**Albano GRP 1.0.0-2015**

(Revision 2015-03-29)

**Research Findings**

**for Lustre Distributed File System**

**Graduate Research Project**

Sponsor

**The Department of Electrical, Computer, Software & Systems Engineering at**

**Embry-Riddle Aeronautical University**

Released March 29, 2015

**Justin Albano**

**Abstract:** Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean sit amet dolor turpis. Nam posuere lorem nibh, nec posuere lorem ultrices et. Proin est diam, volutpat nec leo ac, congue ultricies odio. Fusce turpis sapien, porta sed nunc eget, interdum dictum odio. Vestibulum id est id lacus feugiat dictum. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec viverra augue orci, ac posuere elit interdum id. Vestibulum justo orci, suscipit non elit sed, placerat consectetur mi. Vivamus et odio ullamcorper, semper lorem nec, auctor urna. Proin risus nisi, ullamcorper a varius eget, elementum vel lorem. Maecenas justo ligula, dignissim et diam et, rhoncus lobortis erat. Nam molestie lorem ac mauris blandit eleifend. Nunc gravida sodales nisl, sed eleifend ante dapibus vitae. Duis convallis quam sit amet rutrum placerat. Aenean blandit in elit eu luctus. Maecenas nec tortor vitae leo rhoncus placerat.

Department of Electrical, Computer, Software & Systems Engineering

Senior Design 2013-2014

Embry-Riddle Aeronautical University

600 S. Clyde Morris Blvd.

Daytona Beach, FL 32114

This document is property of the Justin Albano, 2015, at Embry-Riddle Aeronautical University and may not be reproduced in any form or altered without the permission of the aforementioned author.

# Revision History

|  |  |  |
| --- | --- | --- |
| **Date** | **Reason for Change** | **Version** |
| Mar. 19, 2015 | Initial draft of document | 1.0.0 |

**Contents**

[Revision History ii](#_Toc415394653)

[1. Introduction 1](#_Toc415394654)

[2. Background 2](#_Toc415394655)

[2.1 An Overview of the Lustre File System 2](#_Toc415394656)

[2.2 Object Storage & Striping 4](#_Toc415394657)

[2.3 Client Interface to a Lustre File System 7](#_Toc415394658)

[2.4 Linux VFS & Lustre 9](#_Toc415394659)

[2.5 End-to-End Lustre Operation 19](#_Toc415394660)

[3. Installation Procedures 21](#_Toc415394661)

[3.1 Lustre Server Installation & Configuration Procedures 21](#_Toc415394662)

[3.2 Configuring Server Nodes 27](#_Toc415394663)

[4. Problem Statement 33](#_Toc415394664)

[5. Solution 34](#_Toc415394665)

[6. Outstanding & Unresolved Issues 35](#_Toc415394666)

[6.1 Failure to Connect OSS to MGS/MDS Node 35](#_Toc415394667)

[Glossary 40](#_Toc415394668)

[Acronyms & Abbreviations 41](#_Toc415394669)

[References 42](#_Toc415394670)

[Appendix A: Original Research Proposal 44](#_Toc415394671)

[Objective 44](#_Toc415394672)

[Background 44](#_Toc415394673)

[Problem Statement 44](#_Toc415394674)

[References 44](#_Toc415394675)

[Index 45](#_Toc415394676)

# Introduction

# Background

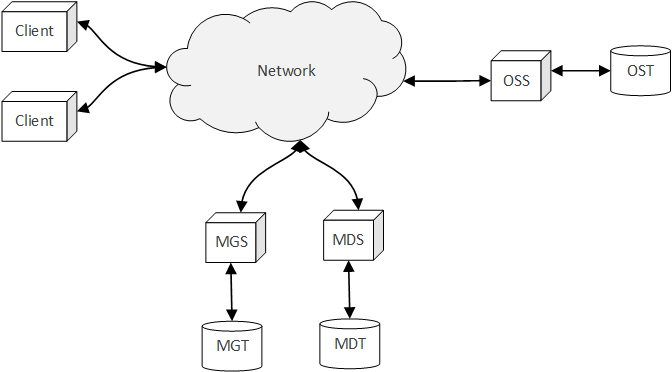
## An Overview of the Lustre File System

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 within 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)

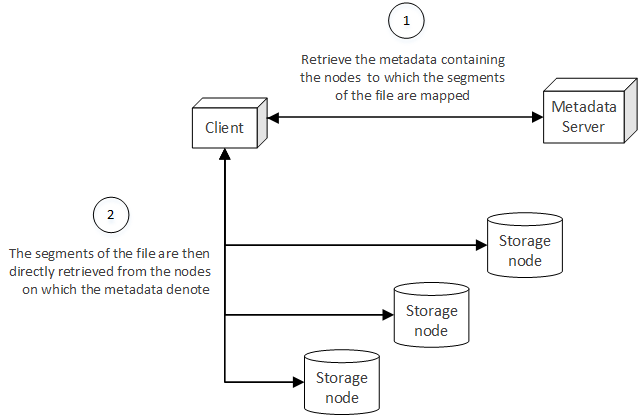
A simple, example Lustre topology is illustrated below in **Figure 1**.



**Figure 1.** 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 of storage [1]. Thus, the OSS, paired with its multiple OSTs, make up the backbone of the storage capability of a Lustre file system: During operation, the data that makes up a file resides 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, where 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 2**.



**Figure 2.** 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 the 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. This MDS stores the location of the segments that make up a file in the Lustre file system; in the case of Lustre, these locations 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 [1], [2].

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 [3]. 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[[1]](#footnote-1).

An MDS 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 [1]. 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 1** is a simple example of the server and client portions of Lustre, enterprise Lustre file systems contain many more nodes than simply 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 [1]. Therefore, the metadata portion of the Lustre file system can be established to serve a fail-over capacity, 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 [1]:

* 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) [1]:

* 1 primary MDT with 1 MDT backup
* 500 OSSs with a total of 4,000 OSTs[[2]](#footnote-2)

Likewise, the connections between the components in a Lustre cluster are more complex than those presented in **Figure 1**. 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 input/output (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 that 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 [1].

With an understanding of the topology and components of a Lustre cluster, it is important to the 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.

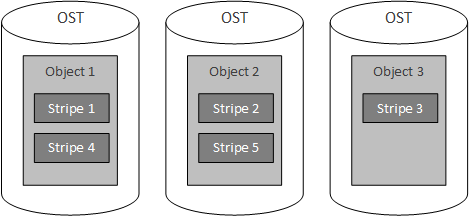
## Object Storage & Striping

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 requires to retrieve a file from the file system become 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 system), the client and 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 single machine.

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 object.

Using this approach, a Lustre file system matches a RAID 0 arrangement, where data is sequentially distributed, or striped, across a collection of disks [5]. 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 in a physical disk organization), 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[[3]](#footnote-3). 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 [1].

