kernel loads or the file system is mounted.

While the dentry stores the name of a file (in the sense of a file on disk, not a file in the sense of an open file, as described by the file structure) in the file system, it does not store information about that file. Instead, dentries are used as components in a path, where each component in the path points to a file or directory (as represented by an inode). In essence, dentries form a tree structure, where each dentry references a list of sub-dentries, as well as the parent of the dentry.

For example, assume a sample path of `/home`, where `/home` contains two files: (1) `foo.txt` and (2) `bar.txt`. The first dentry would represent the root directory, or `/`. The next dentry, representing `home`, would have the dentry representing `/` as its parent, and contain two sub-dentries, one dentry representing `foo.txt` and another representing `bar.txt`. This tree structure is illustrated below in **Figure 14**.

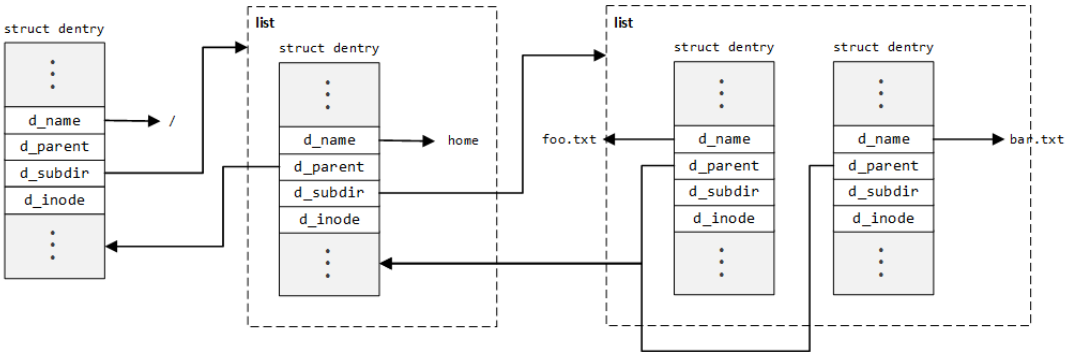

Figure 14. The chaining of dentry structures to represent a path creates a double-linked tree, where each element has a reference to a list of sub-elements as well as a reference to its parent element.

Each of these dentry elements then references an inode element, mapping the component of a path to an inode. For example, in the above, the path `/home/bar.txt` maps to the inode referenced (`d_inode`) by the right-most dentry in **Figure 14**. It is important to note that a single inode can be referenced by multiple dentries. For example, multiple paths in a Linux file system can point to the same file (hard links). Therefore, inodes are not destroyed until all dentries referencing the inode are unattached[11] [30]. Due to the fact that dentries are used so often (for example, if finding files within the same directory), dentry elements are cached in the dentry cache[12]. This significantly reduces the time requires to find the inode associated with a path, since the dentry components for the path do not need to be created each time the path is walked.[13]

With a foundational understanding in the dentry structure established, it is important to explore the inode structure in greater detail. An inode is created each time a file is accessed and stores information about a file, including the permissions of the file, the access times, and, in the case of a local file system, pointers to the locations of the blocks of a file stored on the local hard disk. It is not required that a file system store the location of the file blocks on the local hard disk, and in the case of the Lustre file system, this information is ignored (since the file does not exist on the local hard disk, but rather, exists on a specific set of OSTs in the Lustre cluster).

In order to understand the inode structure, it is important to understand the composition of data on the hard disk. In the case of the forth extended file system (Ext4) used by Linux, all data stored on a hard disk is divided into blocks, or logical chunks of data. A block is the smallest unit that can be stored on the disk and represents the building blocks for a file in the file system [32]. This concept is illustrated below in **Figure 15**.

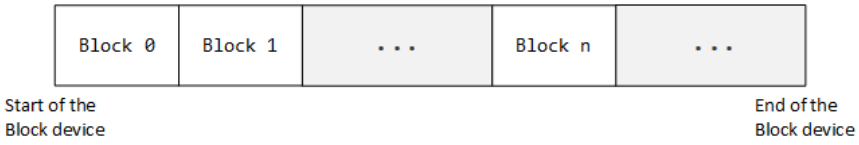

Figure 15. All data on the hard disk is stored in blocks of a fixed size, even if the data within the block does not match the block size.

[11] In fact, the inode structure contains a field for reference counting, `i_count`, that stores the number of current references to the inode; when this count drops to 0, the inode is no longer referenced by any dentries [26].
[12] For more information on the dentry cache, see [26], [28], and [31].
[13] For more information on walking a path and the involvement of dentries in this process, see [28].

Using this knowledge of the layout of the hard disk on which a file is stored, the inode must simply contain pointers to the blocks on the hard disk that make up the file. It is important to note that the blocks that make up a file need not be sequential (and in fact, rarely are). For example, an inode may include pointers to block 100, block 450, and block 777 if the file consists of only three blocks of data. The resulting inode structure is illustrated in **Figure 16**[14].

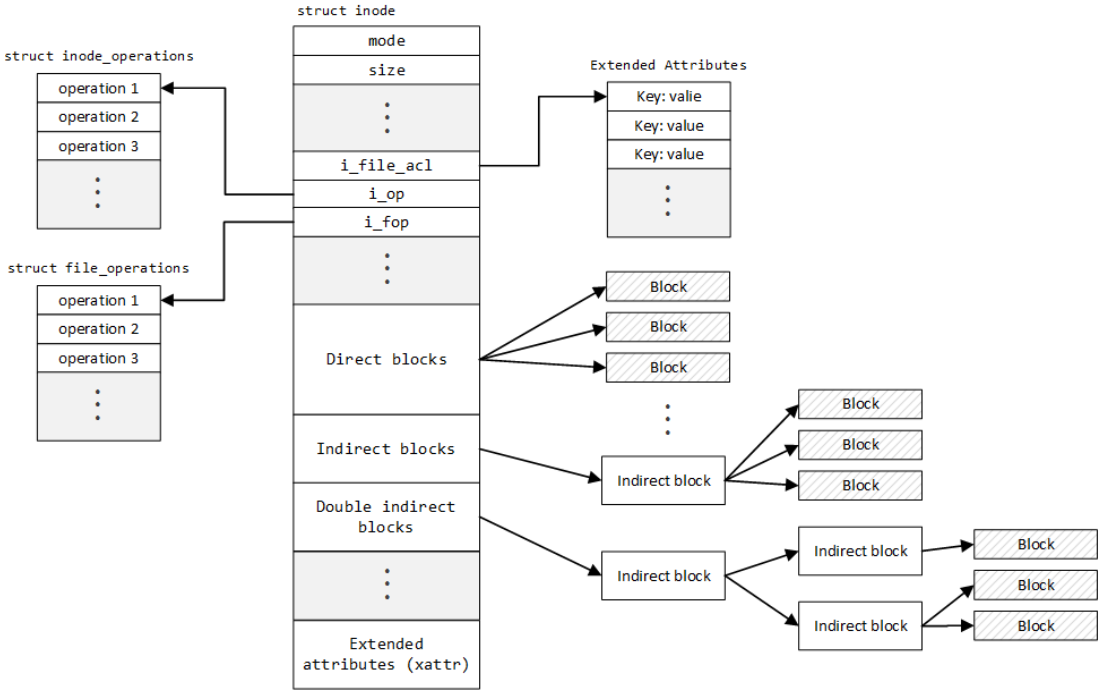

Figure 16. An inode stores the metadata about a file, including its mode (permissions), the pointer to the inode and file operations, data block pointers, and extended attributes [22], [26], [32], [33].

The inode structure contains many direct fields (fields were data is directly stored in the inode), such as the mode, or permissions, of the file, and the size of the file. Note that an inode does not have a name (recall that a name, as contained in a path string, is represented by a dentry), but rather, have a unique identifying number (the inode number[15]). The inode structure also contains pointers to an `inode_operations` structure, containing pointers to functions that operate on an inode, and a `file_operations` structure, containing pointers to functions that operate on files (such as regular files and directories).

Continuing to the data block portion of an inode, there are direct blocks, which are pointers to the data blocks stored on disk that make up the file. If more data blocks are needed to store the file, indirect blocks can be used, where the indirect block pointer points to a block containing pointers to data blocks on disk. Likewise, if more blocks are needed to store a file, double indirect blocks can be used, where the double indirect block stores pointers to blocks, which in turn store pointers to blocks, which ultimately store pointers to the data blocks on disk. In Ext4, an inode also contains triple indirect blocks, following the same scheme as double indirect blocks, but includes one more level of indirection [32][16].

---

[14] This figure is also supported by referencing the source code for Linux kernel 3.19 under `fs/ext4/xattr.c`.

[15] While inodes use an inode number, denoted by the field `i_ino` in the inode structure, to uniquely identify an inode, Lustre uses a different implementation for unique identifiers. This Lustre-specific identifier is called a file ID (FID). For more information on the FID used by Lustre, see **MDS VFS Implementation**.

[16] These triple indirect blocks are not shown in **Figure 16** for the sake of brevity.

Lastly, an inode contains extended attributes (xattr). Extended attributes are key-value pairs that allow implementers of the Linux VFS and end-users who can access the inode of a file to include supplemental attributes in the inode, such security data or other non-essential inode data [32]. In the case of Ext4, extended attributes can be found in two locations: (1) between inodes (when stored on the disk) if the inode does not consume all the space allocated to it, and (2) in a block pointed to by the `i_file_acl` pointer in an Ext4 inode [32].

In the context of the Lustre file system, it is not as important to understand where the extended attributes are located as much as it is to understand that inodes can store supplemental data not included in the VFS inode structure. In essence, if a file system wished to store extra data about a file, it could do so by creating a key for the data and storing the data as the associated value in the extended attributes of the inode. It is also worth reiterating that an inode does not need to provide data for all the fields in the inode structure. As will be seen shortly, this is an important concern for Lustre, which cannot simply reference data blocks on the local disk to obtain the data associated with a file, since the data associated with the file does not exist on the local disk.

The last component in the VFS is the superblock. The superblock gets its name from the fact that it is stored on disk in a predefined location (is a special type of block on the disk), much like the bootloader, and acts as the structure containing the metadata about a mounted file system. For example, the superblock is responsible for containing the dentry associated with the mount point where the file system is mounted [26]. The superblock also contains a pointer, `s_op`, to a structure, `super_operations`, containing function pointers to functions that operate on the file system structures, such as inodes. In this sense, the superblock is higher in the file system hierarchy than inodes, and in turn, are responsible for managing the inodes of the file system. While the superblock is essential to understanding how a file system is mounted, understanding the Lustre implementation does not require in-depth knowledge of this structure, and therefore, it is not discussed in detail in this section. For more information on the superblock data structure in the Linux VFS, see [26], [28], [32], [33], [34], and [35].

2.5.1 llite VFS Implementation

With this foundational understanding of the structures that the Linux VFS is composed of, the llite implementation of the VFS can be understood. As the client implementation of the VFS (the file system seen by the end-user), llite is responsible for presenting a coherent and seamless file system to the end-user, even though the files of the file system, and their accompanying data, are not stored on the local machine (as in Ext4). In order to do this, llite overrides the operations associated with inodes (inode operations) and the superblock (superblock operations). This overriding is illustrated in **Figure 17** (on the following page).

In order to select the llite operation structures for the inodes and superblock, the pointer for the operations structure must be set to the correct operations structure. In the case of the superblock operations, this pointer is set in the

```
static int client_common_fill_super(struct super_block *sb, char *md, char *dt,
                                    struct vfsmount *mnt)[17]
```

function with the following logic:

```
sb->s_op = &lustre_super_operations[18]
```

The `client_common_fill_super` function is an internal function within the Lustre code base that is in turn called by the function

```
int ll_fill_super(struct super_block *sb, struct vfsmount *mnt)[19]
```

This function is then registered as the as the superblock fill function through the function

[17] Revision `355a283fce6998f5b5621adc9697d98d0fb72dfe`, `/lustre/llite/llite_lib.c`, line 169 of [9]
[18] Ibid., line 490
[19] Ibid., line 1007

`lustre_register_client_fill_super`[20]

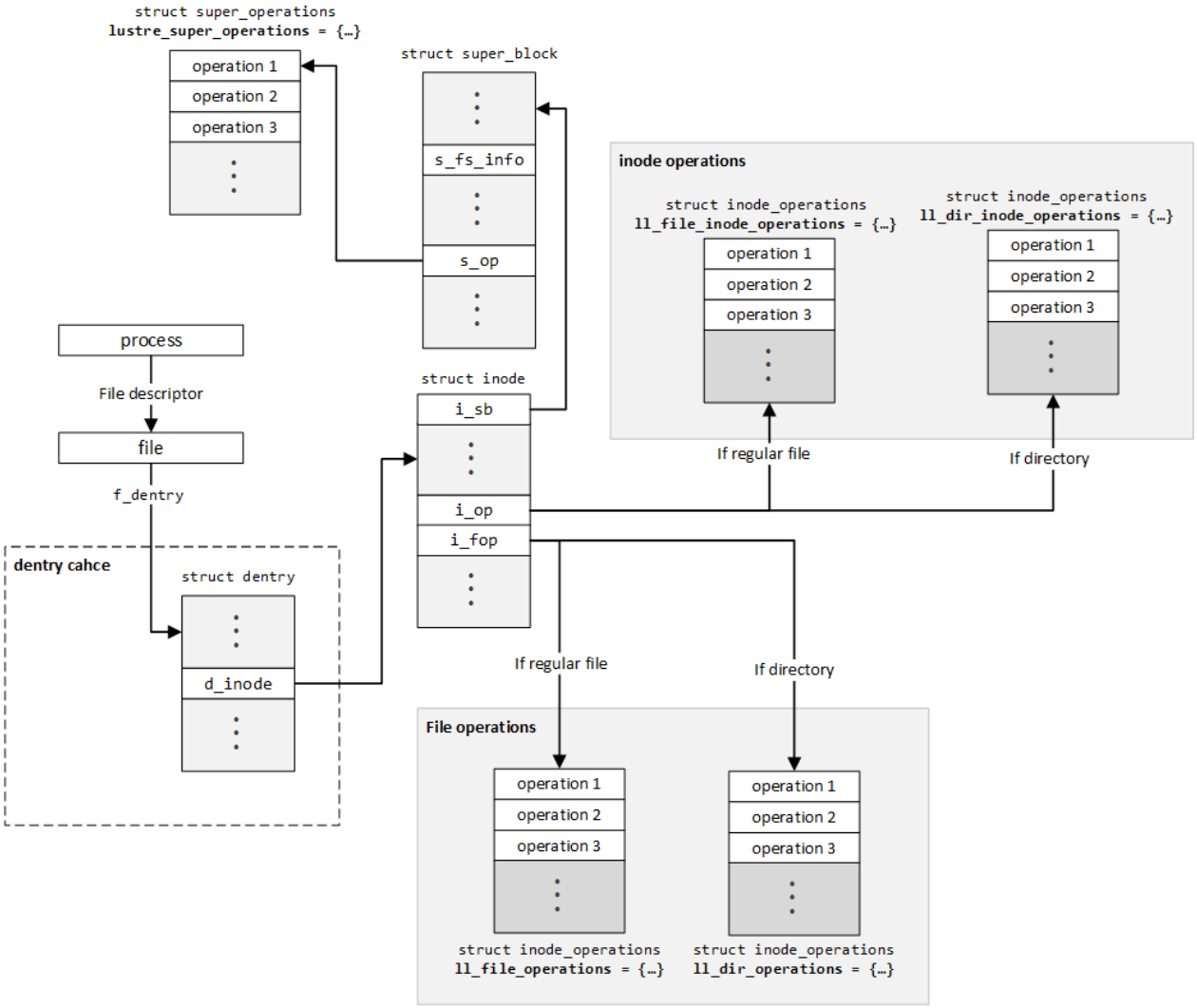

Figure 17. By overriding the operations structures in the Linux VFS, Llite is able to override the functionality of the VFS and provide a coherent file system to the end-user [22].

This function simply sets the `client_fill_super`[21] variable in OBD mount module of llite, which is called within the entry-point function

`static int lustre_fill_super(struct super_block *sb, void *data, int silent)`[22]

[20] Revision `355a283fce6998f5b5621adc9697d98d0fb72dfe`., `/lustre/llite/super25.c`, line 183 of [9]
[21] Ibid., `/lustre/obdclass/obd_mount.c`, line 1357
[22] Ibid., line 1268

This entry point function is called when the Lustre file system is mounted by client. Therefore, the proper superblock operations are set in a chain of calls originating at the function that is called when the Lustre file system is mounted by a client. This execution call chain is illustrated in a more succinct manner in **Figure 18**.

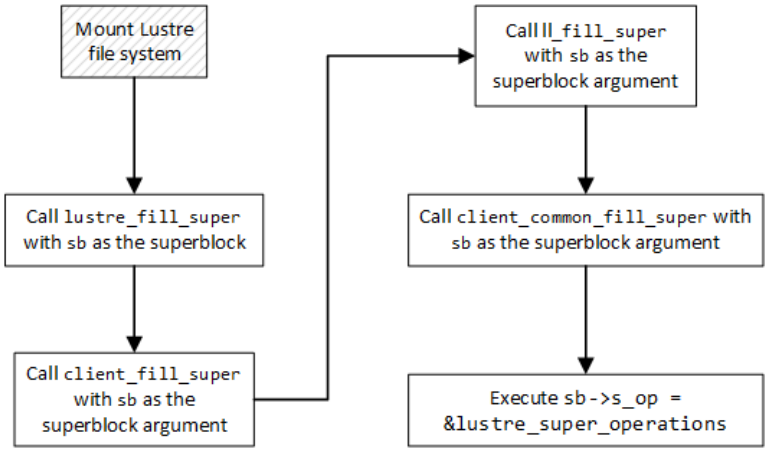

Figure 18. When the Lustre file system is mounted by a client, a call chain is initiated that results in the superblock operations of the superblock being set to the Lustre specific operations.

With `client_fill_super` set through the registration function `lustre_register_client_fill_super`, the call chain in **Figure 18** is established; this call chain is then initiated when the Lustre file system is mounted by a client. In much the same way as its superblock counterpart, the inode and inode file operations are set in the `ll_read_inode2()`[23] function, which is responsible for initializing inodes [22]. The logic used to initialize these operations structures in the inode is[24]

```
if (S_ISREG(inode->i_mode)) {
        // ...
        inode->i_op = &ll_file_inode_operations;
        inode->i_fop = sbi->ll_fop;
        // ...
} else if (S_ISDIR(inode->i_mode)) {
        inode->i_op = &ll_dir_inode_operations;
        inode->i_fop = &ll_dir_operations;
        // ...
} else if (S_ISLNK(inode->i_mode)) {
        inode->i_op = &ll_fast_symlink_inode_operations;
        // ...
} else {
        inode->i_op = &ll_special_inode_operations;
        // ...
}
```

In the first condition, `S_ISREG(inode->i_mode)`, if the inode is a regular file, the inode operations are set to `ll_file_inode_operations` and the inode file operations are set to the file operations specified in the superblock information structure (`sbi`). In the second condition, `S_ISDIR(inode->i_mode)`, if the inode is a directory, the inode operations are set to `ll_dir_inode_operations` and the inode file operations are set to `ll_dir_operations`. In the third condition, if the inode is a link, the inode operations are set to `ll_fast_symlink_inode_operations`

[23] Revision `355a283fce6998f5b5621adc9697d98d0fb72dfe`, `/lustre/llite/llite_lib.c`, line 2004 of [9]
[24] Ibid., line 2034 to line 2054

(fast symbolic link, or symlink, inode operations). Lastly, if the inode is not a regular file, directory, or symlink, the inode operations are set to `ll_special_inode_operations`.

While the specifics of how the correct operation structures are set in the inode and superblock differ, since the superblock and inode are initialized at differing periods in the execution of the file system logic, a simple rule can be used to sum up who is responsible for setting the correct operations structures: The party or function that is creating the new superblock or inode is responsible for setting its correct operations structures [22]. With these operations structures established, llite can present to the end-user a file system that appears to be interacting with the local disk, but instead leverages the services established in the server-side portion of the Lustre cluster.

2.5.2 MDS VFS Implementation

While the llite implementation of the VFS is used on the client nodes of a Lustre file system, there is also a second implementation of the VFS on the MDS of a Lustre cluster. In the case of the client VFS implementation, llite is responsible for creating a coherent interface for accessing files that are not present on the machine on which the client is running; in the case of the MDS implementation, the MDS is responsible for creating a shadow file system, providing information about the mapping of file objects to OSTs in the Lustre cluster.

By shadow file system, we mean to say that the MDS maintains the structure for the file system, just as would a local file system (for example, maintaining an inode for each of the files in the file system, a superblock for the entirety of the file system), but instead of maintaining pointers to the data blocks on the local disk, the inode maintains a mapping of the objects of the file to the OSTs containing those objects. In order to accomplish this mapping, the MDS inodes use the extended attributes of the inode to map each object to the OST contain that object.

This mapping, stored in what is called the layout extended attribute (EA[25]), is stored as an object on the MDT associated with the MDS and identified by the Lustre FID for the file. The FID for a Lustre file is an identifier that is unique among all non-management targets (MDTs and OSTs) in a Lustre file system and is a 64-bit unique sequence, followed by a 32-bit object ID (OID), followed by a 32-bit version number. This 128-bit sequence ensures that files can be referenced in a globally unique manner, rather than managing inode number clashes among the dispersed targets of a Lustre file system [21]. This layout EA object simply contains a key-value mapping of the objects of a file to the OST containing that object, as well as the stripe size and stripe count[26] [21], [22], as illustrated below in **Figure 19**.

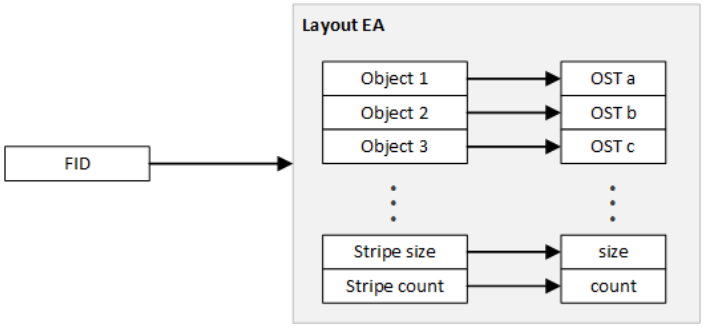

Figure 19. The mapping of objects to their respective OST is maintained in the layout EA object stored on the MDT, as referenced by an FID.

[25] While the Linux vernacular refers to extended attributes as xattr, as do the functions in the operations structure of the VFS inode structure, Lustre refers to the layout extended attribute as the "Layout EA." This document uses the lexicon established for each context and therefore refers to the extended attributes of an inode as xattr (the Linux VFS lexicon) and refers to the layout extended attributes as layout EAs. While a distinction is made in the terminology, both xattr and EA can be used interchangeably, and unless explicitly stated, refer to the same concept.

[26] This information is important when reconstructing the file, where stripes must be pulled from the objects in the OST and reassembled in the correct order.

While the Layout EA exists as an object in-and-of-itself on the MDT, the layout EA can be thought of as a logic extension of the inode structure maintained on the MDS (and persisted on the MDT), as illustrated in **Figure 20**.

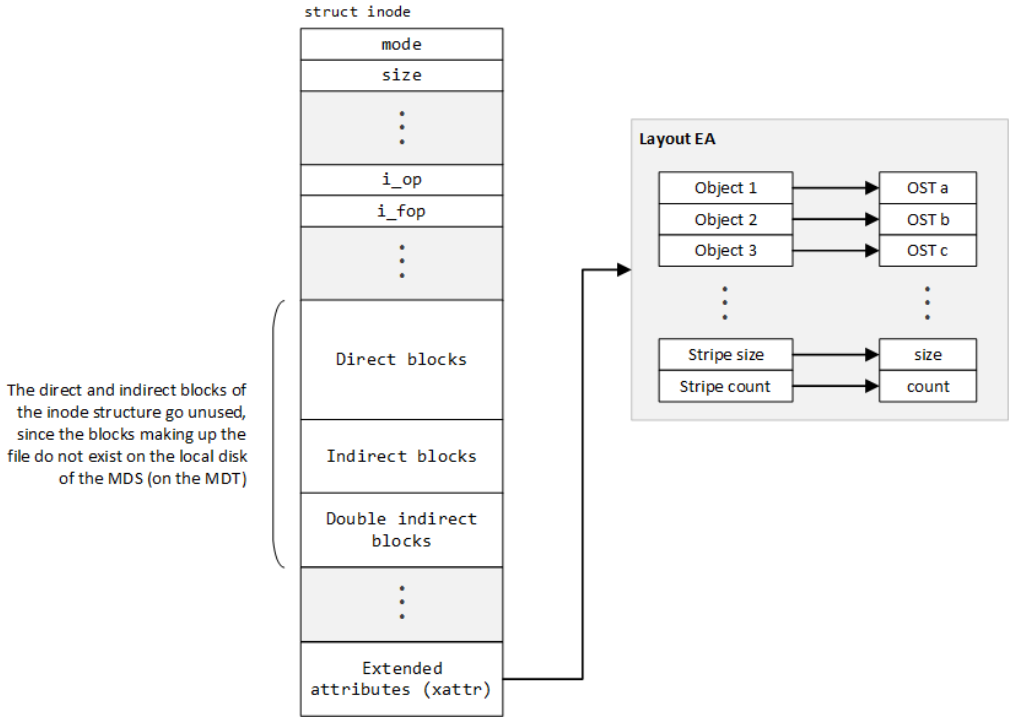

Figure 20. The layout EA can be thought of as a logical part of the inode structure, where the layout EA replaces the need for the direct and indirect block references of the inode.

It is important to reiterate that there is a distinction between the VFS implementation of llite, which represents the client implementation of the VFS for the purpose of presenting a file system that appears to be to the client machine, and the MDS implementation of the file system, which maintains the metadata for the files within the file system in the form of inodes with layout EAs. While both VFS implementations use the same mechanism to alter the functionality of the VFS (using the setting of inode and superblock operation structures to implement the interface presented by the Linux VFS), llite and the MDS VFS implementation are two separate implementations: Although the means and mechanism by which the extension of VFS functionality is achieved is the same (namely, through setting the proper operations structures), the end goal of each implementation is different.

2.6 End-to-End Lustre Operation

With an understanding of the three main portions of logic in the Lustre file system, namely objects and their striping, the llite client implementation of the Linux VFS, and the MDS implementation of the Linux VFS, a full picture can be drawn of how a file exists in the Lustre file system. **Figure 21** illustrates this picture, showing the objects of a single file existing on the OSTs to the right of the graphic; the inodes, which exist in the persistent storage of the MDT and are loaded into memory on the MDS, at the bottom of the graphic; and the clients, using llite, that request access to the file on the left side of the graphic.

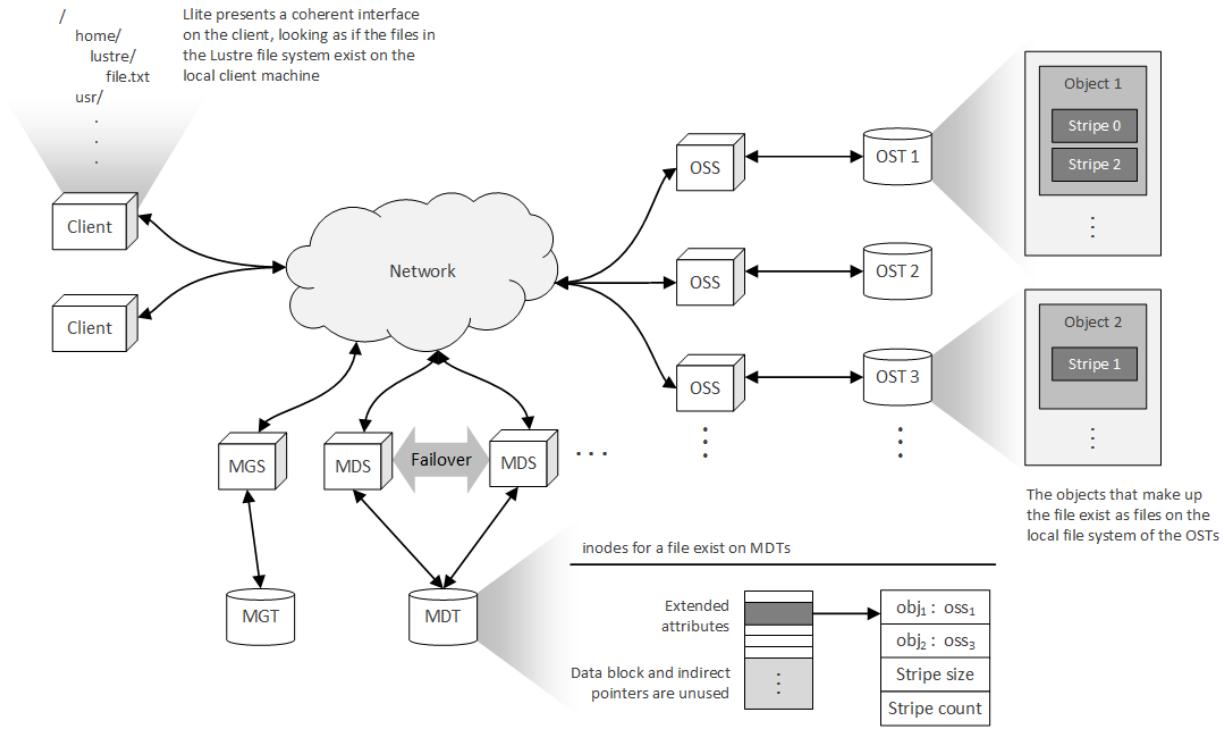

Figure 21. The llite component of the client, the layout EA of the MDS and MDT, and the object storage and striping scheme of the OSSs and OSTs work in concert with one another to provide the appearance of a local file system, while distributing the objects of a file throughout the Lustre cluster.

While this scheme appears complex in visual form, the techniques used by Lustre are simple. For example, when an end-user wishes to a read a file, a `read(..)` request is initiated by the operating system on the client machine. Being that Lustre is mounted as the file system, this read request is propagated through the Linux VFS to the Lustre client file system VFS implementation, or llite. llite responds to this request by making a request to the MDS, asking for the OSTs on which the objects reside, as well as the stripe size and count. This metadata is returned to the client, who then directly contacts the OSTs on which the objects are stored. Once the objects are returned to the client, the client uses the stripe size and count to reconstruct the file. With the completion of the reconstruction of these objects, the file now exists on the local machine of the client and can be read by the user[27].

Using this logic, it can be seen that from the perspective of the end-user, the file system appears to be a local file system, where the files exist on the local disk. In reality, though, the llite layer has overridden the VFS `read(..)` function and included the logic necessary to retrieve the objects of the file from the Lustre cluster and reassemble the stripes contained within these objects in a single file.

[27] For the sake of brevity, as well as to reduce the chance of overwhelming the reader with details, many of the steps in the client, MDS, and OSS/OST interactions have been abstracted. For example, upon reading a file, a global read lock on the file must be obtained through the LDLM. This ensures that no other client can overwrite the file, much the same way that a local file system ensures that read and write locks are taken upon file access to ensure the consistency of the file. Likewise, each of the requests sent across the Lustre cluster use an RPC sent through the Portal RPC layer, where the RPC is interpreted and a response is computed accordingly. While these details are important in understanding the steps involved in a simple read request, they have been omitted in order to present the user with a high-level overview of how a read request is handled. For more information on these omitted intricacies, see [22].

2.7 MapReduce Architecture

The MapReduce architecture has seen an explosion is growth since the advent of large-scale, distributed software systems. This growth is due in large part to the ability of MapReduce engines, such as Apache Hadoop, to aggregate enormous amount of data from millions of nodes. While many primitive aggregation techniques suffice for small networks, the enormity of many modern distributed systems requires that a more powerful tool. In order to meet this challenge, many enterprises have turned to MapReduce as a scalable solution.

MapReduce was first introduced in 2004 in a Google research paper (see [36]), and since that time, MapReduce engines such as Hadoop have become an integral part of distributed infrastructure. While the implementation of MapReduce engines can be complex, the concepts behind MapReduce are simple. An example of a real-world situation is illustrated below in **Figure 22**.

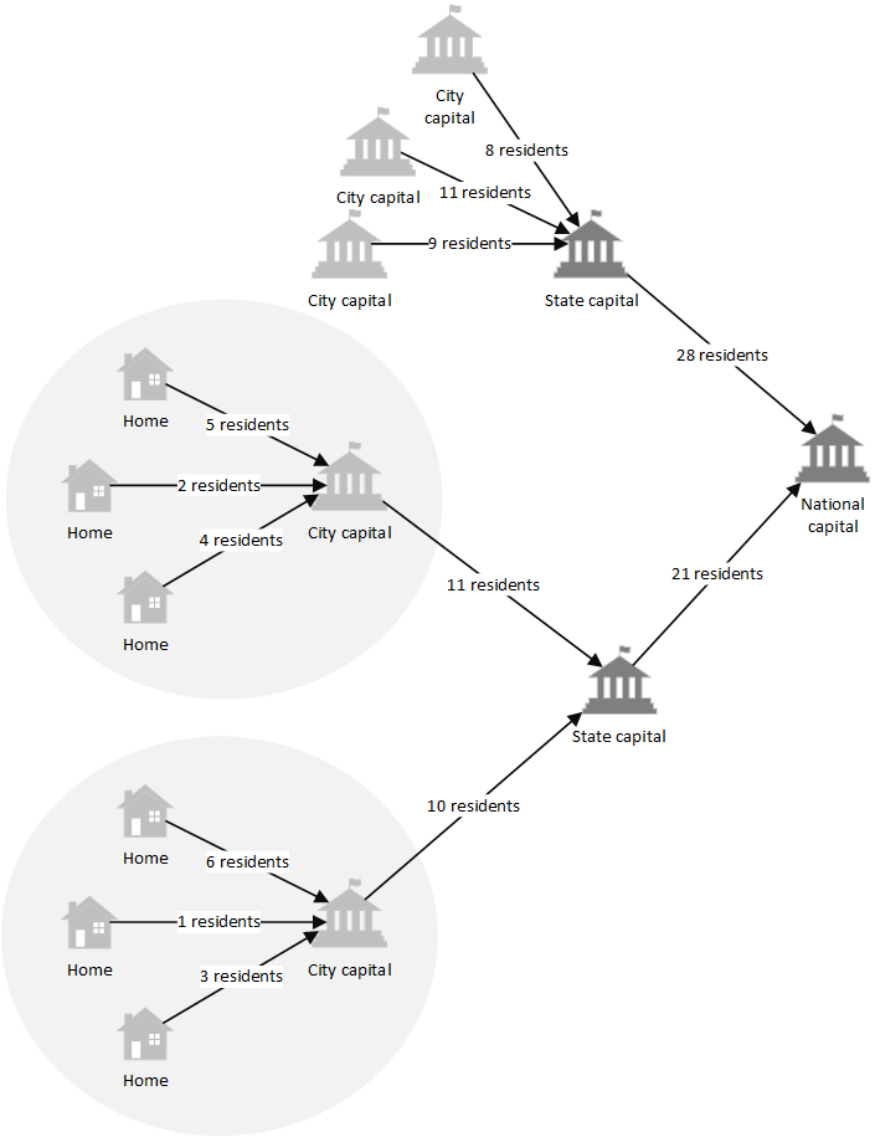

Figure 22. The process of census taking for a nation is broken down into the aggregation of the number of residents in each of the level of government (town, state, and national), as was down by the Roman Empire millennium ago.

In this example, a census is conducted, where every house must report the number of residents occupying the dwelling. The ultimate goal of this census is for the national capital to obtain a count of all residents living in all towns under its jurisdiction. A naïve solution to this problem would be to simply have every residence in the nation send its number of occupants directly to the national capital. In this way, the national capital would receive one message, containing the number of occupants, from every home in the nation. This solution is impractical for many reasons: (1) every home in the nation must know the location in the national capital to which its message must be sent, (2) an enormous volume of message traffic will be generated if every home must send a message directly to the capital, and (3) the national capital will be solely responsible for sifting through the enormous volume of aggregate data received from the homes of the nation (and will be responsible for obtaining the number of residents from this volume of data). While this solution is simple, it is clear to see that is it completely impractical.

Instead, each home can send its number of occupants to the government to which it has immediate contact. For example, homes in a town can send a message containing its number of occupants directly to its town or municipal government. This was the technique used by both the Roman government during world-wide census-taking and the US Constitution for the delegation of laws, and has many advantages over the naïve approach: (1) each home needs to know only the location of its town government, (2) the volume of messages sent to the town government is very small in comparison to the number of messages sent to the national government in the naïve approach, and (3) the town government must only sift through a relatively small amount of data to obtain the number of residents for the town. Once the governments of each town have tallied the number of residents for the town, each town passes on its tally to the state government. The state government then tallies the total number of residents for the state, which is then passed onto the national government. The national government must then tally the number of residents in each state to find the total number of residents in the entire nation.

In the case of the example presented in **Figure 22**, each town tallies the number of residents within its jurisdiction (e.g. 10 residents, in the case of the town at the bottom of the figure), and passes this tally onto the state government. The state government then tallies the number of residents in each town under its jurisdiction, and passes this tally onto the national government. The national government then tallies the number of residents from each of the state governments under its jurisdiction, which results in the overall number of residents in the nation. Apart from the advantages previously stated, this process is also parallelized, allowing each state to concurrently tally its residents prior to sending the results to the national capital.

In much the same way as the census example, MapReduce brakes up the process of aggregating data into two major steps (where the name of the steps join together to form the name of the algorithm):

1. **Map**: the data at the originating node is mapped into some key-value format. This mapping allows all nodes in the network to view data in the same format. In the census example, the homes in each town are the nodes where data originates and therefore, the town government is the first entity that interacts with this data. The town government would then collect the total number of residents in the town and create a key-value pair, where the key is the ID of the nation and the value is the number of residents in the town. This message is then passed to the state government, which performs the reduction.
2. **Reduce**: the data sent from the map is reduced until a final result is obtained. For example, each state government will receive messages containing a key-value pair from each town (where the key is the ID of the nation and the value is the number of residents in the town). These pairs can then be summed, resulting in a single key-value pair, keyed on the ID of the nation and containing the number of residents of the state as a value. This information is then sent to the national government, which acts as a reducer and sums the total residents from the messages received from each state.

There is also a third steps in the MapReduce algorithm: The shuffle step. In order to see the importance of this step, a more complex example is required. In the case of the census, the government for each town knows that it is responsible for all messages that it receives and that no messages should be forwarded to another town. This may not always be the case. For example, what if the law stated that any citizen of the nation could submit his or her census report to any town government, not just his or her own town government. In this case, each town would have to forward the reports not intended for it to the town government of which the submitter is a resident. This extra step of forwarding mapped data to the correct town government is the called the **shuffle** step.

A second example of a MapReduce environment is illustrated on the following page in **Figure 23**.

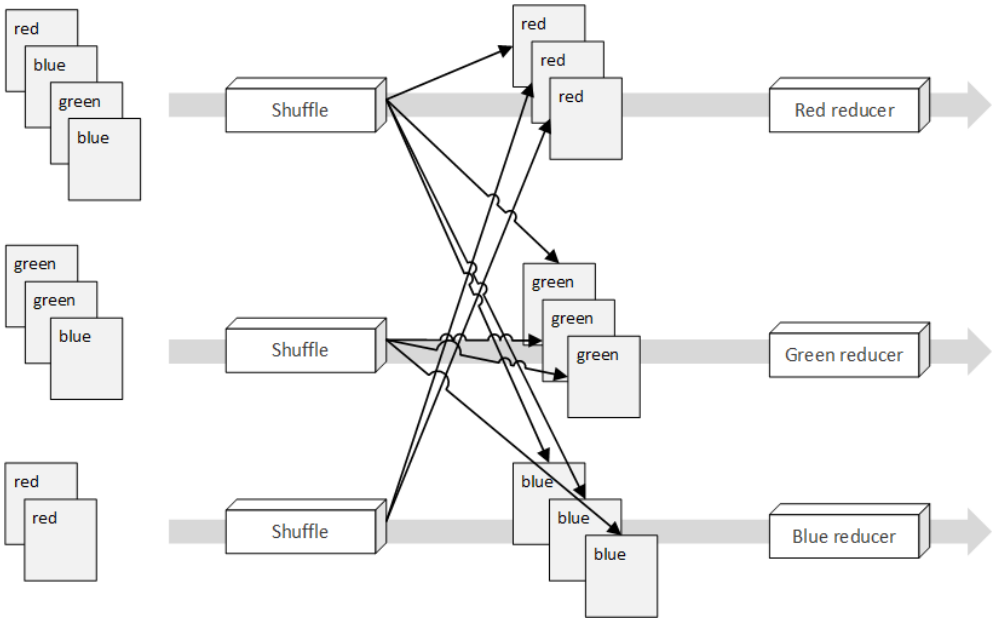

Figure 23. Since any color can appear before the shuffle process, input values are shuffled and directed to the appropriate reducer, optimizing the input data set supplied to the reducer [37].

In this example, a random assortment of colors is supplied at the start of the MapReduce process. In order to pipe the appropriate colors into the appropriate reducer, a shuffle is performed. As a result of this shuffle, all red units are provided to the red reducer, all green units are provided to the green reducer, and all blue units are provided to the blue reducer. During the reduction step, each reducer will produce a reduced key-value pair representing the colors it has received. For example, the red reducer may produce a result such as $(red, 3)$.

Note that there does not need to be *layers* or *levels* of reducers, as in the case of the census example. Instead, the reducers act as standalone components that simply reduce the lists of inputs that are received. This allows a MapReduce network to scale, dividing the data to be reduced into small parts and passing it along to thousands or millions of compute nodes that continue to reduce the data until a final solution results.

While the examples presented in this section provide a foundational understanding of the MapReduce process, there is a great deal of detail that has been omitted for the sake of brevity. For more information on MapReduce, see [36].

3. Installation Procedures

This section contains the installation instructions used to create the Virtual Machine (VM) image used to install the Lustre server and client software. While [21] describes the installation procedures for Lustre, this source does not describe the process through which the VMs for the Lustre file system are created. Therefore, the following walkthrough describes the installation and configuration process, from start to finish, for both creating the VMs as well as installing the Lustre software. For more detailed information on how to install the Lustre software and create a Lustre cluster, see **Part II: Installing and Configuring Lustre** of [21].

Section 1 describes the process used to install the Lustre server software on a VM and configure the VM to act as a server node in a Lustre cluster. Section 2 describes the procedures required to configure a series of Lustre server VMs, as created in section 1, to act as the server-side nodes (MGS, MDS, and OSSs) in a Lustre cluster. Section 3 describes the installation procedures for installing and configuring a VM to act as a client in a Lustre cluster. Section 4 describes the configuration procedures required to connect the client or clients created in section 3 to the Lustre cluster created in sections 1 and 2.

Note that each of these sections contain information on any issues encountered during the procedures. Likewise, at the time of writing, the server-side portion of the Lustre cluster (MGS, MDS, and OSSs) could not successfully connect to one another. For more information, see **Appendix D: Outstanding & Unresolved Issues: Failure to Connect OSS to MGS/MDS Node**.

3.1 Lustre Server Installation & Configuration

This section contains the detailed procedures for creating a base server VM used to create the MGS, MDS, and OSSs in the Lustre cluster. This section includes information on how to create the VM image for the MGS, MDS, and OSSs, as well as how to configure this image to be included in a Lustre cluster (the scope of this configuration stops at network configuration; subsequent sections cover the configuration required to create a MGS, MDS, or OSS from this base-image).

3.1.1 Creating Virtual Machine Image

The VM used to run the Lustre software was created and executing using VMWare Player 7[28] using the auto-installer for CentOS 6.6[29] 64-bit. In order to create the VM for CentOS 6.6, complete the following steps:

1. Open VMWare Player
2. Under the **Welcome to VMWare Player** heading, press the **Create a New Virtual Machine**
3. In the **New Virtual Machine Wizard** window, select the **Installer disc image (iso)** option
4. Select the **Browse** button and select the International Organization for Standardization (ISO) file representing the CentOS 6.6 installation image
5. Press the **Next** button
6. Enter the personalized information for the CentOS installation, such as the full name of the user, the login username, and the password for the login user
7. Press the **Next** button
8. Enter the **Virtual Machine name**, which will displayed in the list of VMs in VMWare Player
9. Select a location to store the VM files on the local machine
10. Press the **Next** button
11. Select a **Maximum disk size**
12. Select the **Split disk into multiple files** option

[28] VMWare Player 7.1.0 build-2496824
[29] CentOS-6.6-x86_64

13. Press the **Next** button
14. Ensure that the **Power on this virtual machine after creation** option is checked
15. Press the **Finish** button

VMWare Player will then execute the auto-installer for CentOS 6.6, installing a Graphical User Interface (GUI) for CentOS. While this GUI is not required, some of the tools needed, such as Wireshark, are arguably easier to use with a GUI, and therefore, a GUI for CentOS is installed. The auto-installer will take a few minutes to install the operating system; once this installation is completed, CentOS will automatically boot. Once the installation is complete (and the VM boots), shutdown the VM in order to change the network settings.

3.1.2 Installing Lustre Software

Throughout the following steps, it is assumed that CentOS VM is configured to use a Network Address Translation (NAT) network configuration. In order to change the network configuration for the VM,

1. Select the **Player** dropdown at the top-left of the VMWare window executing the CentOS VM
2. Select the **Manage** option
3. Select the **Virtual Machine Settings…** option
4. Select the **Network Adapter** option under the **Hardware** tab
5. Change the network configuration options under the **Network Configuration** heading on the right
6. Press the **OK** button once the desire configuration is set

Once complete, start the VM. When presented with the CentOS login screen, enter the login username and password specified in step (6) above. Once logged into the CentOS VM, the Lustre software and supporting tools can be installed. In order to install the software required to run the Lustre file system, a shared directory is used and mounted in the CentOS VM, thus allowing the needed RedHat Package Management (RPM) files to be transferred to and installed on the CentOS VM. To create the shared directory,

1. Select the **Player** dropdown at the top-left of the VMWare window executing the CentOS VM
2. Select the **Manage** option
3. Select the **Virtual Machine Settings…** option
4. Select the **Options** tab (next to the **Hardware** tab used when configuring the VM network)
5. Select the **Shared Folders** option on the left column
6. Check the **Always Enabled** option under the **Folder sharing** section in the right column
7. Press the **Add…** button at the bottom of the **Folders** section below the **Folder sharing** section
8. Press the **Next** button
9. Press the **Browse…** button under the **Host path** section
10. Select the directory to be shared between the host machine and the CentOS VM
11. Change the name of the shared directory, if desired, under the **Name** section (the name of this directory will be referenced as `<shared_dir>` for the remainder of the installation procedures)
12. Press the **Next** button
13. Ensure that the **Enable this share** checkbox is checked under the **Additional attributes** section
14. Press the **Finish** button

To verify that the shared directory has been properly mounted in the CentOS VM, open a shell in the VM and execute the following command,

```
$   ls -l /mnt/hgfs/<shared_dir>
```

where `<shared_dir>` is the name of the directory selected in step (11) when creating the shared directory. Once the shared directory has been established, the needed packages can be moved into this directory and installed. In order to install the Lustre file system on the CentOS VM, the following packages are required:

- `kernel-2.6.32-431.20.3.el6_lustre.x86_64.rpm`
- `lustre-2.6.0-2.6.32_431.20.3.el6_lustre.x86_64.x86_64.rpm`
- `lustre-iokit-2.6.0-2.6.32_431.20.3.el6_lustre.x86_64.x86_64.rpm`
- `lustre-modules-2.6.0-2.6.32_431.20.3.el6_lustre.x86_64.x86_64.rpm`
- `lustre-osd-ldiskfs-2.6.0-2.6.32_431.20.3.el6_lustre.x86_64.x86_64.rpm`
- `lustre-tests-2.6.0-2.6.32_431.20.3.el6_lustre.x86_64.x86_64.rpm`
- `libcom_err-1.42.12.wc1-7.el6.x86_64.rpm`
- `libss-1.42.12.wc1-7.el6.x86_64.rpm`
- `e2fsprogs-1.42.12.wc1-7.el6.x86_64.rpm`
- `e2fsprogs-libs-1.42.12.wc1-7.el6.x86_64.rpm`
- `compat-openmpi-1.4.3-1.2.el6.x86_64.rpm`
- `environment-modules-3.2.10-1.el6_5.x86_64.rpm`
- `libesmtp-1.0.4-15.el6.x86_64.rpm`
- `libgfortran-4.4.7-11.el6.x86_64.rpm`
- `libgssglue-0.1-11.el6.x86_64.rpm`
- `libibverbs-1.1.8-3.el6.x86_64.rpm`
- `librdmacm-1.0.18.1-1.el6.x86_64.rpm`
- `plpa-libs-1.3.2-2.1.el6.x86_64.rpm`
- `sg3_utils-1.28-6.el6.x86_64.rpm`
- `tcl-8.5.7-6.el6.x86_64.rpm`

While not all of the files listed above are required directly for a Lustre installation, this list includes all dependencies of the core Lustre packages, as well, allowing a user to install the complete Lustre file system server files without the need for an internet connection (which may not be present in the environment of the VM). Each of these files can be downloaded (to the shared directory on the host) directly from `https://github.com/albanoj2/grp/tree/master/lustre-packages/server`. Apart from the core Lustre server files, the following RPMs should also be installed:

- `wireshark-gnome-1.8.10-7.el6_5.x86_64.rpm`

This package provides the Wireshark application, and its associated GUI. This application will be used to analyze the network traffic originating from the Lustre file system[30]. The non-core packages can likewise be found at `https://github.com/albanoj2/grp/tree/master/lustre-packages/tools`.

To install these packages, login as the root user using the following command:

```
$   su
```

When prompted, enter the login password selected during the creation of the CentOS VM (the default root password is the login password selected during the creation of the CentOS VM). Once logged in as the root user, change directory to the shared directory containing the RPMs to be installed and execute the following command:

[30] Although this tool is not directly used within the research compiled in this document, it is useful when examining the traffic created by the Lustre file system when common use cases, such as creating a file, are executed.

```
#   yum --nogpgcheck install *
```

This command assumes that all of the packages to be installed (both the core Luster server packages, as well as the non-core packages) reside in the same directory. If this is not the same, simply change directory to any directory containing packages to be installed and execute the following command:

```
#   yum --nogpgcheck install <rpm_1> <rpm_2> ... <rpm_n>
```

where `<rpm_1>`, `<rpm_2>`, etc. are the names of the RPMs to install, including the `.rpm` file extension. The `--nogpgcheck` flag disables the GNU Privacy Guard (GPG) check, which allows unsigned packages to be installed (note that the authenticity of unsigned packages cannot be determined). While this is not a suggested practice when downloading packages from unknown or unsafe locations (such as from an unknown repository), the authenticity of these files is known *a priori*, since they were obtained from the official Lustre repository at [38] and [39].

Upon executing this command, the installation process will begin. When prompted to confirm the installation of the packages, enter `y`. The installation may take a few minutes. Once the installation is complete, a restart is required for the installation of the Lustre kernel to complete (the Lustre kernel will not be loaded until the CentOS VM is restarted). Therefore, reboot the system using the following command as the root user:

```
#   reboot
```

Although logging in as the root user is required to install packages, changing user to the root user using the `su` command is not always advised. Instead, the user created during the creation process for the CentOS VM can be given `sudo` rights. Once given `sudo` rights, this user will no longer be required to switch to the root user. Instead, the user can simply prepend the `sudo` command to each of the commands requiring root access. For example, `sudo echo “Hello, world!”` For more information on granting `sudo` rights to a user, see [40]. The remainder of these installation procedures will assume that user executing commands has `sudo` rights and therefore, the `su` command will not be used to switch to the root user.

Once the system has restarted, login to the CentOS VM. To ensure that the Lustre kernel has properly installed, open a terminal and execute the following command:

```
$   uname -r
```

This prints the release name of the kernel used on the system. After the previous reboot, the CentOS VM should have loaded the Lustre kernel previously installed. Therefore, the output from this command should be

```
2.6.32-431.20.3.el6_lustre.x86_64
```

Once the Lustre kernel installation has been confirmed, the CentOS VM can be configured.

3.1.3 Configuring the Server VM

The configuration of the CentOS VM can be divided into two main parts: (1) configuring Security-Enhanced Linux (SELinux) and (2) configuring the VM hostname and IP address. In order for Lustre to run on a Linux machine, SELinux must be disabled. In order to do this, open the `/etc/selinux/config` file and change the line

```
SELINUX=enforcing
```

to

```
SELINUX=disabled
```

For this change to take effect, the system must be rebooted. However, before rebooting, the hostname can also be changed (saving time by only rebooting the system once, after the hostname has been configured).[31]

In order to configure the hostname, the IP address of the system must be made static. To obtain the current IP address used by the CentOS VM, execute the `ifconfig` command. This command should return output similar to the following:

```
eth0      Link encap:Ethernet  HWaddr <some_mac_address>
          inet addr:192.168.aaa.bbb  Bcast:192.168.aaa.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe6e:ef4a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:73 errors:0 dropped:0 overruns:0 frame:0
          TX packets:51 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20412 (19.9 KiB)  TX bytes:4570 (4.4 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:480 (480.0 b)  TX bytes:480 (480.0 b)
```

Under the `eth0` section, the IPv4 address can be found under `inet addr:`. In the case of this example output, the IPv4 address is `192.168.aaa.bbb`. Using this existing address, a static address can be selected from range `192.168.aaa`. In the case of this example, the static IP address `192.168.aaa.140` is chosen (this IP address will be referenced by `<chosen_ip>` for the remainder of this installation procedure). To set this static IP address, open the `/etc/sysconfig/network-scripts/ifcfg-eth0` file in a text editor and perform the following actions:

1. Comment out the line containing `UUID=”<some_uuid4>”` by placing a # at the beginning of the line (i.e., change the line to `#UUID=”<some_uuid4>”`)
2. Change the line `BOOTPROTO="dhcp"` to `BOOTPROTO="static"`
3. Add the line `IPADDR=”<chosen_ip>”` (replacing `<chosen_ip>` with the IP address selected in the previous paragraph, not literally `<chosen_ip>`)
4. Add the line `NETMASK=”255.255.255.0”` (using the literal value `255.255.255.0`)

Save the file. The resulting configuration should resemble the following:

```
DEVICE="eth0"
BOOTPROTO="static"
IPADDR="<chosen_ip>"
NETMASK="255.255.255.0"
HWADDR="<some_mac_address>"
IPV6INIT="yes"
NM_CONTROLLED="yes"
```

[31] For more information on disabling SELinux see [41].

```
ONBOOT="yes"
TYPE="Ethernet"
#UUID="<some_uuid4>"
```

In order for this new static IP configuration to take effect, the `eth0` network adapter must be restarted. To do this, execute the following commands:

```
$    ifdown eth0
$    ifup eth0
```

Once the network adapter is brought up (using the `ifup` command), the IP address of the machine can be verified using the `ifconfig` command. Upon running this command, the new IP address should be set to `<chosen_ip>`. With the static IP address of the CentOS VM set, the hostname of the VM must be referenced to this address. This step must be performed, since the hostname of a machine running the Lustre server software cannot resolve to `localhost` (for more information, see the **Troubleshooting `llmount.sh`** section of [42]).

To change the hostname of the CentOS VM, open the `/etc/sysconfig/network` file in a text editor, and change the value of the `HOSTNAME=` key to the new hostname (for example, `lustre-vm`). The resulting file contents should resemble the following:

```
NETWORKING=yes
HOSTNAME=lustre-vm
```

Save and close the file. The next step is to map the hostname to the static IP previously set. To do this, open the `/etc/hosts` file in a text editor and add the following line to the end of the file (be sure to add the following on its own line within the file):

```
<chosen_ip>    lustre-vm
```

Note that `lustre-vm` should be replaced with the hostname selected in the `/etc/sysconfig/network` file. The file contents of the hosts file should resemble the following:

```
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.44.140  lustre-vm
```

Where `lustre-vm` is the hostname selected in the `/etc/sysconfig/network` file. For these changes to take effect, reboot the CentOS VM using the command `sudo reboot`.

3.1.4 Creating Copies of the Server VM

With the base server image created, copies of this image can be used to create the server nodes (MGS, MDS, and OSS) in the Lustre cluster. In order to copy the base image, the VM must be shutdown. Therefore, if the VM created in the previous section is still running, shutdown the VM by completing the following steps:

1. Select the **Player** dropdown at the top-left of the VMWare window executing the CentOS VM
1. Select the **Power** option
2. Click the **Shut Down Guest** option

Be sure not to simply suspend the guest, as suspending the guest saves the state of the VM and may cause issues when copying the VM. To create a copy of the server image, complete the following steps:

1. Open a file explorer in the directory in containing the virtual machines used by VMWare Player[32]
2. Duplicate (copy and paste) the directory containing the VM files representing the VM created in the previous section (the directory will have a name similar to that of the VM name set in the previous section, with spaces replaced by underscores)
3. Rename the duplicated directory to the desired name of the new VM directory (for example, the desired name of the VM, replacing spaces with underscores)
4. Open VMWare Player
5. Select the **Player** dropdown at the top-left of the VMWare window
6. Select the **File** option
7. Select the **Open…** option
8. Navigate to the newly copied directory (the duplicate directory created in step 2 and renamed in step 3)
9. Open this duplicated directory
10. Select the `.vmx` file in this duplicated directory
11. Press the **Open** button
12. Right click the newly added VM in the list of VMs on the left (the name of the VM will match the name of the original VM from which the copy was made)
13. Click the **Settings…** option
14. Select the **Options** tab
15. Change the name of the VM under the **Virtual machine name** section on the right column to the desired name of the new VM
16. Press the **OK** button at the bottom of the settings window
17. Play the renamed VM
18. Press **I Copied It** option from the window warning *This virtual machine might have been moved or copied* after playing the duplicated VM

With these steps completed, the duplicated VM is a direct copy of the VM from which it was duplicated. This process should be repeated for each of the server nodes desired. In the case of this walkthrough, the MGS and MDS are combined into a single VM, and only one OSS will be created. Therefore, one copy of the original CentOS VM is sufficient (providing two VMs: the original VM and the copied VM). It is highly suggested that the base server image (the CentOS VM at this point in the walkthrough) is copied or archived. Archiving this VM will allow new server VMs to be created at will by copying this archived VM using the steps above.

3.2 Server-side Communication Setup

In order to configure the server nodes, the one of the server VMs must be configured to act as a MGS and MDS. While the MGS and MDS can be configured as separate nodes in a larger Lustre cluster, for the purposes of this research, a combined MGS/MDS will suffice to support the cluster. Once the MGS/MDS node has been configured, the remaining server VM must be configured as an OSS.

In the case of the MGS/MDS VM, the configuration of the node entails creating a virtual block device (representing the disk that will act as the MGT/MDT) and mounting this block device. Likewise, in the case of the OSS, a block device must be created and mounted for the OST.

[32] In Windows, the directory containing the virtual machine images for VMWare Player is `C:\Users\<username>\Documents\Virtual Machines`. For more information on location the directory containing the files that make up a VM, see [43].

3.2.1 Creating & Mounting MGT/MDT Block Device

To create the block device used as the MGT/MDT disk, the MGS/MDS VM must be shutdown. Either of the server VMs created in the previous steps may be used as the MGS/MDS; whichever is selected, power down the VM and complete the following steps:

1. Right click on the VM selected as the MGS/MDS VM
2. Click the **Settings…** option
3. Click the **Add…** option at the button of the **Hardware** section on the left (if prompted to approve administrative access, agree)
4. Click **Hard Disk** in the left menu
5. Click the **Next** button
6. Select the **SCSI** option
7. Click the **Next** button
8. Select **Create a new virtual disk**
9. Click the **Next** button
10. Select an appropriate disk size (for the sake of this research, $5\ GB$ will suffice)[33]
11. Select the **Split virtual disk into multiple files** options at the bottom of the window
12. Click the **Next** button
13. Change the name of the disk image in the **File** field under the **Disk file** section
14. Click the **Finish** button

Once the virtual hard disk is created, it will appear under the MGS/MDS VM under `/dev/sdb` (where `/dev/sda` is the primary block device on which the CentOS operating system is installed). With the block device created, the device must be formatted and mounted in order for the MGS/MDS to serve the Lustre cluster. To format the block device, play the MGS/MDS VM, and open a terminal. Once the terminal has opened, format the block device with the following command:

```
$   sudo mkfs.lustre --fsname=lustre --mgs --mdt --index=0 /dev/sdb
```

This command assumes that the newly created block device is located at `/dev/sdb`; if this is not the case, simply replace `/dev/sdb` with the location of the block device. Note that the above command assumes that the `mkfs.lustre` has not been previously run on the device. If the disk is being reformatted (the `mkfs.lustre` is being run on a disk that has already been formatted using the `mkfs.lustre` command), include the `--reformat` flag before the location of the block device. For example,

```
$   sudo mkfs.lustre --fsname=lustre --mgs --mdt --index=0 --reformat /dev/sdb
```

Once the format process has completed, the following output (or similar output) should be seen:

```
   Permanent disk data:
Target:     lustre:MDT0000
Index:      0
```

[33] For more information on selecting an appropriate size for the MGT/MDT block device, see section **Determining Hardware Configuration Requirements and Formatting Options** of [1].

```
Lustre FS:  lustre
Mount type: ldiskfs
Flags:      0x65
              (MDT MGS first_time update )
Persistent mount opts: user_xattr,errors=remount-ro
Parameters:

device size = 2048MB
formatting backing filesystem ldiskfs on /dev/sdb
       target name  lustre:MDT0000
       4k blocks     524288
       options        -J size=81 -I 512 -i 2048 -q -O
dirdata,uninit_bg,^extents,dir_nlink,quota,huge_file,flex_bg -E lazy_journal_init -F
mkfs_cmd = mke2fs -j -b 4096 -L lustre:MDT0000  -J size=81 -I 512 -i 2048 -q -O
dirdata,uninit_bg,^extents,dir_nlink,quota,huge_file,flex_bg -E lazy_journal_init -F
/dev/sdb 524288
Writing CONFIGS/mountdata
```

With the block device formatted, the disk must be mounted in order to start the MGS/MDS service in the Lustre cluster. To mount the disk, first create a mount point and then mount the Lustre file system, using the following set of commands:

```
$    sudo mkdir -p /mnt/mgs-mds

$    sudo mount -t lustre /dev/sdb /mnt/mgs-mds
```

To ensure that the MGS/MDS has been successfully added to the Lustre cluster, execute

```
$    sudo cat /proc/fs/lustre/mgs/MGS/live/*
```

This command should produce the following output [21][34]:

```
fsname: lustre
flags: 0x20     gen: 7
lustre-MDT0000

Secure RPC Config Rules:

imperative_recovery_state:
    state: startup
    nonir_clients: 0
    nidtbl_version: 3
    notify_duration_total: 0.000000
    notify_duation_max: 0.000000
    notify_count: 1
```

[34] For more information on maintaining the Lustre file system, see section Lustre Maintenance of [21].

```
fsname: params
flags: 0x21     gen: 1

Secure RPC Config Rules:

imperative_recovery_state:
    state: startup
    nonir_clients: 0
    nidtbl_version: 2
    notify_duration_total: 0.000000
    notify_duation_max: 0.000000
    notify_count: 0
```

If the MGT/MDT block device must be unmounted, execute the following command:

```
$    sudo umount /dev/sdb
```

Note that the command is `umount` (without the *n*), not `unmount`. Secondly, note that if the MGT/MDT block device must be reformatted, the block device must be unmounted prior to running the reformat command. With the block device mounted, the MGS/MDS is now running in the Lustre cluster. In order to complete the server-side portion of the Lustre cluster, at least one OSS, with an accompanying OST, must be connected to the cluster.

3.2.2 Creating & Mounting OST Block Device

Using the remaining server VM, create a virtual hard disk using the process presented in above in **Creating & Mounting MGT/MDT Block Device**. This new virtual hard disk will act as the block device for the OST associated with the OSS. Before formatting this hard disk, the static IP configuration of this OSS VM must be changed: Because the OSS VM is a copy of the MGS/MDS VM, it will have the same static IP configuration as the MGS/MDS VM. Leaving this duplicate static IP configuration will result in an IP clash on the NAT network that both VMs are connected to. In order to resolve this conflict, play the OSS VM and open the `/etc/sysconfig/network-scripts/ifcfg-eth0` file and change the `IPADDR` value to a different IP address (called `<oss_ip>` for the remainder of this document). Save and close the file.

Once the static IP has been changed, the IP-host mapping must also be changed. To change this mapping, open the `/etc/sysconfig/network` file and change the IP address to `<oss_ip>` for the `lustre-vm` host name. Save and close this file. In order for these changes to take effect, restart the OSS VM. Note that through the remainder of this document, the static IP address of the MGS/MDS VM (the unchanged IP address originally established for the server VM, called `<chosen_ip>` in section **Configuring the Server VM** above) will be called `<mgs_ip>` and the changed static IP for the OSS will be called `<oss_ip>`.

To format the block device as an OST, execute the following command:

```
$    sudo mkfs.lustre --fsname=lustre --mgsnode=<mgs_ip>@tcp0 --ost --index=0 /dev/sdb
```

Note that `<mgs_ip>` should be replaced with the static IP address of the MGS/MDS VM. Note that if the OST must be reformatted, the `--reformat` flag must be included in the above command. Once this command completes, the OST must be mounted. To mount the newly formatted OST, a mount must be created and the OST block device must be mounted to this mount point. To accomplish this, execute the following commands:

```
$    sudo mkdir -p /mnt/ost0

$    sudo mount -t lustre /dev/sdb /mnt/ost0
```

Once the OST is mounted, the creation and configuration of the server-side portion of the Lustre cluster is complete. The resulting Lustre cluster contains a single MGS/MDS node, with an accompanying MGT/MDT block device, and a single OSS, with an accompanying OST block device. Note that this is essentially a minimum Lustre cluster, and is not common of enterprise Lustre file systems. To expand the cluster, simply add more OSTs to the OSS, and likewise, increase the number of OSSs (and accompanying OSTs) to the file system using the methods described above.

For example, to add another OST to the OSS VM, simply create another virtual hard disk (using the procedure presented at the begin of section **Creating & Mounting MGT/MDT Block Device**), format the hard disk using the same command as the original hard disk, but incrementing the index to `1`. For example,

```
$    sudo mkfs.lustre --fsname=lustre --mgsnode=<mgs_ip>@tcp0 --ost --index=1 /dev/sdc
```

Note that the device location is no longer `/dev/sdb`, but rather, `/dev/sdc`. As more OSTs are added to the OSS, both the index and the location will increment (`/dev/sdd`, `/dev/sde`, etc.). Also note that the supplied index is not local to the OSS. Therefore, if a second OSS were added, and three OSTs are mounted on the first OSS (OST0, OST1, and OST2), an OST added to the second OSS would have an index of 4, not 0. Since there are three OSTs existing in the file system prior to the mounting of the OSTs on the second OSS, the index must be incremented to 4.

The same pattern does not apply for the block device location on each of the OSS: The location of the OST device blocks on each of the OSSs is unique to the OSS. Therefore, the virtual hard disk representing the OST on the second OSS would be located at `/dev/sdb`, even though there are three existing OSTs on the first OSS. Since the locations of the virtual hard disks are local to the machine mounting the hard disk, the locations of the three existing virtual hard disks representing OST0, OST1, and OST2 are unknown to the second OSS. This scheme is illustrated below in **Figure 24**.

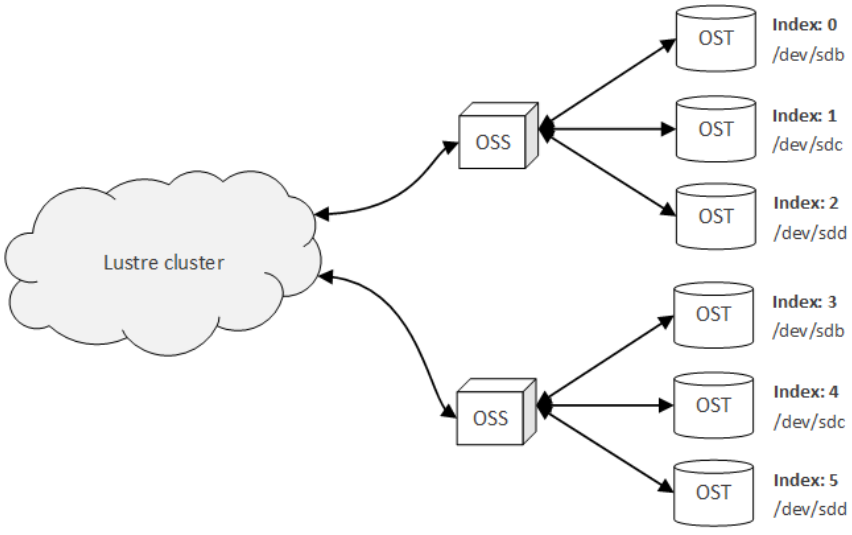

Figure 24. The OST indices are global in scope and are therefore sequential, even when associated with different OSSs, while the device locations of the OSTs are local to each OSS, and therefore are sequential only within the scope of each OSS.

In general, the capacity of a Lustre file system is equal to the aggregate storage provided by each of the OSTs in the file system. Therefore, if more storage space is needed, it is advised that more OSTs are created, rather than increasing the size of the existing OSTs.

At this point in the installation process, an unresolved issue was discovered. Due to the nature of the issue, the OSTs associated with the OSS were unable to mount, and therefore, the server-side portion of the Lustre file system could not be constructed. The nature of this issue, as well as the approaches taken thus far to resolve the issue is documented in **Appendix D: Outstanding & Unresolved Issues: Failure to Connect OSS to MGS/MDS Node** of this document.

3.3 Lustre Client Installation & Configuration

This section contains the steps needed to create and configure a Lustre client VM that is used to access the file system managed by the Lustre server nodes created in the previously presented steps. Many of the steps in these procedures are identical to those of the server VM creation, and completion of the steps in the two previous subsections will aid greatly in the understanding of the steps presented in this and the following subsection.

3.3.1 Creating Virtual Machine Image

Just as with the server VM, the client will be created using VMWare Player 7, leveraging the auto-installer for CentOS 6.6. In order to create the client VM using CentOS 6.6 as the base operating system, complete the following:

1. Open VMWare Player
2. Under the **Welcome to VMWare Player** heading, press the **Create a New Virtual Machine**
3. In the **New Virtual Machine Wizard** window, select the **Installer disc image (iso)** option
4. Select the **Browse** button and select the ISO file representing the CentOS 6.6 installation image
5. Press the **Next** button
6. Enter the personalized information for the CentOS installation, such as the full name of the user, the login username, and the password for the login user
7. Press the **Next** button
8. Enter the **Virtual Machine name**, which will displayed in the list of VMs in VMWare Player
9. Select a location to store the VM files on the local machine
10. Press the **Next** button
11. Select a **Maximum disk size**
12. Select the **Split disk into multiple files** option
13. Press the **Next** button
14. Ensure that the **Power on this virtual machine after creation** option is checked
15. Press the **Finish** button

Similar to the creation of the server VM, the client VM will then start and the auto-installer will begin; this installation process may take a few minutes to complete. Once completed, the network settings of the VM must be configured and the Lustre client software can be installed.

3.3.2 Installing Lustre Software

In order to properly communicate with the other nodes in the Lustre cluster, the client VM must be configured to connect to the same NAT network as the previously created server VMs. To do this, complete the following steps:

1. Select the **Player** dropdown at the top-left of the VMWare window executing the CentOS VM
2. Select the **Manage** option
3. Select the **Virtual Machine Settings…** option
4. Select the **Network Adapter** option under the **Hardware** tab
5. Change the network configuration options under the **Network Configuration** heading on the right
6. Press the **OK** button once the desire configuration is set

If the VM was running when these steps were completed, it must be restarted. In order to install the Lustre client RPMs, a shared directory is used and mounted in the CentOS client VM, allowing the VM to directly access the required packages on its local file system (instead of downloading the packages from the internet, which will not be available to the VM while using the previously configured NAT network settings). To create the shared directory,

1. Select the **Player** dropdown at the top-left of the VMWare window executing the CentOS VM
2. Select the **Manage** option
3. Select the **Virtual Machine Settings…** option
4. Select the **Options** tab (next to the **Hardware** tab used when configuring the VM network)
5. Select the **Shared Folders** option on the left column
6. Check the **Always Enabled** option under the **Folder sharing** section in the right column
7. Press the **Add…** button at the bottom of the **Folders** section below the **Folder sharing** section
8. Press the **Next** button
9. Press the **Browse…** button under the **Host path** section
10. Select the directory to be shared between the host machine and the CentOS VM
11. Change the name of the shared directory, if desired, under the **Name** section (the name of this directory will be referenced as `<shared_dir>` for the remainder of the installation procedures)
12. Press the **Next** button
13. Ensure that the **Enable this share** checkbox is checked under the **Additional attributes** section
14. Press the **Finish** button

To ensure that the shared directory has been properly mounted to the CentOS VM, execute the following command:

```
$    ls -l /mnt/hgfs/<shared_dir>
```

Installing the client software on the client VM requires root access to the VM. In order to grant `sudo` access to a user, see [40]. For the remainder of this walkthrough, it will be assumed that the user executing commands on the client VM has `sudo` access.

Creating a client in a Lustre cluster requires substantially fewer packages than creating a server Lustre node. In particular, the Lustre-specific kernel modules needed by the server nodes are not required by client nodes, which greatly reduces the number of Lustre packages and dependencies that must be installed. For client nodes, only the following packages must be installed:

- `lustre-client-2.7.0-2.6.32_504.8.1.el6.x86_64.x86_64.rpm`
- `lustre-client-modules-2.7.0-2.6.32_504.8.1.el6.x86_64.x86_64.rpm`
- `lustre-client-tests-2.7.0-2.6.32_504.8.1.el6.x86_64.x86_64.rpm`
- `lustre-iokit-2.7.0-2.6.32_504.8.1.el6.x86_64.x86_64.rpm`

Similar to the server packages, these RPMs can be downloaded from `https://github.com/albanoj2/grp/tree/master/lustre-packages/client`. Once these packages have been downloaded to the shared directory, change directory to the shared directory:

```
$    cd /mnt/hgfs/<shared_dir>
```

Once within the shared directory, install the downloaded RPMs:

```
$    sudo yum --nogpgcheck install *
```

When prompted to confirm the install, enter `y` to confirm. Once the installation has completed, the client VM must be restarted. To do so, execute the following command:

```
$    sudo reboot
```

3.3.3 Connecting Client to Lustre File System

Similar to the comparative ease of installing the Lustre client software when compared to installing the Lustre server software, connecting a client to a Lustre file system is likewise easier than connecting a server node. Once the client software is installed, all that is required to connect a client to an existing Lustre file system is to mount the directory established by the MGS of the cluster to the local file system of the client. To do this, a mount point must be created and then the Lustre file system must be mounted using the following commands:

```
$    sudo mkdir /mnt/lustre

$    sudo mount -t lustre <mgs_ip>@tcp0:/lustre /mnt/lustre
```

where `<mgs_ip>` is the IP of the MGS in the existing Lustre cluster (as created in the previous subsections of this section). Once the file system is mounted, the file system can be accessed by entering the `/mnt/lustre/` directory on the local file system of the client. It is suggested by [21] that the following commands be executed while in the newly mounted Lustre directory:

```
$    lfs df

$    dd

$    ls
```

In order to create more clients in a Lustre file system, the steps presented in this section for creating and configuring a client can be repeated for as many clients as desired. At this point in the configuration of the Lustre client, the client was unable to connect to the MGS. For more information about this outstanding issue, see **Appendix D: Outstanding & Unresolved Issues: Failure to Connect OSS to MGS/MDS Node**.

4. Problem Statement

File system forensics is a long and historied field of computer systems and has become a well-researched field over its lifetime. In particular, file recovery has been of primary interest for both personal use, as well as enterprise use. In the former case, forensic analysis tools are used to recover a file that may have accidentally been deleted from the file system of a local machine or to recover a file system that may have been corrupted. In these scenarios, the end-user is commonly an individual user that is interested in recovering a file from the file system as a result of unintended deletion, rather for the purposes of analyzing the file system for illegal or nefarious behavior. In the latter case, enterprises, including companies, law enforcement, and intelligence agencies, are interested in recovering files for the sake of using them as evidence in legal proceedings, recovering from large-scale accidental deletions, or obtaining information that was previously thought to be deleted for the purpose of intelligence gathering.

Until recently, both cases focused on a local file system, were all files where contained on a local disk, or a local volume that presented itself to the operating system as a single, unified disk (such as in the case of RAID arrays or other collections of redundant disks). With the wide-spread use of distributed file systems, the field of digital forensics and file recovery have evolved to include the analysis and recovery of files on file systems spread across geographically and logically disperse networks. This presents multiple challenges, including

1. **Legal**: Since many of the nodes in a distributed file system are located in various geographic locations throughout the world, questions of legal jurisdiction arise. For example, do the files of a file system owned by an American company, but existing on a file system node located in Germany, fall under the jurisdiction of the American legal system or German legal system? Furthermore, must the authorities involved in the country where the server node exists be consulted when performing file system forensics or must only the country of which the entity owning the files be consulted? Questions such as these are still in the process of being answered and codified into law. This paper does not focus on answering questions such as these, but it is important to bear in mind these important considerations.
2. **Finding the location of the disks containing the data**: In the case of the local file system, this question is already answered: The files exist on the disk mounted directly to the machine on which the forensics are being performed. In the case of distributed file systems, such as Lustre, this information is not known *a priori*. Instead, this information must be discovered somehow, based on the configuration of the file system and the constraints of the operating environment of the file system. For example, if files were to be recovered from a Lustre file system, the index of the OSTs containing the objects of a file must be known before the objects of a file can be found and consolidated into the a single, coherent file.
3. **Stitching together the components of a file**: In distributed file systems, files are divided into components and dispersed across nodes in a network. This functionality allows the file system to be *distributed*: Since files do not exist as a single, indivisible entity, they can be distributed across multiple nodes in the network. In order to reconstruct a file, the location of these file components must be maintained (as is the case in the MDS/MDT and OSS/OST relationship in a Lustre file system). Once the locations of these file components are known, they must be retrieved and stitched back together to form the singular, full file.

In general, the issues associated with file recovery on a distributed file system are issues associated with a local file system, replicated twice. This is particularly true for the Lustre file system, which uses a local file system on the OSTs to store the objects associated with the files in the Lustre cluster. For example, in order to find the individual blocks that make up the object of a file on an OST[35], the Lustre file system must first look-up which OSTs the objects that make up the file are striped across, and then read the objects from these OSTs. This reading process then requires the file system of the OST to find the data blocks of the file and construct the object. These constructed objects are then sent to the client that wishes to interact with the file. In the complete process, two look-ups must be made: (1) a look-up at the Lustre file system level, where the OSTs over which the file is striped are discovered, and

[35] Since files are represented as objects on the OSTs of a Luster cluster, and the objects are in turn represented as files on the local file system of each OST, the components of a file in a Lustre file system are ultimately represented as the data blocks on the local disk of the OSTs on which the objects of the file reside.

(2) a look-up on each of these OSTs to find the data blocks on the disks associated with the OSTs. A sequence diagram representing this logic is illustrated below in **Figure 25**.

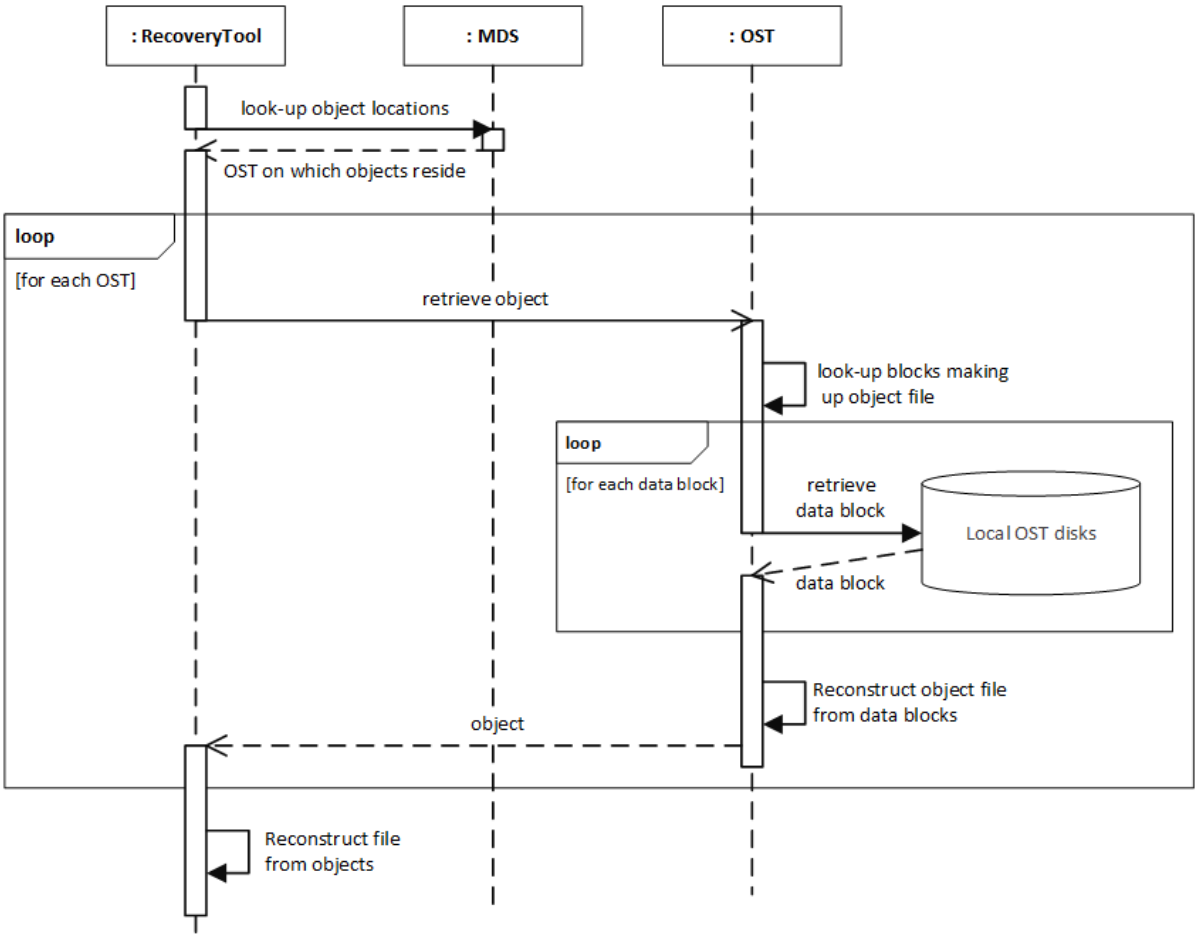

Figure 25. For the Lustre file system, the challenges involved in file recovery are replicated twice: Once for the Lustre level (node-to-node) and again at the OST level (OST-to-disk).

Note that the issue of stitching together the components of a file are also replicated twice: (1) the object file must be reconstructed from the data blocks at the OST level, and (2) the requested file must be reconstructed from its corresponding objects at the Lustre level. In the case of the OST level look-up and stitching, local file system recovery tools can be used to recover the object file for each OST. For example, conceptually, given the information for the object file to be retrieved, the file recovery tools can be used to reconstruct the object file if it has been deleted from the OST. In essence, if the object file on an OST has been deleted once a file has been deleted from the Lustre file system, it can recovered using existing tools. This recovery process results in three distinct possibilities:

1. The deleted file has not be overwritten and can be recovered in its entirety
2. The file has been partially overridden and only a partial representation of the file can be recovered
3. The deleted file has been overwritten and cannot be recovered in its entirety

In the first case, the object file can be sent to the entity requesting the recovered object file and be used to reconstruct the deleted file. In the second case, the file cannot be recovered in its entirety, and as a result, the deleted file on the Lustre file system cannot in turn be recovered in its entirety. In this case, the partial object file can be sent to the entity requesting the recovered object file and can be used to create a partially recovered file.

Since the problem of recovering deleted files from a local file system is a well-researched and highly involved field of study in-and-of-itself, a detailed discussion of this process is not included in this paper. Likewise, the challenge of recovering a file from the local file system of an OST is not contained within the scope of this paper. Instead, when the object file from an OST is required to be recovered, this paper will defer to the tools and research available to recover the object file. In essence, this portion of the recovery process can be abstracted into an tool that takes in the identifier (index) of the OST on which the object file resides and the object file identifier, and returns one of the three mutually exclusive results: (1) the complete object file, recovered in its entirety, (2) a partial object file, containing 1 to $n-1$ data blocks, exclusively, where n is the total number of data blocks that made up the object file before its deletion[36], or (3) an error stating that no part of the object file could be recovered (no data blocks for the object file could be recovered). This depiction of the Abstract Object File Recovery Tool (AOFRT) is illustrated below in **Figure 26**.

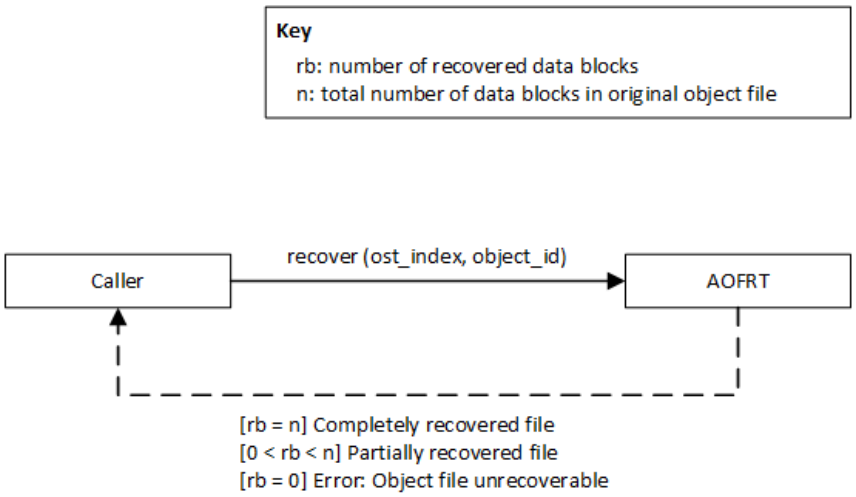

Figure 26. The mechanism for recovering object files from an OST can be abstracted into an AOFRT, which takes in an OST index and object file ID and produces one of three outputs.

With the inclusion of the AOFRT, the sequence diagram illustrated in **Figure 25** can be simplified. Instead of directly contacting the OST on which the object file resides (which is no longer possible, since the file has been deleted and thus the object file that make up the file are in turn deleted), the recovery entity contacts the AOFRT, providing it with the index of the OST to recover the file from, as well as the ID of the object to recover. Deferring the recovery of object files to the AOFRT, two recovery challenges remain: (1) discovering where the objects associated with a deleted file reside and (2) reconstructing the recovered objects into the original, deleted file. The simplified sequence diagram, including the AOFRT and unanswered question of the components responsible for discovering the location of the objects and reconstruction the recovered objects, is illustrated in **Figure 27**. The following section provides a solution for the missing components.

[36] For example, if the object file, prior to deletion, is made up of 52 data blocks, then $n = 52$. Once the object file is deleted, and the recovery tool is used to retrieve the file, one of three results can occur: (1) all 52 data blocks are retrieved and reconstructed in order, resulting in the fully recovered object file; (2) anywhere from $[1,51]$ data blocks are retrieved and reconstructed in order, resulting in a partially recovered file; or (3) 0 data blocks are recovered and an error is raised to denote that the object file could not be recovered.

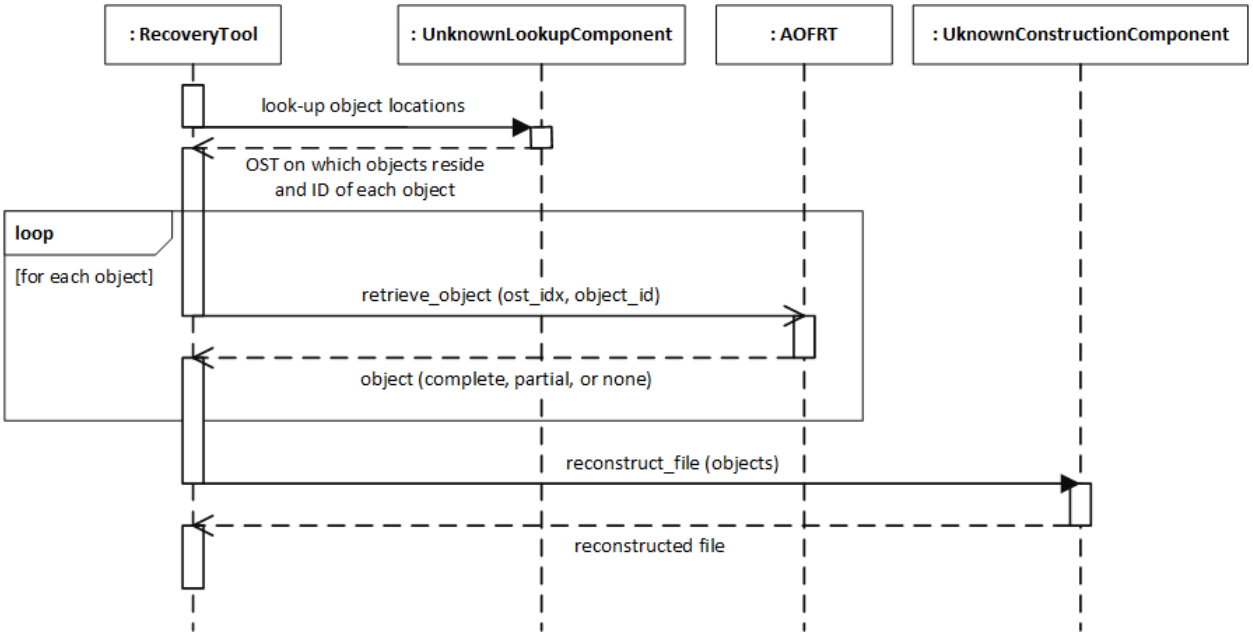

Figure 27. With the inclusion of the AOFRT, only two questions remain: (1) how are the OSTs containing the objects discovered and (2) how are the recovered objects reconstructed?

5. Solution

Given the premise presented in the previous section, any solution to the problem of file recovery on a Lustre file system requires that two objectives be completed: (1) retrieval of object information, including the striping data and mapping of objects to their respective OSTs, and (2) the reconstruction of the objects obtained from the AOFRT. Based on the research conducted by the author, there exists a simple solution to both these problems, involving a combination of the recovery methods of the AOFRT and the existing Lustre source code.

Prior to presenting this solution, an array of prerequisite terminology must be defined and a series of concepts must be understood. Following the presentation of this prerequisite knowledge, the devised solution, known as the Three-Step Recovery Solution, is illustrated in detail. For more information on possible solutions that require further research, see **Appendix B: Incomplete Solutions**.

5.1 Background Knowledge & Terminology

In order to understand the devised solution to this problem, the terminology used in the solution must be precisely defined. As described in the **Client Interface to a Lustre File System** section of this document, each of the targets (MDT, MGT, and OST) in a Lustre file system is formatted using a local file system, referred to as the backing file system of the target. For example, when an OST is created, it can either be formatted as ldiskfs file system or as ZFS. This backing file system allows the target to store the files and data of the Lustre file system, and makes the recovery of the object files on the OSTs by the AOFRT possible.

Therefore, when the recovery process presented in the following solution is executed, it is assumed that the targets of interest (MDTs, MGTs, and OSTs) are disconnected from the Lustre cluster and mounted to the machine conducting the forensic analysis. This process is referred to in this paper as *offline* examination of the targets. In contrast, an *online* solution would examine the targets while they are still connected in the Lustre cluster. Although it is not impossible to perform the process presented in this solution online, the logic behind the solution is clearer and more succinct when presented in an offline manner.

In summary, the following points are integral in understanding the solution presented in this section:

- Each target (MDT, MGT, and OST) have a local file system, referred to as the backing file system of the target, that is used to store the Lustre data, such as metadata inodes (in the case of MDTs) and object files (in the case of OSTs)
- The solution is executed in an *offline* manner, where the targets of interest are unmounted (disconnected) from the Lustre cluster and subsequently mounted to the machine performing the analysis, allowing the analysis machine to access the targets as if they were local, mounted file systems

5.2 Three-Step Recovery Solution

In order to construct the Three-Step Recovery Solution[37], the inputs to the second objective (how to reconstruct the object files into a single, recovered file) are used as the goal for completing the first objective. While the creation of a mechanism to reconstruct object files into a single, coherent file can be constructed in a custom manner, this functionality already exists within the Lustre file system code base. In essence, the reconstruction of objects into a single file is what is performed by a Lustre client when the client interacts with a file. Recall from the discussion in the **An Overview of the Lustre File System** section of this paper that a client obtains the metadata for a file from the MDS and then contacts the OSTs to obtain the objects that make up the file. Once these objects have been retrieved, the objects are then stitched together to create the single file that the end-user interfaces with.

This is precisely the functionality needed by the solution in order to reconstruct a file from the objects that make up the file (along with the striping information about the file). Therefore, instead of creating a new component to

[37] This name for the solution is discussed further in the latter parts of this section and require knowledge of the solution to make sense of the selected name. Therefore, a description of the name selection for this solution is deferred until later in this section.

reconstruct a file, the existing Lustre client code base responsible for file reconstruction can be abstracted in much the same way as the existing file recovery tools can be abstracted into the AOFRT. The resulting Abstract File Reconstruction Tool (AFRT) has three inputs and one output; the inputs being: (1) the order list of objects that make up the file to be recovered, (2) the stripe size (the length of each stripe), and (3) the file size[38]; the output being: (1) the reconstructed file. The input-output view of the AFRT is illustrated below in **Figure 28**.

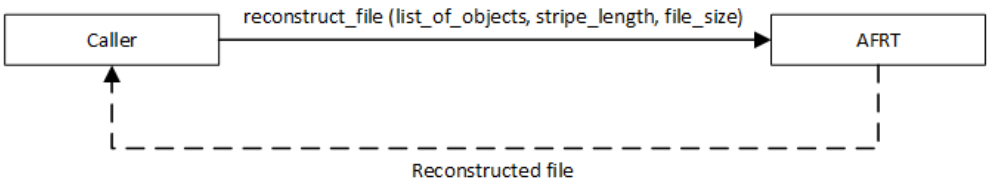

Figure 28. The AFRT can be used to reconstruct a file from its component objects by leveraging the file reconstruction logic already available in the Lustre client node.

Using the AFRT to abstract the logic required to reconstruct the file from its respective objects, the sequence diagram presented in **Figure 27** is simplified to the sequence diagram depicted in **Figure 29**.

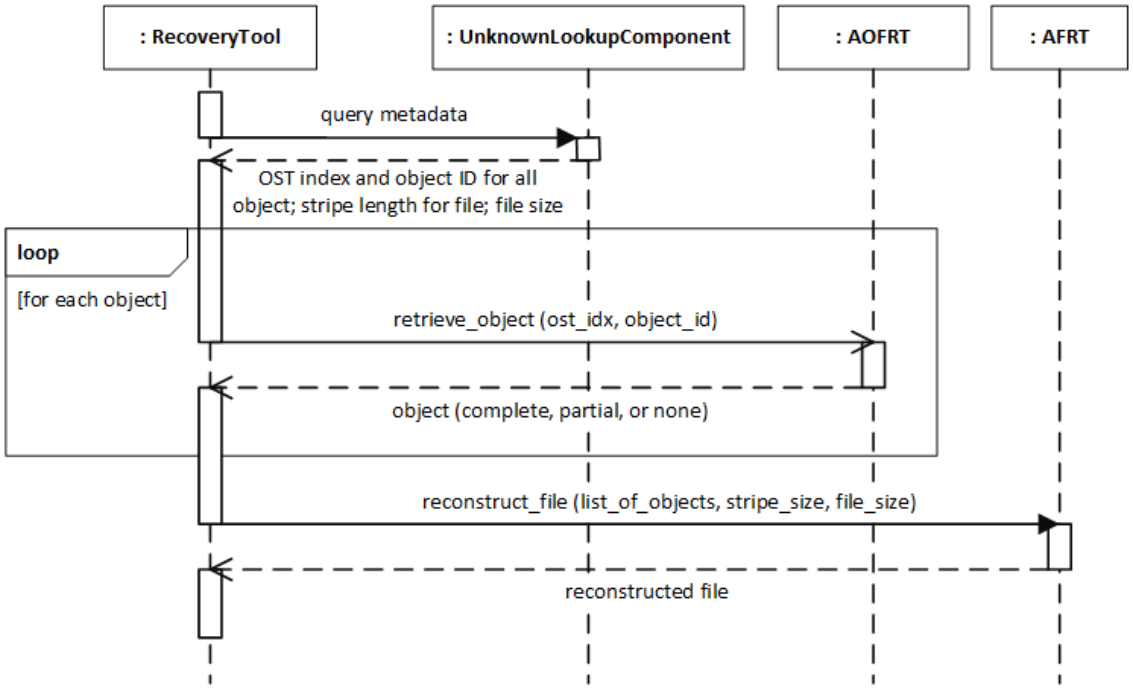

Figure 29. The introduction of the AFRT results in the need for the recovery machine to provide the OST index and object ID for all objects, the stripe length of the file, and the file size in order to recover the file.

[38] The keen reader will note that these are the same pieces of data required by the client, as presented in the **Object Storage & Striping** section of this paper, to reconstruct a file from the objects the file is composed of.

With the inclusion of the AFRT, the following information is needed in order to recover a deleted file:

1. The OST index and object ID for each object that make up the file (this list must be ordered); this information is needed by the AOFRT in order to recover the object files from the OSTs
2. The stripe size of the stripes that make up the file; this information, along with the ordered list of recovered objects from the AOFRT, is needed by the AFRT in order to reconstruct the file
3. The size of the file; this information is needed by the AFRT to know that it has completed reconstruction of the file (the correct number of bytes have been read from the objects)

Note that the list of objects that is supplied to the AFRT must be ordered (so that the AFRT can simply iterate through the objects, using the stripe size to obtain data from each object, until all available data is exhausted, as denoted by reading a number of bytes equal to the file size), and thus, the ordering of the objects must also be known. Therefore, the following information is needed from the `UnknownLookupComponent`:

1. The OST index and object ID of all objects that make up the file
2. The order of the objects
3. The stripe length for the file
4. The size of the file

Viewing this problem from the perspective of modules or components, it is natural to question which component or group of components in the Lustre file system contains the needed pieces of information. In actuality, there is a single component that stores this information: the MDS. The MDS is responsible for storing this information during the normal operation of a Lustre cluster in order to provide a client with the needed metadata to access a file. Therefore, it is natural to seek out this information from the MDS, allowing the analysis machine to obtain all the information required to recover the file from a single source.

Before a file is deleted, the MDS represents the files in the Lustre file system as inodes in its associated MDT and stores the mapping information for the objects of the file in an extended attribute associated with the FID of the file (for more information on this storage scheme, see the **MDS VFS Implementation** section of this paper). When a file is deleted in the Lustre file system, its associated metadata is also deleted [22]. Therefore, this metadata no longer exists on the MDT. Bearing in mind that the MDT is simply a local file system on which data is stored, it is therefore possible to recover the inode and associated xattr information for the deleted file. This identically mirrors the process used by the AOFRT: The metadata inode for the deleted file is recovered from the backing file system of the MDT, mirroring the recovery of the object file from the backing file system of the OST by the AOFRT.

This process can be abstracted into a component called the Abstract Metadata Recovery Tool (ABRT). Although this appears to be a separate component in this solution, it is identical to that of the AOFRT: Both are simply abstraction of the known methods for recovering files from a local file system. Using the AMRT, the sequence diagram for recovering a file reduces to the diagram illustrated in **Figure 30**.

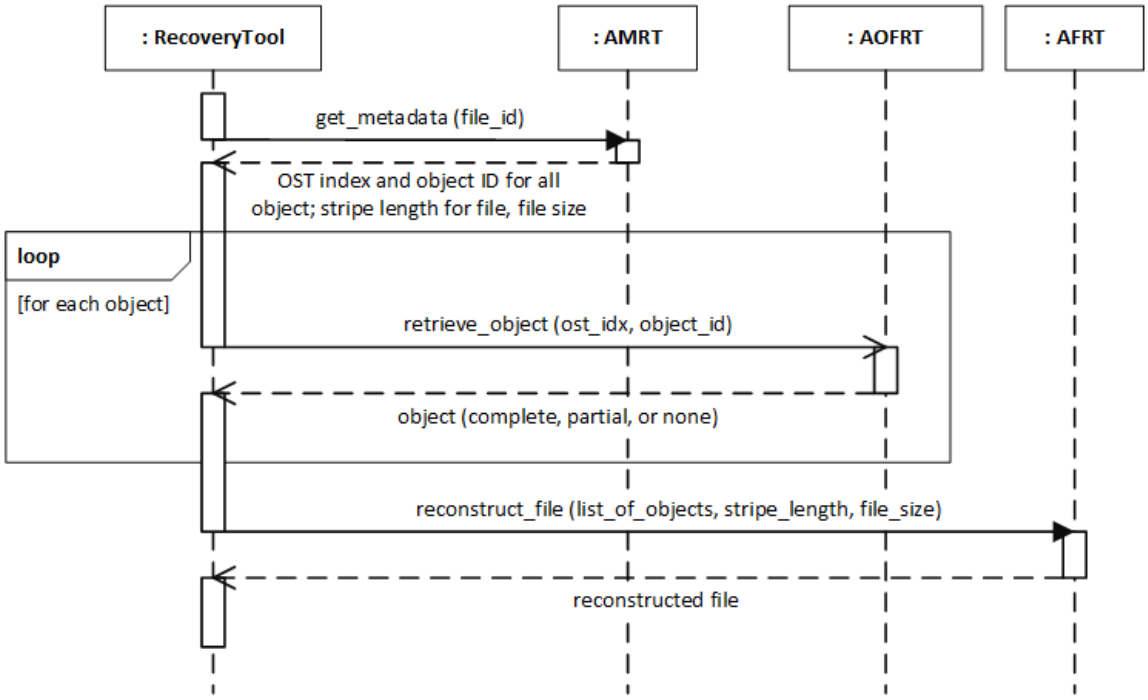

Figure 30. The introduction of the AMRT results in a three-step solution: (1) recover the file metadata from the MDT, (2) recover the objects corresponding to the file from the OSTs, and (3) reconstruct the deleted file using the recovered metadata and objects.

Using the combination of the AMRT, AOFRT, and the AFRT, all the information needed to recover a file in Lustre file system can be obtained. The resulting solution requires three major steps: (1) recovering the metadata from the MDT using the AMRT, (2) recovering all of the objects from the OSTs using the AOFRT, and (3) reconstructing the file from the metadata obtained from the AMRT and objects obtained from the AOFRT, using the AFRT. Thus, a file can be recovered from the Lustre file system by simply providing a reference or identifier for the file (such as the FID or path, depending on the mechanism used to implement this abstract solution).

Although it may appear that the devised solution simply abstracts the actual recovery of a file into unobtainable components (e.g. the solution is still incomplete because this paper does not suggest how to create such components or how such components can be devised to make the solution possible), this is not the case. In essence, what is done through the devised solution is to the dissolve the problem into smaller problems, for which solutions have already been created. Namely,

- The problem of obtaining the metadata for a file (finding the ordered locations of the objects that make up the file) is deconstructed into the recovery of inodes and extended attributes on a local file system (the backing file system for the MDT), for which solutions have already been created
- The problem of obtaining the objects for a file is deconstructed into the recovery of inodes and data blocks of a file on a local file system, for which (as is the case with the recovery of metadata) solutions have already been created
- The problem of reconstructing a file from its metadata and ordered list of object is deconstructed into creating a file from Lustre metadata and ordered objects, for which the Lustre client code base has already devised a solution (which is used by Lustre clients to access and present the file to end-users during the normal operation of the Lustre file system.

Viewed a different way, the problem is translated into the scheme used by the Lustre file system during normal operation, as illustrated in **Figure 3**. Instead of simply retrieving metadata from the MDT and retrieving the objects

for the file from the OSTs, a recovery tool is used before the MDT and OSTs to recover the now-deleted file. This scheme is illustrated below in **Figure 31**.

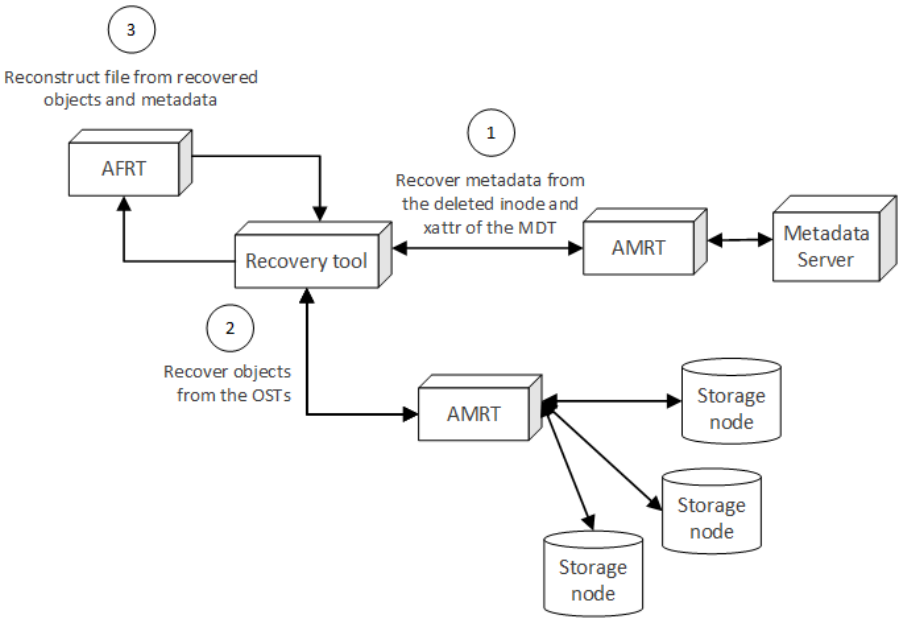

Figure 31. The Three-Step Recovery Solution can be framed as an alteration of the logic used by the Lustre file system in normal operation, where each piece of data required by the client must be recovered from the target on which the data resided prior to the deletion of the file to be recovered.

Using either perspective, the Three-Step Recovery Solution is a product of decomposing the recovery of a deleted file in the Lustre file system into three problems for which solutions have already been devised: (1) recovering the metadata for a file from the local, backing file system of the MDT, (2) recovering the objects associated with the deleted file from the local, backing file system of the OSTs on which the objects reside, and (3) reconstructing the file from its component objects using the logic contained in the client software for Lustre.

5.3 Solution Architecture

While the solution presented in **Figure 31** provides a simple, conceptual understanding of how to recover a deleted file on a Lustre file system, improvements to this basic model can be made. Being that a Lustre file system is distributed by nature, the solution to this recovery problem can be distributed as well, greatly reducing the overall workload on the recovery tool, as well as increasing the speed with which large, widely striped files[39] can be recovered. Instead of viewing the AFRT as a single, indivisible unit used by the recovery tool, where all the objects that make up a file must be present, the process of reconstructing a file can be broken down into partial reconstructions. For example, instead of reconstructing a complete file from a complete listing of all objects, a partial file can be constructed from a partial list of objects that make up the file. These partial files can then be combined and a larger, but still partial file, can be produced. When all partial components of the file are collected together, the complete, recovered file results.

In order to distinguish the indivisible AFRT from this partial file recovery tool, the logic for creating partial files from a partial list of objects can be culminated into a component called the Partial Striping Component (PSC). Just

[39] The term *widely striped* is used to denote a file that is striped across many OSTs, and therefore, a large number of objects (and likewise, stripes) exist for this file.

as with the AFRT, given the stripe size, file size, and an ordered list of object-to-OST mappings, the stripes corresponding to an object can be retrieved. For example, if given an object-to-OST mapping of

Object 0 → OST 1
Object 1 → OST 4
Object 2 → OST 7
Object 3 → OST 17

a stripe size of 5 MB, and a total file size of 90 MB (the first megabyte being megabyte 0 and the last megabyte being megabyte 89), a PSC given object 2 and this metadata would be able to compute that object 2 holds the following stripes (starting with index 0 as the first stripe):

Stripe 1: $[5\ MB, 10\ MB)$
Stripe 5: $[25\ MB, 30\ MB)$
Stripe 9: $[45\ MB, 50\ MB)$
Stripe 13: $[65\ MB, 70\ MB)$
Stripe 17: $[85\ MB, 90\ MB)$

Since Lustre uses round-robin striping, finding the stripes contained within a given object (knowing the index of object in the ordered list of objects, the stripe size, and the file size) can be visualized as a table, where each column represents an object over which a file is striped and where each row represents a full rotation of the round-robin algorithm (placing one stripe on each object, starting at the first object and ending at the final object). This visualization is illustrated below in **Figure 32**.

Object 1 on OST 1	**Object 2 on OST 4**	Object 3 on OST 7	Object 4 on OST 17
Stripe 0: [0, 5)	**Stripe 1: [5, 10)**	Stripe 2: [10, 15)	Stripe 3: [15, 20)
Stripe 4: [20, 25)	**Stripe 5: [25, 30)**	Stripe 6: [30, 35)	Stripe 7: [35, 40)
Stripe 8: [40, 45)	**Stripe 9: [45, 50)**	Stripe 10: [50, 55)	Stripe 11: [55, 60)
Stripe 12: [60, 65)	**Stripe 13: [65, 70)**	Stripe 14: [70, 75)	Stripe 15: [75, 80)
Stripe 16: [80, 85)	**Stripe 17: [85, 90)**		

Figure 32. Given an object, the index of the object, the stripe size for the object of which the object is a part, and the file size for which the object is a part, the stripes for an object can be discovered by viewing the round-robin striping as a table (stripe components given in MB).

Given this algorithm, a PSC can extract the stripes for a file (and calculate the index of each stripe with respect to its sequential placement in the file of which the stripe is a part) given an object, the index of the object, the stripe size, and the file size. With these inputs, the PSC can produce the stripes contained in the object, keyed by the index of that stripe in the file of which the object is a part. This key-value pairing of stripe index to stripe can be thought of as a partial file, containing only a subset of the stripes of the complete file. If this partial file is combined with all other partial files (for the file of interest), the complete file can be obtained.

For example, given the object-to-OST mappings, stripe size, and file size above, a PSC for each of the objects would produce the stripes of the file corresponding to each object, keyed by the stripe index (a partial file for each of the objects). If these partial files are combined, and ordered by stripe index, the complete file is obtained. This aggregation of partial files into the complete file is illustrated below in **Figure 33**.

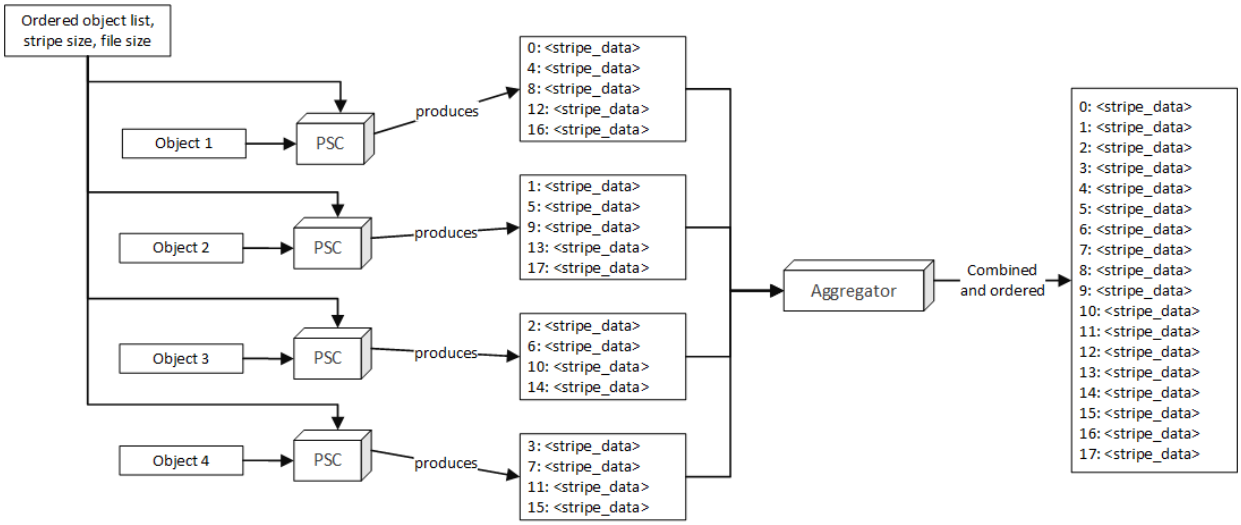

Figure 33. By providing each PSC with an object and file metadata for the file to be recovered, the each PSC can create a partial file, keyed by stripe index, which can be combined by an external entity into a complete file.

This process of creating partial files by means of mapping the stripes of an object into key-value pairs, keyed by the stripe index, and then ordering the partial files by stripe index is not required to be a two-step process: Partial files can be combined into larger partial files, which are then combined again with other partial files until the complete file is produced. This process is illustrated in **Figure 34**.

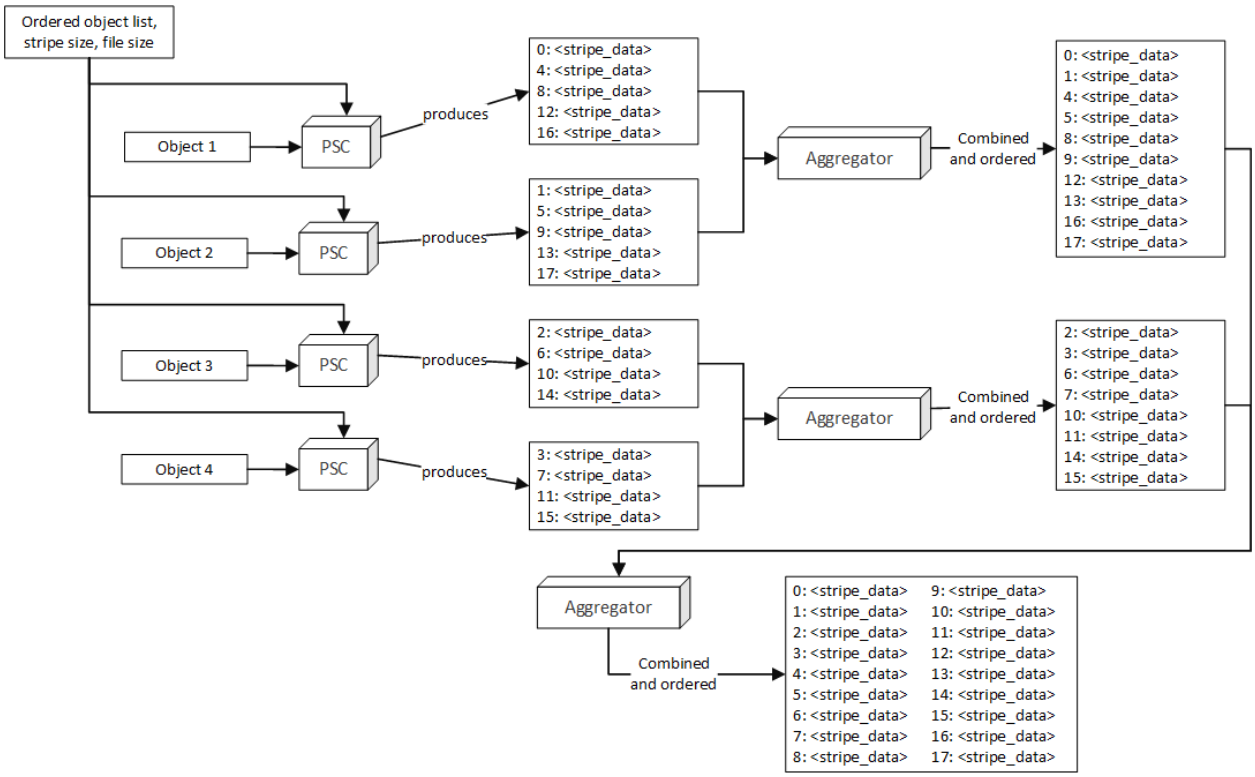

Figure 34. The partial files can be combined by any number of aggregators, allowing the aggregation process to scale depending on the number of objects and PSCs used to reconstruct the file.

By logical extension, this process can be replicated for any number of partial files and aggregation steps. Due to nature of this process, MapReduce is a natural solution architecture that allows the distribution of the combination process, while maintaining the overall consistency of the partial files as they are propagated along into a complete file. For example, the creating of partial files by the PSCs can be viewed as a mapping of the stripes from the recovered files and the combination and ordering of these partial files into further partial files (and ultimately to the complete file) can be viewed as the reduction step. Note that there is no reduction in the sense that the data size is reduced during the reduce job: Instead, the partial files are combined until a single, complete file is obtained. The solution architecture, using MapReduce to obtain a complete recovered file, is illustrated below in **Figure 35**.

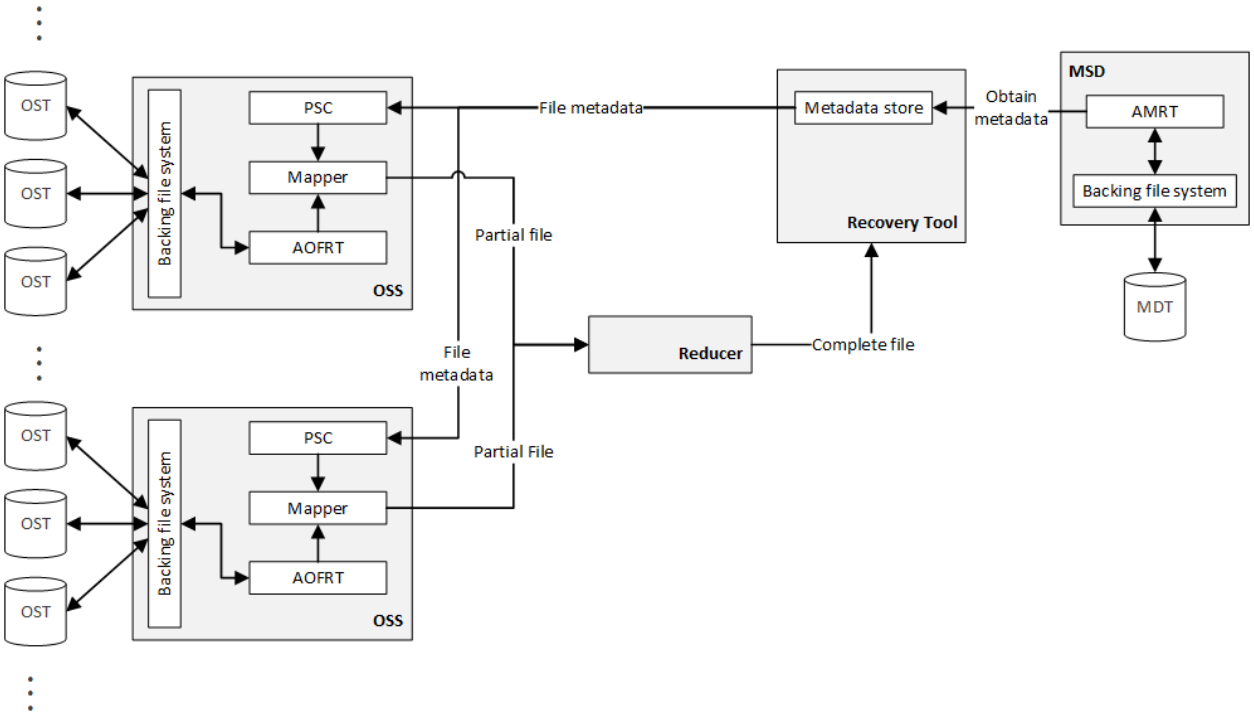

Figure 35. By leveraging the PSC and AOFRT on each of the OSSs, along with the metadata from the AMRT, partial files can be created (map job) and can be combined (reduce job) by any number of reducers until the complete file is reconstructed and propagated back to the recovery tool.

Before the MapReduce process can be begin, the metadata for the file to be recovered must be retrieved from the MDT. This step is accomplished in the same manner as before, using the AMRT to recover the inode and extended attributes representing the file to be recovered. This metadata is sent by the AMRT to the recovery tool, which stores this data in a metadata store. Now having the metadata for the file, the MapReduce job can be initiated. On each of the OSSs, the PSC retrieves the metadata for the file to be recovered. If the PSC recognizes that any of the objects exist on its OSS (that is, if any of the objects exist on the OSTs connected to the OSS on which the PSC is located), the AOFRT recovers these objects from the backing file system. These recovered objects are then sent to the mapper, which in combination with the metadata from the PSC, produces the stripes for the partial file, keyed by the stripe index[40]. This data is then sent from the mapper to the reducer, which combines the partial files into a complete file, which is then returned to the recovery tool.

[40] Following the MapReduce architecture, the mapper can be used to obtain the stripe indices and strip data associated with these indices and create the partial file, which is then sent to the reducer to be combined with other partial files until the complete file is recovered.

Note that although **Figure 35** only shows a single reducer, this MapReduce job can be spread across any number of compute nodes. For example, if a file is striped across hundreds of OSTs, then there may exist tens or hundreds of reducers that combine the partial files into the complete file. This configuration is a matter of preference and can be tuned to meet the needs of the file system and recovery process in question. Using this MapReduce method, the objects of a file are decomposed into their respective stripes, which allows the reducer to combine them with a much greater level of granularity: While objects cannot be fitted together into a complete file, stripes can be rearranged into a complete file, and therefore, the level of granularity is shifted from the objects existing on the OSTs to their corresponding stripes.

In essence, this MapReduce solution architecture is an algorithm for combining the objects of a file into a single file. While Lustre does this through the client node (which retrieves the metadata for a file and then retrieves the objects for that file from their respective OSTs), this method gathers the objects in-place and propagates the partial components of the file back to the recovery tool. This method has one main advantage over the Lustre-client method of reconstructing a file: The OSTs on which the objects of a file reside do not need to be mounted directly to the recovery tool. Instead of having all OSTs on which an object for the file exists be mounted to the recovery tool, the recovery tool simply executes a MapReduce job, contacting the PSC already on the OSS (to which the OSTs can be mounted offline), and the PSCs (through the reducer compute nodes) propagate the file fragments back to the recovery tool until the complete file is assembled. Using this technique, even large-scale, widely striped files can be recovered with the same logical simplicity as a small, concentrated file. In short, this solution architecture allows the recovery process to scale to meet the needs of any file, both large and small.

6. Conclusion

With the increased interest by both industry and academia in cloud computing and distributed software, high-throughput, distributed file systems such as the Lustre file system have become an essential part of network infrastructure. Similar to the hard disk of a localize computer system, the file system is often the greatest impediment to speed and performance in a distributed cluster. In order to overcome this hurdle, distributed file systems spread the data of a file system across disperse regions of a network, allowing clients to access files (enormous files by standards of a layman user) with aggregate I/O rates of petabytes per seconds. Since its inception in 1999, the Lustre file system has been foremost among these file systems, and has since been included as a core file system component in over 60% of the TOP100 HPCs worldwide.

While research into the distributed file system field has grown exponentially in recent years, this research has far outpaced research into supporting fields, such as forensics on file system where the files exist on many, disparate nodes in a network. This paper serves the purpose of filling this gap and contributing to this fledgling field. Although a solution to the problem of file recovery in a Lustre file system is presented in this paper, research in such topics is far from complete.

The main problem with file recovery on distributed file systems such as Lustre is that the objects that make up a file are spread throughout a network, and the metadata that accompanies these objects (such as where the objects for a file reside) is also spread throughout the same network, but is not co-located with these objects. Therefore, any effort to recover a file requires that the recovery instrument first obtain the metadata of the file of interest, retrieve the objects that make up the file (using the obtained metadata), and then reconstruct the file from its constituent objects. While this process is conceptually simple, there are many challenges and idiosyncrasies that exist during each step of this process, further complicating the already complex interactions in a distributed file system.

The solution presented in this paper, termed the Three-Step Recovery Solution, takes these complexities and decomposed them into problems already researched and solved. In particular, the Three-Step Solution reduces the recovery of both the metadata and objects associated with a file into the recovery of an inode and file on the MDT and OSTs, respectively, of a Lustre cluster (for which numerous solutions have already been devised). Lastly, the reconstruction of a file system is deferred to the client software of the Lustre file system, which already performs this operation in normal use (when a file is accessed and must be reconstructed from its constituent objects). In essence, the solution for the recovery of a deleted file on a Lustre file system is a series of already-researched solutions, which can be aggregated together to form a coherent, single solution to a very relevant problem.

Throughout the research surrounding this solution, a great deal was learned about both distributed file systems, as well as Lustre in particular. Specifically, there are a great deal of intricacies within the Lustre community, foremost among these is the strikingly lack of relevant and pertinent documentation. The Lustre Operations Manual (see [21]) is the definitive source on the Lustre file system and is an essential document in the portfolio of any Lustre administrator, but it is underwhelming in its detail. For example, while the manual provides information about the layout of the Lustre file system (such as the interconnections between the server and client nodes in a Lustre cluster), it lacks any detailed technical information about the inner-workings of each node. Serving its primary purpose as an operations manual, this document is not intended to be a technical description of the underpinnings of Lustre, and therefore is insufficient as a technical guide for developers.

While the operations manual lacks many technical details, another document exists as a technical supplement: *Understanding Lustre File System Internals* (see [22]). While this document contains a great deal of technical information about Lustre, is has two major draw-backs: (1) it requires a detailed, in-depth understanding of the Linux VFS and file system structures, without which many of the details in the document are overwhelming, and (2) the document was written in 2009, and many major changes have been made to Lustre since this time.

With regard to the first point, a thorough understanding of the Linux file system structure must be achieved prior to reading this document, thus extending the time required to understand the details of the document. While this appears to be a matter of time, there is a striking lack of visual material supporting the Linux VFS: Many of the descriptions of the VFS, although accurate and detailed, are strictly textual and do not provide a highly intuitive understanding of the VFS. The background material presented in paper is a direct result of the textual-centered nature of the Linux VFS descriptions; the proportionally large number of figures in the background description of

the Linux VFS contained in this document is due in large part to the lack of imagery found in both books on the Linux kernel and the literature surrounding the VFS.

Regarding the second point, both anecdotal and factual evidence support this observation. After communicating with a researcher at ORNL about the document, it was discovered that the *Understanding Lustre File System Internals* document is quite old by Lustre standards and a large portion of the data contained within it is not verifiably correct in the context of the Lustre of today. Although many of the fundamentals contained within the document are still true (after independently corroborating the information with other sources and the Lustre source code), not all of the details presented can be trusted as accurate in the latest releases of Lustre. Compounding this, large feature sets have been added to Lustre since 2009, rendering the document wholly insufficient for gaining a complete understanding of the technical details of the latest Lustre file system releases.

In an effort to obtain the most up-to-date information about the Lustre file system, the Lustre source code is used as the authoritative source of technical information for the data presented in this paper. While this is a solid approach to gaining a deep technical understanding of Lustre, the source code is likewise insufficient in its commenting and source-code-level documentation, and therefore, static analysis is one of the only means of following the numerous call chains that make up the common Lustre use cases. In general, this is a slow and arduous process and requires a great deal of time in order to understand the simplest of concepts. In order to combat this deficiency, OpenSFS has contracted (SFS-DEV-005[41]) many of the key Lustre developers with documenting the Lustre code base (specifically the Lustre protocol in the case of SFS-DEV-005).

This lack of solid and centralized documentation creates both a steep learning curve, as well as disparity in the knowledge level of Lustre users: Novice or expert, without many degrees between these two poles. While Lustre is an amazing system and it a well-developed and well thought-out distributed file system, the documentation and conceptual support for the file system is greatly lacking. It is the humble hope of the author that when further research is conducted on the topic file recovery and forensic analysis of the Lustre file system, this document, in terms of the background information and solution, may provide a foundation from which to grow beyond the knowledge of the author and provide the reader with a guide into a field that can otherwise be overwhelming.

[41] http://wiki.opensfs.org/Contract_SFS-DEV-005

Glossary

Entry	Definition	Aliases
client	The entity that ultimately interacts with the file system through the common file system actions, such create, move, copy, and delete. The client is the node that the end-user directly interacts with, and therefore, Lustre must present a coherent file system that appears to be local to the machine (or machines) on which the client is running. Note that Lustre files do not exist on the client, but rather, an implementation of the Linux VFS, called llite, is used to present this coherent, seemingly local file system to the end-user (see also **llite**). Although llite is a part of the client, it is not the entirety of the client. Instead, the client represents the complete node, including operating system, kernel, etc., while llite encompasses the Lustre client VFS implementation used to interact with the Lustre cluster.	
dentry	A dentry (pronounced “dee-entry”), or directory entry, is a VFS data structure representing a single component in a path. For example, for a path `/home/lustre/`, three dentry objects are created: (1) one representing `/`, (2) one representing `home/`, and (3) one representing `lustre/`. Together, these dentries create a double-linked tree structure, where each dentry stores a reference to its parent and contains a list of references to its children dentries. Each dentries references an inode that represents the directory or file found at the specified location. For example, the inode referenced by the dentry in (3) is the directory found on the file system at the path `/home/lustre/`. Since dentries are frequently used when traversing a path, a dentry cache is created by the VFS that stores dentries that have been loaded from disk. This provides fast lookup when walking a path in a file system.	
FID	A file identifier used by the Lustre file system to uniquely identify files on all targets of a Lustre file system. While inode numbers uniquely identify the files within a single file system, the distributed nature of Lustre requires that files be uniquely identified across all nodes in the Lustre network. The FID is composed of a 64-bit unique sequence, followed by a 32-bit OID, followed by a 32-bit version number [21].	
inode	An inode is the VFS data structure that maintains the metadata about a file in the Linux VFS. This metadata includes the mode, or permissions, of the file, the last time of access, pointers to the data blocks on disk that make up the file, and any user-defined extended attributes. Extended attributes are key-value pairs that are custom attributes that can be specified by the user outside of the default (standardized) inode fields that are included by the VFS. It is important to note that not all fields in an inode must be specified. In the case of the Lustre file system, the data block pointers of the inode are not used, since the data that make up a file in a Lustre file system are not located on the local disk of the machine accessing the file, but rather, distributed throughout the OSTs in the Lustre cluster. Secondly, it is important to note that the name of a file is not contained in the inode structure, but rather, in a dentry object which points to the inode. Inodes in a Linux file system are uniquely identified by an inode number, but in the case of the Lustre file system, all files represented by inodes are uniquely identified by a file identifier, or FID, that is globally unique among	

	all target nodes in a Lustre cluster, not just the local node on which the inode resides. This FID is also used as the key through which the extended attributes of the inode are obtained in the Lustre file system (see also **FID**).	
llite	The Lustre client implementation of the Linux VFS. This client implementation is responsible for providing the end-user with the appearance of a local file system, while concurrently abstracting the details of distributed file interactions with the Lustre cluster. The common operations performed by llite are retrieval of metadata from the MDS and retrieval of objects from the OSTs storing the objects for a file. For example, when a file is opened, llite first retrieves the metadata for the file, including the identifiers of the OSTs on which the objects for the opened file are located. Using this information, llite then contacts the OSTs containing these objects. Once all objects for the file have been retrieved, llite then reconstructs the file using the object and the stripe metadata.	
MDS	Metadata server that is responsible for managing the metadata in a Lustre file system, such as the mappings from object to OST for a file, as well as the stripe size for a file. This server is the first point of contact when a client wishes to interact with a file: The client will contain the MDS in order to obtain the metadata associated with a file and then proceed to interact with the OSSs and OSTs containing the objects that make up the file. The MDS is known as the single point in the Lustre file system that contains the metadata for files, and therefore, is frequented often in a Lustre cluster (the MDS should be a node with a great deal of network bandwidth to support these frequent interactions with numerous clients).	
MDT	Metadata target that stores the metadata information associated with the files in a Lustre file system. The files in a Lustre cluster exist as inodes on the MDT, where the object to OST mappings and other Lustre file information are contained in the extended attributes of the inode. These extended attributes are referenced by the FID of the file (see also **FID**).	
MGS	Management server in a Lustre cluster that is responsible for storing configuration and other managerial data for the Lustre file system. On small Lustre file systems, the MGS and MDS are commonly co-located on the same network node.	
MGT	The target associated with a management server. This target stores the management data using a backing (local) file system.	
object	In the context of the Lustre file system (and other object-based file systems), an object is the segment of a file stored on the OSTs of a Lustre cluster. An object in the Lustre file system should not be confused with an object (an instantiated class) in OOP.	
OSS	Storage server used to manage the I/O transfer of data for a series of OSTs in a Lustre cluster. The OSS is responsible for acting as a proxy on behalf of the OSTs managed by the OST and interacts directly with the other components in a Lustre cluster.	
OST	Lustre file system node on which the objects that make a file and acts as the storage unit for the Lustre cluster. The OST is under the responsibility of the OSS and is contacted through the OSS by a client when a client wishes to retrieve an object from the OST. The index of an OST acts as the unique identifier for the OST and must be globally unique among all OSTs in a Lustre cluster (not just among the OSTs attached to the same OSS).	
striping	Striping is the process of distributing the parts of a file across multiple nodes in a distributed file system. For example, if there are ten parts to a file, and	

	five nodes on which the file is to be stored, a round-robin striping pattern would place the first part on the first node, the second part on the second node, and so forth, until the fifth part is placed on the fifth node. Then, the sixth part is placed on the first node, and this pattern repeated until there are no more parts to place on the nodes assigned to the file. Due to the nature of this technique, the parts of the file stored on the distributed nodes are referred to as stripes. Apart from distributed file system, striping is also common in local disk file systems, where stripes over a file are dispersed among the disks of a local file system (this technique commonly uses RAID 0 or another type of striping RAID algorithm). Note that striping does not by itself provide redundancy: Only one instance of a stripe exists in the file system.	
superblock	The primary data structure in the Linux VFS. This data structure contains the metadata for the entire file system, including the status and size of the file system. This data structure obtains its name mainly because it is the primary block that is stored on a local file system (similar to the bootloader of a bootable disk). The superblock contains references to the other important data structures in the Linux VFS, including the list of file system inodes.	
VFS	The Linux VFS is the file system interface to the kernel that all file systems designed for Linux are required to implement. This interface provides a uniform abstraction through which the kernel is able to interact with the file systems mounted to a Linux installation, regardless of the implementation details of the mounted file system. The VFS provides the major data structures used throughout Linux file systems, including the superblock, inode, and dentry. Although the Linux kernel is implemented in the C programming language, the VFS is strikingly similar to a class hierarchy in OOP languages. For example, the functions that operate on data structures are overridden by the implementing file system, allowing the implementing file system to operate on the basic VFS data structures in a manner that is specific to the needs of that file system. This technique, although implemented using function pointers and function pointer tables, is analogous to overriding the methods of a base class when implementing an interface.	

Acronyms & Abbreviations

Entry	Expanded Phrase
ABC	Abstract Base Class
AFRT	Abstract File Reconstruction Tool
AMRT	Abstract Metadata Recovery Tool
AOFRT	Abstract Object File Recovery Tool
API	Application Programming Interface
CFS	Cluster File System
CLI	Command Line Interface
CRUSH	Controlled Replication Under Scalable Hashing
DEC	Digital Equipment Corporation
DoE	[US] Department of Energy
EA	Extended Attribute (see also xattr)
ECSSE	Electrical, Computer, Software, and Systems Engineering [Department at ERAU]
ERAU	Embry-Riddle Aeronautical University
Ext4	Fourth Extended File System
FAL	File Access Listener
FID	File Identifier (ID)
FSFilt	File System Filter
GPG	GNU Privacy Guard
GRP	Graduate Research Project
GUI	Graphical User Interface
HPC	High Performance Computer (or High Performance Computing)
I/O	Input/Output
IP	Internet Protocol
ISO	International Organization for Standardization
Ldiskfs	Lustre Disk File System
LDLM	Lustre Distributed Lock Manager
llite	Lustre Lite
llog	Lustre Log
LNET	Lustre Network (or Lustre Networking)
LOV	Logical Object Volume
MDC	Metadata Client
MDS	Metadata Server
MDT	Metadata Target
MGS	Management Server
MGT	Management Target
NAT	Network Address Translation
NFS	Network File System
OBD	Object Based Disk
ODBFS	Object-Based File System
OID	Object Identifier (ID)

OOP	Object-Oriented Programming
OpenSFS	Open Scalable File Systems
ORNL	Oak Ridge National Laboratory
OSC	Object Storage Client
OSS	Object Storage Server
OST	Object Storage Target
POSIX	Portable Operating System Interface
PSC	Partial Striping Component
RAID	Redundant Array of Independent Disks (or Redundant Array of Inexpensive Disks)
RPC	Remote Procedure Call
RPM	RedHat Package Manager
SELinux	Security-Enhanced Linux
SHA	Secure Hash Algorithm
symlink	Symbolic Link
TCP	Transmission Control Protocol
VFS	[Linux] Virtual File System (sometimes Virtual File Switch)
VM	Virtual Machine
xattr	Extended Attributes
ZFS	Z File System

References

[1] Jones, M. Tim. "Network File Systems and Linux." *Network File Systems and Linux*. IBM DeveloperWorks, 10 Nov. 2010. Web. 02 Apr. 2015. <http://www.ibm.com/developerworks/library/l-network-filesystems/>.

[2] Braam, Peter J. “Lustre, The Inter-Galactic File System.” *Lawrence Livermore National Laboratory*. 4 Aug. 2002. Web. 2 Apr. 2015. < https://asc.llnl.gov/computing_resources/bluegenel/talks/braam.pdf>.

[3] Morgan, Timothy P. "DOE Doles out Cash to AMD, Whamcloud for Exascale Research." *The Register*. N.p., 11 July 2012. Web. 02 Apr. 2015. <http://www.theregister.co.uk/2012/07/11/doe_fastforward_amd_whamcloud/>.

[4] “Lustre® File System.” OpenSFS: The Lustre File System Community. Open Scalable File Systems, Inc., n.d. Web. 04 Dec. 2014. <http://opensfs.org/lustre/>.

[5] "Project 1: Layout Enhancement Design." Scope of Work: n. pag. OpenSFS: The Lustre File System Community. OpenFSF. Web. 26 Mar. 2015.

[6] "Layout Enhancement Scope Statement." (n.d.): n. pag. OpenSFS: The Lustre File System Community. OpenFSF, 10 Oct. 2013. Web. 26 Mar. 2015.

[7] “Layout Enhancement Solution Architecture.” OpenSFS: The Lustre File System Community. Open Scalable File Systems, Inc., 20 Dec. 2013. Web. 26 Mar. 2015.

[8] Hammond, John. “Layout Enhancement High Level Design.” Ed. Richard Henwood. OpenSFS: The Lustre File System Community. Open Scalable File Systems, Inc., 7 Feb. 2014. Web. 26 Mar. 2015.

[9] Lustre Release Git. N.p., n.d. Web. 28 Mar. 2015. <http://git.whamcloud.com/fs/lustre-release.git>.

[10] "Lustre: The Future of High Performance Computing." *OpenSFS: The Lustre File System Community*. N.p., n.d. Web. 3 Apr. 2015. <http://cdn.opensfs.org/wp-content/uploads/2013/10/lustre_infographic_nov2013.jpg>.

[11] "Lustre 2.6.0." *OpenSFS: The Lustre File System Community*. N.p., 12 Mar. 2015. Web. 03 Apr. 2015. <http://wiki.opensfs.org/Lustre_2.6.0>.

> Note that at the time of writing, the existing Lustre Wiki page was transitioned to `wiki.old.lustre.org`. Therefore, if the above link directs to a blank or incomplete page, replace `wiki.lustre.org` with `wiki.old.lustre.org`.

[12] "Lustre 2.7.0." *OpenSFS: The Lustre File System Community*. N.p., 13 Mar. 2015. Web. 03 Apr. 2015. <http://wiki.opensfs.org/Lustre_2.7.0>.

> Note that at the time of writing, the existing Lustre Wiki page was transitioned to `wiki.old.lustre.org`. Therefore, if the above link directs to a blank or incomplete page, replace `wiki.lustre.org` with `wiki.old.lustre.org`.

[13] Petersen, Torben K. *Inside The Lustre File System* (n.d.): n. pag. Seagate. Seagate. Web. 26 Mar. 2015. <http://www.seagate.com/files/www-content/solutions-content/cloud-systems-and-solutions/high-performance-computing/_shared/docs/clusterstor-inside-the-lustre-file-system-ti.pdf>.

[14] Wülfing, Britta. "Sun Assimilates Lustre Filesystem » Linux Magazine." *Linux Magazine*. N.p., 13 Sept. 2007. Web. 03 Apr. 2015. <http://www.linux-magazine.com/Online/News/Sun-Assimilates-Lustre-Filesystem?category=13402>.

[15] Montalbano, Elizabeth. "Update: Oracle Agrees to Buy Sun for $7.4B." *InfoWorld*. N.p., 20 Apr. 2009. Web. 3 Apr. 2015. <http%3A%2F%2Fwww.infoworld.com%2Farticle%2F2632056%2Fm-a%2Fupdate--oracle-agrees-to-buy-sun-for--7-4b.html>.

[16] Brueckner, Rich. "Inside Track: Oracle Has Kicked Lustre to the Curb - InsideHPC." *InsideHPC*. N.p., 10 Jan. 2011. Web. 03 Apr. 2015. <http://insidehpc.com/2011/01/inside-track-oracle-has-kicked-lustre-to-the-curb/>.

[17] Gorda, Brent. "Whamcloud Aims to Make Sure Lustre Has a Future in HPC - InsideHPC." *InsideHPC*. N.p., 20 Aug. 2010. Web. 03 Apr. 2015. <http://insidehpc.com/2010/08/whamcloud-aims-to-make-sure-lustre-has-a-future-in-hpc/>.

[18] "Whamcloud Signs Multi-Year Lustre Development and Performance Contract With OpenSFS." *Marketwire*. N.p., 16 Aug. 2011. Web. 03 Apr. 2015. <http://www.marketwired.com/press-release/whamcloud-signs-multi-year-lustre-development-and-performance-contract-with-opensfs-1549955.htm>.

[19] Jackson, Joab. "Intel Purchases Lustre Purveyor Whamcloud." *PCWorld*. N.p., 16 July 2012. Web. 03 Apr. 2015. <http://www.pcworld.com/article/259328/intel_purchases_lustre_purveyor_whamcloud.html>.

[20] “Xyratex Advances Lustre® Initiative, Assumes Ownership of Related Assets.” *Seagate*. N.p., 19 Feb. 2013. Web. 03 Apr. 2015. <http://www.xyratex.com/news/press-releases/xyratex-advances-lustre%C2%AE-initiative-assumes-ownership-related-assets>.

[21] "Lustre Software Release 2.x Operations Manual." (n.d.): n. pag. *HPDD Community Space Documentation*. Intel Corporation, 19 Mar. 2015. Web. 19 Mar. 2015. <https://build.hpdd.intel.com/job/lustre-manual/lastSuccessfulBuild/artifact/lustre_manual.pdf>.

> The Lustre Operations Manual is created in an automated build (continuous deployment), and therefore, the link contained in this citation may refer to a newer manual than was used in the research contained in this document. In order to find the exact operations manual used throughout this paper, see
>
> ```
> https://github.com/albanoj2/grp/blob/master/docs/Research/Lustre/Lustre%20Operations%20Manual%20Release%202-x.pdf
> ```

[22] Wang, Feiyl, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. "Understanding Lustre Filesystem Internals." (n.d.): n. pag. Oak Ridge Leadership Computing Facility. *Oak Ridge National Laboratory*, Apr. 2009. Web. 27 Mar. 2015. <http://users.nccs.gov/~fwang2/papers/lustre_report.pdf>.

[23] Weil, Sage A., Scott A. Brandt, Ethan Miller, and Carlos Maltzahn. *CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data.* Tampa, FL: IEEE, 2006. Storage Systems Research Center, University of California Santa Cruz. The Institute of Electrical and Electronics Engineers, Nov. 2006. Web. 26 Mar. 2015. <http://www.ssrc.ucsc.edu/Papers/weil-sc06.pdf>.

[24] Honicky, R. J., and Ethan L. Miller. *Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution*. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004) (2004): n. pag. Storage Systems Research Center. Jack Baskin School of Engineering, University of California, Santa Cruz, Apr. 2004. Web. 26 Mar. 2015. <http://www.ssrc.ucsc.edu/Papers/honicky-ipdps04.pdf>.

[25] Stallings, William. *Computer Organization & Architecture: Designing for Performance (9th Edition)*. Upper Saddle River, NJ: Prentice Hall, 2013. Print.

[26] Love, Robert. *Linux Kernel Development Second Edition*. N.p.: Sams, 2005. Sams Publishing, 12 Jan. 2005. Web. 27 Mar. 2015. <http://www.makelinux.net/books/lkd2/main>.

[27] Meyer, Bertrand. *Object-Oriented Software Construction*, 2nd Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1997. Print.

[28] Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel*. Beijing: O'Reilly, 2006. Print.

[29] Zhang, Yang. "Best Diagram to Explain How Linux VFS Works." *Yang Zhang*, 16 Apr. 2013. Web. 28 Mar. 2015. <http://www.yzhang.net/blog/2013-04-16-linux-vfs.html>.

[30] "Dentries and Inodes." Fieldses.org, n.d. Web. 28 Mar. 2015. <http://www.fieldses.org/~bfields/kernel/vfs.txt>.

[31] "Path Walking and Name Lookup Locking." *The Linux Kernel Archives*. Kernel.org, n.d. Web. 28 Mar. 2015. <https://www.kernel.org/doc/Documentation/filesystems/path-lookup.txt>.

[32] Djwong. "Ext4 Disk Layout." *Kernel.org Ext4 Wiki.* Kernel.org, 16 Mar. 2015. Web. 28 Mar. 2015. <https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout>.

[33] Rusling, David A. "The Linux Kernel." *The Linux Documentation Project*. N.p., 1999. Web. 28 Mar. 2015. <http://www.tldp.org/LDP/tlk/fs/filesystem.html>.

[34] Jones, M. Tim. "Anatomy of the Linux File System." *IBM Developer Works*. IBM, 30 Oct. 2007. Web. 28 Mar. 2015. <http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/>.

[35] Brouwer, Andries. "The Linux Kernel: The Linux Virtual File System." *The Linux Kernel: The Linux Virtual File System*. N.p., 1 Feb. 2003. Web. 28 Mar. 2015. <http://www.win.tue.nl/~aeb/linux/lk/lk-8.html>.

[36] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *6th Symposium on Operating Systems Design & Implementation (2004)*: n. pag. *Google.com*. Google, Inc., 2004. Web. 6 Apr. 2015. <http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf>.

[37] Ives, Z. "Cloud Case Studies & MapReduce: Shared-Nothing Parallelism." *Computer and Information Science*. University of Pennsylvania, Fall 2014. Web. 8 Apr. 2015. <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CB4QFjAA&url=http%3A%2F%2Fwww.cis.upenn.edu%2F~nets212%2Fslides%2F08-MapReduceIntro.pptx&ei=lF0lVeKEG8HCsAWTxoLwBA&usg=AFQjCNGI5tX0x9Wjeo8IATfOJ5Vx8MQ9gw&sig2=IqIjyt_rvnwxgUeRbhXoVQ>.

[38] “Lustre Enterprise Linux 6.6 x86_64 Server Packages.” *HPDD Community*. Intel Corporation, 19 Mar. 2015. Web. 19 Mar. 2015. <https://downloads.hpdd.intel.com/public/lustre/latest-feature-release/el6.6/server/RPMS/x86_64/>.

[39] “Lustre Enterprise Linux 6.6 x86_64 e2fsprogs Packages.” *HPDD Community*. Intel Corporation, 19 Mar. 2015. Web. 19 Mar. 2015. <https://downloads.hpdd.intel.com/public/e2fsprogs/1.42.12.wc1/el6/RPMS/x86_64/>.

[40] Ellingwood, Justin. "How To Edit the Sudoers File on Ubuntu and CentOS." *How To Edit the Sudoers File on Ubuntu and CentOS*. DigitalOcean, 17 Sept. 2013. Web. 19 Mar. 2015. <https://www.digitalocean.com/community/tutorials/how-to-edit-the-sudoers-file-on-ubuntu-and-centos>.

[41] Thompson, Kerry. "Crypt.Gen.NZ." *How to Disable SELinux*. Crypt.Gen.NZ, n.d. Web. 19 Mar. 2015. <http://www.crypt.gen.nz/selinux/disable_selinux.html>.

[42] Henwood,, Richard. "Testing a Lustre Filesystem." *HPDD Community Space Documentation*. Intel Corporation, 29 Oct. 2013. Web. 19 Mar. 2015. <https%3A%2F%2Fwiki.hpdd.intel.com%2Fdisplay%2FPUB%2FTesting%2Ba%2BLustre%2Bfilesystem>.

[43] "Locating a Hosted Virtual Machine's Files." VMWare Knowledge Base. VMWare, 29 July 2014. Web. 24 Mar. 2015. <http%3A%2F%2Fkb.vmware.com%2Fselfservice%2Fmicrosites%2Fsearch.do%3Flanguage%3Den_US%26cmd%3DdisplayKC%26externalId%3D1003880>.

[44] "How to Create a Lustre File System." How to Create a Lustre File System. New York University: Stern School of Business, n.d. Web. 25 Mar. 2015. <http://pages.stern.nyu.edu/~nwhite/scrc/createlustre.html>.

[45] Marks, Dusty. "[Lustre-discuss] Unable to Activate OST." Google Groups. Lustre-discuss Google Groups, 14 Jan. 2010. Web. 26 Mar. 2015. <https://groups.google.com/forum/#!topic/lustre-discuss-list/7NPC130LYXw>.

[46] G., Sunny, and Dmitry Eremin. "Compiling, Configuring and Running Lustre on Intel® Xeon Phi™ Coprocessor." Intel Developer Zone. Intel Corporation, 25 Nov. 2014. Web. 26 Mar. 2015. <https://software.intel.com/en-us/blogs/2014/11/06/lustre-on-intel-xeon-phi>.

[47] Atchley, Scott. "[Lustre-discuss] Lctl Ping Fails To/from the Client." Lustre Mailing Lists. Lustre, 14 Apr. 2007. Web. 26 Mar. 2015. <http%3A%2F%2Flists.lustre.org%2Fpipermail%2Flustre-discuss%2F2007-April%2F003223.html>.

[48] Ethrbunny. "Lustre - Issues with Simple Setup." ServerFault. StackExchange, 14 Sept. 2012. Web. 26 Mar. 2015. <http://serverfault.com/questions/427447/lustre-issues-with-simple-setup>.

[49] Eldar-aliyev8. "Ubuntu Documentation." IptablesHowTo. Ubuntu, 8 Feb. 2015. Web. 26 Mar. 2015. <https://help.ubuntu.com/community/IptablesHowTo>.

Appendix A: Original Research Proposal

The following is the original research proposal, as submitted by Justin Albano to Dr. Remzi Seker at the Department of Electrical, Computer, Software, and Systems Engineering at Embry-Riddle Aeronautical University on December 4, 2014. All content contained within has remained unchanged since submission.

Objective

To study and implement portions of the solution architecture and solution high level design proposed in [1] and [2], respectively, for the Layout Enhancement (LE) established in the Technical Proposal by High Performance Data Division of Intel for OpenSFS Contract SFS-DEV-003 as signed on Friday 23rd August, 2013. Through the research and application of these solutions, the intricacies of the design can be realized and vetted, ultimately leading to improvements in the design that may not have been foreseen except upon implementation. Likewise, this research may eventually lead to an alternative solution design that incorporates the improvements found during application of the LE solution architecture and LE high level design.

Background

The Lustre File System is a high-performance computing (HPC), POSIX-compliant distributed file system that is used on over 60% of the TOP100 sites, as recorded by Alexa [3]. While there are many open source distributed file systems available for use today, the architecture and design of the Lustre file system is particularly suited for extreme-throughput environments and platforms, and is capable of not only storing petabytes of data on its file system, but is also capable of providing terabytes per second of aggregate I/O bandwidth across the file system. Compounding the interest of many of the TOP100 sites with the capabilities provided, Lustre is quickly becoming the *de facto* standard for HPC file systems and, in turn, has gained increased support from companies such as Intel, providing a financial and technical means for improving the file system [3].

Problem Statement

While Lustre has proven to be a capable file system, there are many improvements that can be made to increase both the efficiency and simplicity of the system. The Lustre distributed file system is based on the concept of dividing a file into objects deposited on various storage nodes and in a network, and storing an accompanying manifest data structure (referred to as a layout in the Lustre nomenclature), that contains the location of the file objects within the network. Currently, there are four main areas of improvement that Lustre is seeking to develop: (1) file replication, (2) Redundant Array of Independent/Inexpensive Disks (RAID) support, (3) compaction for widely stripped files, and (4) handling large layouts.

In order to develop solutions to these outstanding problems, the High Performance Data Division of Intel signed a contract (SFS-DEV-003) with Open Scalable File Systems, Inc. (OpenSFS) to create a solution architecture, solution high level design, and implementation assessment for enhancements to the layouts used in Lustre (see [3] for the scope of these enhancements). At the time of writing, the solution architecture ([1]) and solution high level design ([2]) have been released, but an implementation assessment has yet to be created. This implementation assessment presents an opportunity for improvement to both the proposed architecture and high level design, and provides an avenue for a possible alternative design, leveraging the knowledge gained from the implementation of the proposed design as the basis for additions and modifications to the solution architecture and design.

References

[1] “Layout Enhancement Solution Architecture.” *OpenSFS: The Lustre File System Community*. Open Scalable File Systems, Inc., 20 Dec. 2013. Web. 30 Nov. 2014.

[2] Hammond, John. “Layout Enhancement High Level Design.” Ed. Richard Henwood. *OpenSFS: The Lustre File System Community*. Open Scalable File Systems, Inc., 7 Feb. 2014. Web. 30 Nov. 2014.

[3] “Lustre® File System.” OpenSFS: The Lustre File System Community. Open Scalable File Systems, Inc., n.d. Web. 04 Dec. 2014.

[4] “Layout Enhancement Scope Statement.” *OpenSFS: The Lustre File System Community*. Open Scalable File Systems, Inc., 10 Oct. 2013. Web. 30 Nov. 2014.

Appendix B: Incomplete Solutions

"I have not failed. I've just found 10,000 ways that won't work."

— Thomas Edison

In an effort to provide the reader with a perspective of the possible solutions that were devised prior to creating the Three-Step Recovery Solution, this section contains a description of other, incomplete solutions that were tried but were found to be infeasible or lacked enough information to provide a complete solution to the problem presented in this paper. It is the hope of the author that these incomplete solutions will provide the reader with insight into solutions that should either be avoided (if found to be dead-ends) or that require further research in order to complete.

While some of the incomplete solutions presented in this section are seemingly inefficient, lacking in pragmatism, or imbued with other such deficiencies (hence resulting in an incomplete solution), it is the objective of the author that the revelation of the failed solutions provides the reader with a perspective into the thought process of the author and the failed solutions that ultimately lead to the Three-Step Recovery Solution.

Aggregation of Lustre Log Files

As with many large-scale distributed systems, Lustre uses a distributed logging mechanism, which stores Lustre Log (llog) files on each of the server nodes in the Lustre cluster[42]. These llog files are transactionary files that store information about the file system, including deletion of files (referred to in this context as *unlinking*). According to [22], the deletion procedure for a file is as follows:

> First, the client decides to remove a file and this request is sent to MDS. MDS checks the EA striping and uses llog to make a transaction log. This log contains the following: `<unlink object 1 from ost1, unlink object 2 from ost2, etc.>`. Then, MDS sends this layout and transaction log back to the client. The client takes this log and contacts each OST (actually obdfilter) to unlink each file object. Each successful removal is accompanied by a confirmation or acknowledgment to the unlink llog record. Once all unlink llog records on MDS have been acknowledged, the file removal process is completed.

Based on this information, it is clear that upon deletion of a file, the llog for each of the OSTs on which the objects for that file exist will contain entries for the unlinking of the aforementioned objects. Viewing this from the top-level perspective, an aggregation of all llogs from all OSTs would provide a complete listing of all objects deleted from a Lustre file system and the location (OST on which the object resides) for all deleted objects.

Being that large-scale Lustre file systems often consist of hundreds or thousands of OSTs, the collection and subsequent filtering of all llogs to find object unlinking entries may appear to be a daunting task. Bearing this mind, there exist techniques specifically designed for problems such as these; for example, the MapReduce algorithm presented in [36][43]. Using MapReduce, the llogs from each of the OSTs can be gathered and filtered in the following manner:

- The llog file for each OST is consumed by the mapper for that node and a list of resulting `<object ID, OST ID>` key-value pairs are outputted by the mapper for that node. These mapped key-value pairs are then sent to the reducer.

[42] Clients in a Lustre cluster also have their respective logs, but for the sake of this discussion, these logs are ignored (the presented solution only focuses on the server llog files).

[43] While there are many well-known MapReduce implementations, such as Apache Hadoop, this section will not discuss the details of designing a solution in terms of a specific implementation, but rather, in terms of a generic MapReduce job.

- The reducer then consumes the individual lists of `<object ID, OST ID>` key-value pairs and combines them into a single, coherent list of all of `<object ID, OST ID>` key-value pairs.

Note that object IDs are unique and that no object with the same ID may exist on more than one OST at a given time. Therefore, there will never be a case where there will be two `<object ID, OST ID>` key-value pairs that point to different OSTs. Thus, the reduce portion of this MapReduce job is simply an aggregation function. This MapReduce job is illustrated below in **Figure 36**.

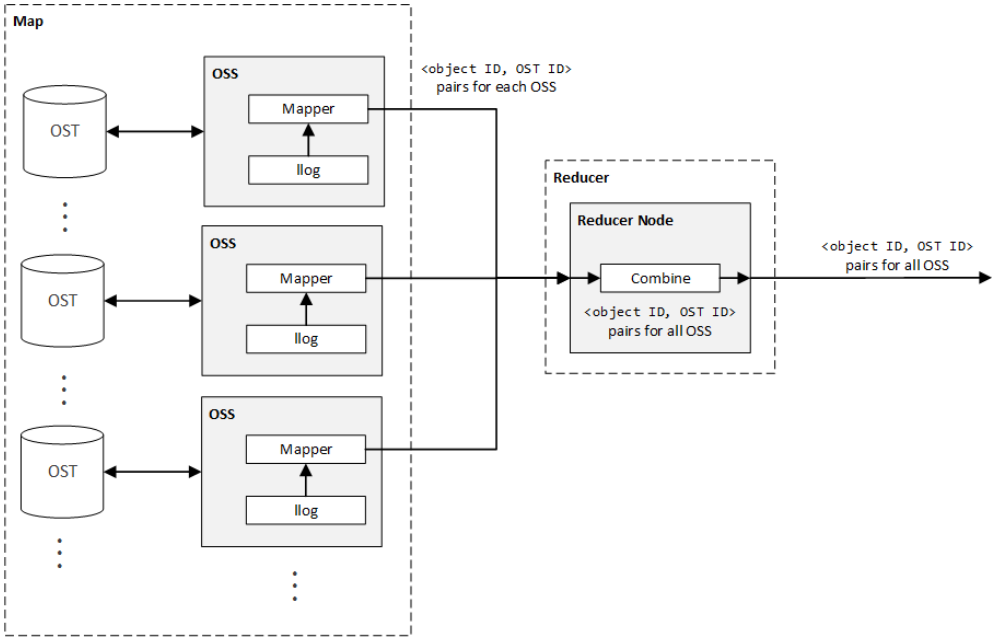

Figure 36. Using MapReduce as an aggregation mechanism (with a reducer that simply combines), a complete list of `<object ID, OST ID>` key-value pairs can be created for all deleted objects on a Lustre file system.

With this combined list of `<object ID, OST ID>` key-value pairs, all deleted objects are known, as well as on which OST these deleted objects reside. The major piece of information missing from this solution is the mapping of a file to the specific objects that make up the file. In essence, using the llog method, we obtain a list of all deleted objects in a Lustre file system, but do not have a means of mapping a deleted file to its constituent objects.

While this information can be found by recovering the inodes of the MDT for a file using the technique prescribed for the AMRT in the Three-Step Recovery Solution, the AMRT also produces the locations of the objects using the extended attributes of the recovered inodes. Thus, the combined list of `<object ID, OST ID>` key-value pairs that map an object to its corresponding OST is superfluous, as this information is already known from the AMRT for the specific file of interest. Therefore, in order for the MapReduce technique to be profitable, a mechanism is needed that produces only the file-to-object-ID mappings for a file to be recovered, rather than the full object-ID-to-OST-ID mappings for a file produced by the AMRT. This concept is illustrated in the **Figure 37** (on the following page).

Until a simplified solution to this issue is resolved, the aggregation of llogs technique results in a composite view of all deleted objects, but does not provide a mapping of file-to-objects that allows the specific objects for a file to be recovered and ultimately reconstructed.

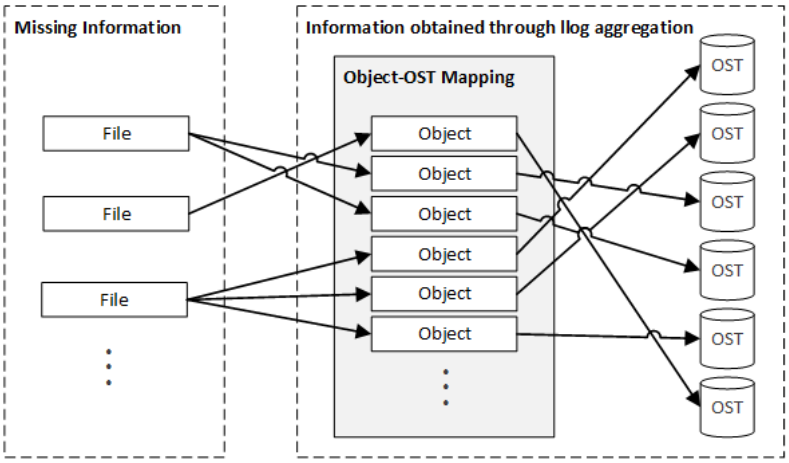

Figure 37. While the mapping of object-to-OST can be found using the llog aggregation technique, the mapping from file to objects is still unknown.

Offline Recording of Files

In order to provide a monitoring interface that external tools can use to observe changes to the metadata and namespace of a Lustre file system, Lustre includes what are referred to as changelogs[44]. This feature, added in Lustre 2.0, changelogs act as a transactionary log of the file-based events that occur in a Lustre file system, including the creation of files, deletion (unlinking) of files, and renaming of files. Taken to its logical extension, this monitoring API allows tools to mirror a Lustre file system, ensuring consistency between the mirrored file system and the original Lustre file system.

Each time such a metadata change or namespace change occurs in the Lustre file system, an event is added to the changelog, which includes the operation type (creation, deletion, etc.), the time- and date-stamp of the event, the FID of the target (the file on which the event occurred), the FID of the parent, and the target name [21]. For example, if a file named `something.txt` were to be created and then deleted, the following events would be published through the changelog API:

```
6 01CREAT 19:00:17.771142384 2015.04.05 0x0 t=[0x200000400:0x4:0x0]
p=[0x200000007:0x1:0x0] something.txt
--snip--
8 06UNLNK 19:00:38.146143558 2015.04.05 0x1 t=[0x200000400:0x4:0x0]
p=[0x200000007:0x1:0x0] something.txt
```

The first event (event 6) denotes the creation of the file, while the second event (event 8) denotes its deletion. While the changelog events do not carry with them the contents of the file at the time the event occurred, this information can be retrieved from the Lustre file system by retrieving the file. Therefore, when an event denoting an update to the contents of a file occurs, such as a creation or modification, a mirrored file system can retrieve the contents of the file and update the file on the mirrored system. While custom tools can be created to accomplish this, tools such as RobinHood[45] already exist, compiling the events in a changelog into a database [13].

[44] It is important to note that Lustre changelogs are not files contained within the project code base that track the changes made by various versions of Lustre over time. Instead, in the context of this section, changelogs refer to the changelogs API provided by Lustre for the purpose of file system monitoring.
[45] https://github.com/cea-hpc/robinhood/wiki

With the ability to mirror a Lustre file system, a possible solution is to instrument the deletion event, so that the mirror file system[46] retrieves the deleted file (prior to the file being deleted on the actual Lustre file system). For example, if possible, retrieve the deleted file before it is actually deleted from the Lustre file system. This instrumentation approach is illustrated below in **Figure 38**[47]. It is important to note that in order to receive change log updates, an outside entity must be registered to receive changelog events. This is accomplished through the `changelog_register` function provided by the Lustre API. This command must be executed as follows:

```
lctl --device <MDT_name_and_index> changelog_register
```

For more information on registering for changelog events and interacting with the changelog API of Lustre, see **Chapter 12: Monitoring a Lustre File System** of [21].

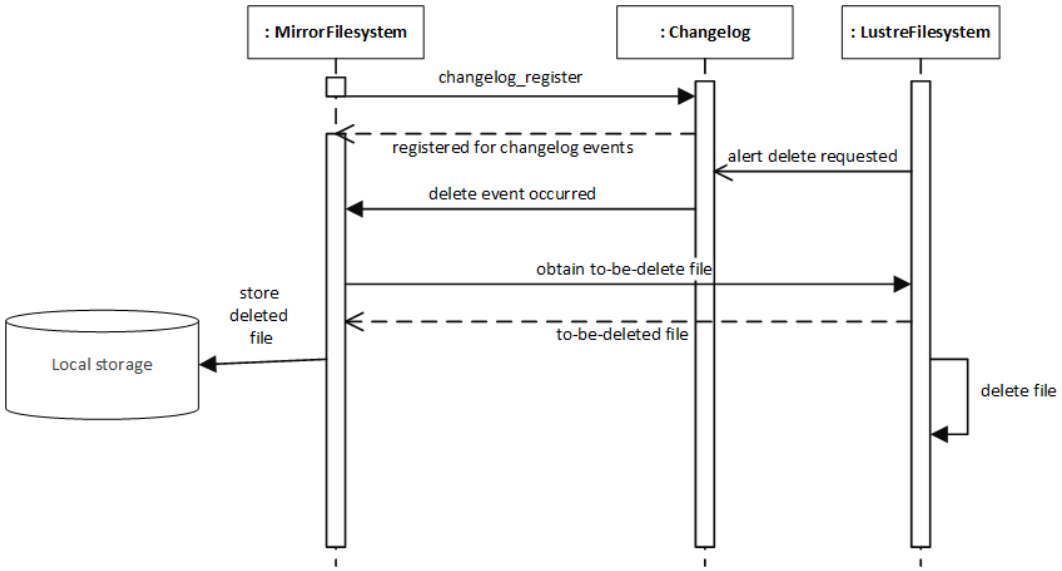

Figure 38. Before file are removed from the actual Lustre file system, a mirrored file system retrieves the file and stores it for later recovery.

Although this solution does not provide for the real-time recovery of files from a Lustre file system, it allows all deleted files to be stored on a mirrored file system and therefore, be recovered at a later time. Since this technique does must be implemented prior to the desire to recover a deleted file (the deleted files must be stored on the mirror file system prior to the need to recover deleted files), this approach is referred to as *offline recording*. In order to recover a file using this method, a recovery tool can simply obtain the deleted file from the mirrored file system,

[46] In this context, the term *mirrored file system* does not mean a fully-mirrored file system, were all files that exist on the Lustre file system exist on the mirrored file system, but rather, a file system in which only the files deleted form the Lustre file system exist on the mirrored file system. The term mirrored file system is maintained in order to denote the fact that the secondary file system is monitoring the events of the Lustre file system and mirroring the deletions of the file system (storing the files prior to the actual deletion from the Lustre file system).

[47] It is important to note that the figure, as depicted, illustrates that the changelog deletion event occurs prior to the file actually being deleted on the Lustre file system, but this is not necessarily the case. This assumption is made to demonstrate the possibility of this solution, given that the event occurs prior to the actual deletion of the file from the file system. Therefore, it is essential that the reader not assume that the events depicted in this figure actually occur in the order presented; rather, the reader should view this order as an assumption used to illustrate the possibility of the presented solution, given the assumption holds true (and a means of instrumenting the deletion process is found).

rather than performing forensic analysis on the live Lustre file system. In short, the mirror file system is responsible for storing all files deleted from the Lustre file system, allowing a recovery tool to simply obtain the deleted files from the mirror file system. This approach it illustrated below in **Figure 39**.

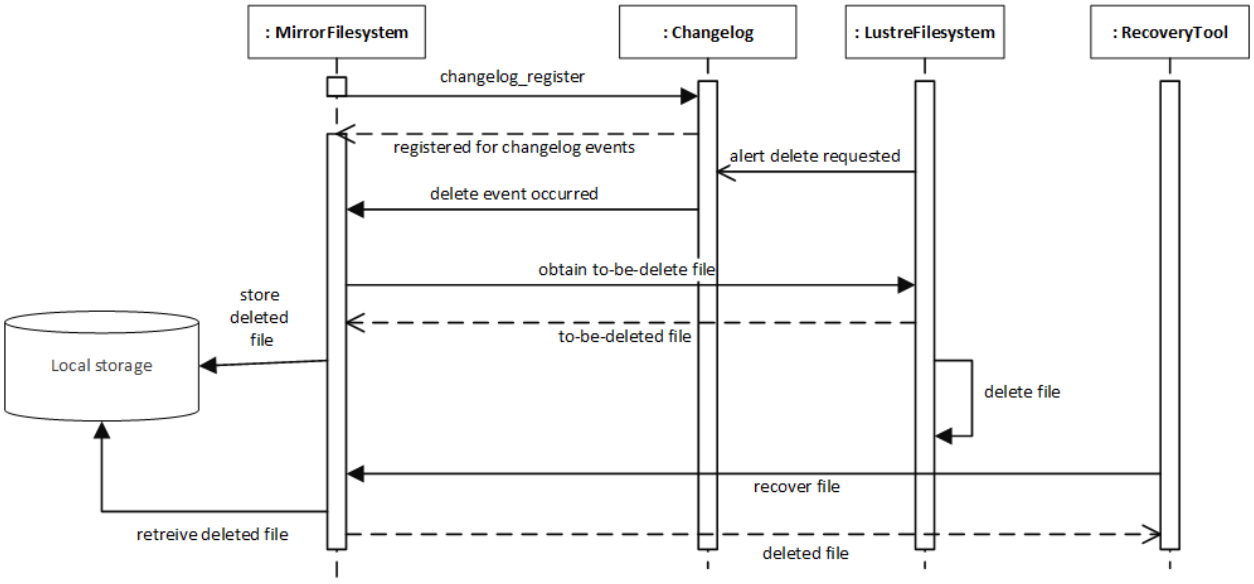

Figure 39. By storing the deleted files from the Lustre file system on a mirrored file system, a recovery tool can simply request the deleted file from the mirrored file system.

Note that this solution assumes that there exists a means of instrumenting the deletion process for the Lustre file system, where a to-be-deleted file can be retrieved between the time that the mirrored file system is alerted of the file deletion and the time that the Lustre file system actually deletes the file.

There are a few major disadvantages of this method, including

1. A secondary file system is required to monitor the Lustre file system
2. The mirror file system stores all files deleted over the life of the Lustre file system, which can require enormous volumes of storage

Currently, a means of instrumenting the deletion process on a Lustre file system has not been discovered (whether through the changelog API or otherwise), which makes this solution infeasible. Without a means of notifying an external entity that a file is *about to be deleted* and then retrieving the deleted file prior to the Lustre file system actually deleting the file, this solution is impractical.

Appendix C: Common Pitfalls

This section contains an enumeration of the common pitfalls discovered during research about the Lustre file system. Accompanying each enumerated pitfall is a brief description about the hazard and any information that may provide the reader with a means of mitigating this risk. This list is not intended to be extensive nor exhaustive, but rather, to capture the common idiosyncrasies of the Lustre file system and the Lustre documentation.

Confusion between OSSs & OSTs

Throughout much of the documentation associated with Lustre, a loose distinction in made between OSSs and OSTs. In general, the two are treated as interchangeable in their usage and there does not appear to be a disciplined effort made to ensure that the proper usage of the two terms are used in documentation and the code base.

For example, on page 6 of [13], the textual description of the *write* data flow states that "The clients communicate directly with each OST, which writes the data in parallel until done." Directly below this textual description, the *write* diagram uses the term "OSSes" when displaying the component with which the client communicates to obtain the objects for a file. The same pattern is repeated for the *read* data flow textual description and diagram.

Likewise, page 9 of [21] states that "When a client wants to read from or write to a file, it first fetches the layout EA from the MDT object for the file. The client then uses this information to perform I/O on the file, directly interacting with the OSS nodes where the objects are stored." Contrary to this description, the accompanying figure, **Figure 1.4**, illustrates the client component communicating directly with the OSTs of the file system, rather than with the OSSs (which would then forward on the request to the OSTs, acting as a proxy for the communication with the client).

To compound these discrepancies, according to [22], the Lustre source tree b16 contained the following:

```
/* Sigh - really, this is an OSS, the _server_, not the _target_ */
static int ost_setup(struct obd_device *obd, obd_count len, void *buf)
{ ... }
```

When viewing Lustre from an end-user perspective, it is not of imperative importance to make the distinction between OSSs and OSTs, but there appears to be an obvious looseness in the manner in which the distinction between storage servers and targets in made. This discrepancy becomes more important when a clear distinction must be made between contact with the OSSs or the OSTs, such as with the contacting of OSTs for the purpose of retrieving the objects that compose a file.

In general, it appears as through the use of the phrase, "a client communicates directly with an OST" means that a client communicates with the OSS responsible for the OST. This interpretation is corroborated by the definition given for the OSC provided in the Glossary of [21]. Therefore, it should be assumed that throughout Lustre documentation, when a client "communicates directly with an OST," the client is actually communicating with the OSS responsible for the OST, and the OSS is forwarding these requests on behalf of the OST with which the client wishes to communicate.

Appendix D: Outstanding & Unresolved Issues

This section contains a detailed description of all outstanding issues, of which a complete solution has not been found. Any information or possible approaches discovered during the research conducted within this document is described and referenced below. This section is intended to provide a background and context to the roadblocks discovered during research and provide the reader with a possible solution to these problem, without having to experience many of the pitfalls him or herself.

Failure to Connect OSS to MGS/MDS Node

Upon completion of the steps outlined in the **Creating & Mounting OST Block Device** section of this document, the singular OST device of the OSS failed to mount to the OSS. Upon completion of the mount command, the following error was received through the command line,

```
mount.lustre: mount /dev/sdb at /mnt/ost0 failed: Input/output error
Is the MGS running?
```

where `/dev/sdb` is the location of the OST block device on the OSS and `/mnt/ost0` is the mount point for the formatted OST block device. In order to remove the possibility of network connection errors as an issue, the `ping` command was issued from the OSS VM to the MGS/MDS VM, and vice versa; both resulted in a successful ping, with relatively low latency (less than 1 *ms*). The result of the `ping` command from the OSS VM to the MGS/MDS VM was

```
PING 192.168.44.130 (192.168.44.130) 56(84) bytes of data.
64 bytes from 192.168.44.130: icmp_seq=1 ttl=64 time=0.582 ms
64 bytes from 192.168.44.130: icmp_seq=2 ttl=64 time=0.771 ms
64 bytes from 192.168.44.130: icmp_seq=3 ttl=64 time=0.183 ms
^C
--- 192.168.44.130 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2749ms
rtt min/avg/max/mdev = 0.183/0.512/0.771/0.245 ms
```

Note that `192.168.44.130` is the static IP address of the MGS/MDS VM. Likewise, the output received by executing the `ping` command from the MGS/MDS VM, with the OSS VM as the target, was

```
PING 192.168.44.200 (192.168.44.200) 56(84) bytes of data.
64 bytes from 192.168.44.200: icmp_seq=1 ttl=64 time=0.211 ms
64 bytes from 192.168.44.200: icmp_seq=2 ttl=64 time=0.803 ms
64 bytes from 192.168.44.200: icmp_seq=3 ttl=64 time=0.304 ms
^C
--- 192.168.44.200 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2343ms
rtt min/avg/max/mdev = 0.211/0.439/0.803/0.260 ms
```

Likewise note that `192.168.44.200` is the static IP address of the OSS VM. Based on the output of the two ping commands, it is clear that there was a network connection between the two machines (using a NAT network, as established through VMWare Player). In order to check if the MGS was indeed running on the MGS/MDS VM, the mounted targets in the Lustre cluster were displayed using the following command:

```
$    sudo cat /proc/fs/lustre/mgs/MGS/live/*
```

The results of this command were:

```
fsname: lustre
flags: 0x20     gen: 7
lustre-MDT0000

Secure RPC Config Rules:

imperative_recovery_state:
    state: full
    nonir_clients: 0
    nidtbl_version: 3
    notify_duration_total: 0.000000
    notify_duation_max: 0.000000
    notify_count: 1
fsname: params
flags: 0x21     gen: 1

Secure RPC Config Rules:

imperative_recovery_state:
    state: full
    nonir_clients: 0
    nidtbl_version: 2
    notify_duration_total: 0.000000
    notify_duation_max: 0.000000
    notify_count: 0
```

These results show that the MDT (`lustre-MDT0000`) mounted successfully. A subsequent command was issued to ensure that the MGS was started[48]:

```
$    sudo cat /proc/fs/lustre/devices
```

The results of this command were as follows:

```
  0 UP osd-ldiskfs lustre-MDT0000-osd lustre-MDT0000-osd_UUID 8
  1 UP mgs MGS MGS 5
  2 UP mgc MGC192.168.44.130@tcp 87b95ad5-7792-e71f-2b63-32b1981ee0ce 5
  3 UP mds MDS MDS_uuid 3
  4 UP lod lustre-MDT0000-mdtlov lustre-MDT0000-mdtlov_UUID 4
  5 UP mdt lustre-MDT0000 lustre-MDT0000_UUID 5
  6 UP mdd lustre-MDD0000 lustre-MDD0000_UUID 4
  7 UP qmt lustre-QMT0000 lustre-QMT0000_UUID 4
  8 UP lwp lustre-MDT0000-lwp-MDT0000 lustre-MDT0000-lwp-MDT0000_UUID 5
```

[48] This command was found in the walkthrough presented in [44].

According to this output, the MGS is in fact running on the MGS/MDS VM. In order to test that the MGS/MDS VM could communicate through Lustre with the OSS VM, and vice versa, the `lctl ping` command was executed from the OSS VM, with the MGS/MDS VM as the target:

```
$    sudo lctl ping 192.168.44.130
```

Note that `192.168.44.130` is the static IP address of the MGS/MDS VM. This command resulted in the following output:

```
failed to ping 192.168.44.130@tcp: Input/output error
```

Pinging the OSS from the MGS/MDS node using the `lctl ping` command resulted in similar output:

```
$    sudo lctl ping 192.168.44.200

failed to ping 192.168.44.200@tcp: Input/output error
```

In order to discover any issues in the network connection between the OSS and MGS/MDS VMs, the `/var/log/messages` file on the OSS VM was scanned. The following pertinent output relating to the issue was found within this log file:

```
Mar 25 20:08:11 oss0 kernel: LDISKFS-fs (sdb): mounted filesystem with ordered data mode.
quota=on. Opts:
Mar 25 20:08:16 oss0 kernel: Lustre: 2667:0:(client.c:1926:ptlrpc_expire_one_request()) @@@
Request sent has timed out for slow reply: [sent 1427339291/real 1427339291]
req@ffff880021e34c00 x1496173068157008/t0(0) o250->MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25
lens 400/544 e 0 to 1 dl 1427339296 ref 1 fl Rpc:XN/0/ffffffff rc 0/-1
Mar 25 20:08:21 oss0 kernel: LustreError: 6084:0:(client.c:1083:ptlrpc_import_delay_req()) @@@
send limit expired   req@ffff880021e34800 x1496173068157012/t0(0) o253-
>MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25 lens 4768/4768 e 0 to 0 dl 0 ref 2 fl
Rpc:W/0/ffffffff rc 0/-1
Mar 25 20:08:21 oss0 kernel: LustreError:
6084:0:(obd_mount_server.c:1165:server_register_target()) lustre-OST0000: error registering with
the MGS: rc = -5 (not fatal)
Mar 25 20:08:26 oss0 kernel: LustreError: 6084:0:(client.c:1083:ptlrpc_import_delay_req()) @@@
send limit expired   req@ffff880021e34800 x1496173068157016/t0(0) o101-
>MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25 lens 328/344 e 0 to 0 dl 0 ref 2 fl
Rpc:W/0/ffffffff rc 0/-1
Mar 25 20:08:31 oss0 kernel: LustreError: 6084:0:(client.c:1083:ptlrpc_import_delay_req()) @@@
send limit expired   req@ffff880021e34800 x1496173068157020/t0(0) o101-
>MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25 lens 328/344 e 0 to 0 dl 0 ref 2 fl
Rpc:W/0/ffffffff rc 0/-1
Mar 25 20:08:31 oss0 kernel: LustreError: 13a-8: Failed to get MGS log lustre-OST0000 and no
local copy.
Mar 25 20:08:31 oss0 kernel: LustreError: 15c-8: MGC192.168.44.130@tcp: The configuration from
log 'lustre-OST0000' failed (-2). This may be the result of communication errors between this
node and the MGS, a bad configuration, or other errors. See the syslog for more information.
Mar 25 20:08:31 oss0 kernel: LustreError: 6084:0:(obd_mount_server.c:1297:server_start_targets())
failed to start server lustre-OST0000: -2
Mar 25 20:08:31 oss0 kernel: LustreError: 6084:0:(obd_mount_server.c:1769:server_fill_super())
Unable to start targets: -2
Mar 25 20:08:31 oss0 kernel: LustreError: 6084:0:(obd_mount_server.c:1496:server_put_super()) no
obd lustre-OST0000
Mar 25 20:08:31 oss0 kernel: Lustre: server umount lustre-OST0000 complete
Mar 25 20:08:31 oss0 kernel: LustreError: 6084:0:(obd_mount.c:1342:lustre_fill_super()) Unable to
mount  (-2)
```

```
Mar 25 20:08:51 oss0 kernel: LDISKFS-fs (sdb): mounted filesystem with ordered data mode.
quota=on. Opts:
Mar 25 20:08:51 oss0 kernel: Lustre: 2667:0:(client.c:1926:ptlrpc_expire_one_request()) @@@
Request sent has failed due to network error: [sent 1427339331/real 1427339331]
req@ffff88003b82ec00 x1496173068157024/t0(0) o250->MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25
lens 400/544 e 0 to 1 dl 1427339336 ref 1 fl Rpc:XN/0/ffffffff rc 0/-1
Mar 25 20:09:01 oss0 kernel: LustreError: 6124:0:(client.c:1083:ptlrpc_import_delay_req()) @@@
send limit expired   req@ffff88003b82ec00 x1496173068157028/t0(0) o253-
>MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25 lens 4768/4768 e 0 to 0 dl 0 ref 2 fl
Rpc:W/0/ffffffff rc 0/-1
Mar 25 20:09:01 oss0 kernel: LustreError:
6124:0:(obd_mount_server.c:1165:server_register_target()) lustre-OST0000: error registering with
the MGS: rc = -5 (not fatal)
Mar 25 20:09:06 oss0 kernel: LustreError: 6124:0:(client.c:1083:ptlrpc_import_delay_req()) @@@
send limit expired   req@ffff88003b82ec00 x1496173068157032/t0(0) o101-
>MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25 lens 328/344 e 0 to 0 dl 0 ref 2 fl
Rpc:W/0/ffffffff rc 0/-1
Mar 25 20:09:11 oss0 kernel: LustreError: 13a-8: Failed to get MGS log lustre-OST0000 and no
local copy.
Mar 25 20:09:11 oss0 kernel: LustreError: 15c-8: MGC192.168.44.130@tcp: The configuration from
log 'lustre-OST0000' failed (-2). This may be the result of communication errors between this
node and the MGS, a bad configuration, or other errors. See the syslog for more information.
Mar 25 20:09:11 oss0 kernel: LustreError: 6124:0:(obd_mount_server.c:1297:server_start_targets())
failed to start server lustre-OST0000: -2
Mar 25 20:09:11 oss0 kernel: LustreError: 6124:0:(obd_mount_server.c:1769:server_fill_super())
Unable to start targets: -2
Mar 25 20:09:11 oss0 kernel: LustreError: 6124:0:(obd_mount_server.c:1496:server_put_super()) no
obd lustre-OST0000
Mar 25 20:09:12 oss0 kernel: Lustre: server umount lustre-OST0000 complete
Mar 25 20:09:12 oss0 kernel: LustreError: 6124:0:(obd_mount.c:1342:lustre_fill_super()) Unable to
mount  (-2)
```

Of particular importance is the line

```
Mar 25 20:08:16 oss0 kernel: Lustre:
2667:0:(client.c:1926:ptlrpc_expire_one_request()) @@@ Request sent has timed out for
slow reply: [sent 1427339291/real 1427339291]  req@ffff880021e34c00
x1496173068157008/t0(0) o250->MGC192.168.44.130@tcp@192.168.44.130@tcp:26/25 lens
400/544 e 0 to 1 dl 1427339296 ref 1 fl Rpc:XN/0/ffffffff rc 0/-1
```

This line states that a request sent from the OSS to the MGS/MDS node timed out. This is likely caused by the inability of the OSS to connect to the MGS/MDS, within its Lustre network, rather than the IP network connection. In an attempt to remedy this time out error (to ensure that it was in fact not an issue of a premature time out, which given enough time, would complete), the timeout for the OST was set to `100` using the `--param="sys.timeout=100"` flag of the `mkfs.lustre` command.[49] Again, an attempt was made to mount the OST block device to the OSS, but this mount attempt failed as well. Due to the less-than-1-millisecond latency between the OSS VM and the MGS/MDS VM, it is not likely that this timeout was caused by any delay in the connections between the two nodes.

Upon further investigation, others had were to have experienced the same problems. In particular, [45] and [46] suggested possible solutions to a problem description matching the issue documented in this section; both of these solutions were attempted, but to no avail. Likewise, the steps suggested in [44] were also tried, but likewise, did not result in a solution to this issue.

[49] For more information on the timeout settings for a Lustre file system, including the timeout configurations possible when creating the Lustre file system, see the **Lustre Operations** and **LustreProc** chapters of [21] (chapter 13 and 31, respectively).

Approaching the problem from a different perspective, both [47] and [48] suggest that opening port `988` in the IPTables would solve similar issues with a simple Lustre cluster. This approach was attempted by opening port `988` in the IPTables, using the following command:

```
$    sudo iptables -A INPUT -p tcp --dport 988 -j ACCEPT
```

This command was executed on both the OSS and MGS/MDS VMs. To ensure that port `988` was in fact opened after the execution of the above command, the following command was executed[50]:

```
$    sudo iptables -L
```

This resulted in the following output:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere            state RELATED,ESTABLISHED
ACCEPT     icmp --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
ACCEPT     tcp  --  anywhere             anywhere            state NEW tcp dpt:ssh
REJECT     all  --  anywhere             anywhere            reject-with icmp-host-prohibited
ACCEPT     tcp  --  anywhere             anywhere            tcp dpt:988

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
REJECT     all  --  anywhere             anywhere            reject-with icmp-host-prohibited

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Based on this output, it can be seen that the Transmission Control Protocol (TCP) port `988` is open from any source to any destination. In order to ensure that no other network-based services conflicted with this configuration, or that no other network-based services required a restart prior to the changes to the IP taking effect, the network service was restarted. After restarting the network service, an attempt was again made to mount the OST block device to the OSS, but again, this attempt failed.

It is worth noting that when an attempt was made to mount the OST to the OSS, after the first failure (without reformatting the OST block device using the `mkfs.lustre` command), the resulting error changed to

```
mount.lustre: mount /dev/sdb at /mnt/ost0 failed: No such file or directory
Is the MGS specification correct?
Is the filesystem name correct?
If upgrading, is the copied client log valid? (see upgrade docs)
```

Even with this change to the reported error, the result remained the same: The OST block device was unable to mount. Also, the `/var/log/messages` file did not reveal any new information (apart from what was seen when the mounting process with the original error message).

[50] For more information on altering the IPTables, see [49].

In order to confirm that the Lustre software was properly installed on the VM, the test scripts for Lustre were executed using the following commands:

```
$    cd /usr/lib64/lustre/tests/
$    sudo ./llmount.sh
```

After executing these commands, the following output was received:

```
Stopping clients: mgs-mds0 /mnt/lustre (opts:)
Stopping client mgs-mds0 /mnt/lustre opts:
Stopping clients: mgs-mds0 /mnt/lustre2 (opts:)
Loading modules from /usr/lib64/lustre/tests/..
detected 1 online CPUs by sysfs
libcfs will create CPU partition based on online CPUs
debug=vfstrace rpctrace dlmtrace neterror ha config                    ioctl super
lfsck
subsystem_debug=all -lnet -lnd -pinger
Formatting mgs, mds, osts
Format mds1: /tmp/lustre-mdt1
Format ost1: /tmp/lustre-ost1
Format ost2: /tmp/lustre-ost2
Checking servers environments
Checking clients mgs-mds0 environments
Loading modules from /usr/lib64/lustre/tests/..
detected 1 online CPUs by sysfs
libcfs will create CPU partition based on online CPUs
debug=vfstrace rpctrace dlmtrace neterror ha config                    ioctl super
lfsck
subsystem_debug=all -lnet -lnd -pinger
Setup mgs, mdt, osts
Starting mds1:   -o loop /tmp/lustre-mdt1 /mnt/mds1
Started lustre-MDT0000
Starting ost1:   -o loop /tmp/lustre-ost1 /mnt/ost1
Started lustre-OST0000
Starting ost2:   -o loop /tmp/lustre-ost2 /mnt/ost2
Started lustre-OST0001
Starting client: mgs-mds0: -o user_xattr,flock mgs-mds0@tcp:/lustre /mnt/lustre
Using TIMEOUT=20
disable quota as required
```

Note that the first three lines are likely the result of the test script being previously executed and the local Lustre block devices being mounted at the time the test script was executed. Note that the local Lustre cluster created through this test script is usable on the server VM, and can be accessed by entering the `/mnt/lustre/` directory. In order to unmount the MDT, MGT, and OSTs created through this test script, execute the following command:

```
$    sudo ./llmountcleanup.sh
```

Upon execution of this command, the following output should be received:

```
Stopping clients: mgs-mds0 /mnt/lustre (opts:-f)
Stopping client mgs-mds0 /mnt/lustre opts:-f
Stopping clients: mgs-mds0 /mnt/lustre2 (opts:-f)
Stopping /mnt/mds1 (opts:-f) on mgs-mds0
Stopping /mnt/ost1 (opts:-f) on mgs-mds0
Stopping /mnt/ost2 (opts:-f) on mgs-mds0
modules unloaded.
```

Note that the local Lustre file system created by the test script above can be used to test Lustre features (and was used to gather information about the Lustre file system in the research contained in this document), but does not represent the full-networked environment of the Lustre file system. Therefore, any features or capabilities that require network interaction or relies directly on the communication between nodes over the network should establish a fully networked Lustre cluster. At the time of writing, a solution to this issue has not be devised.

Index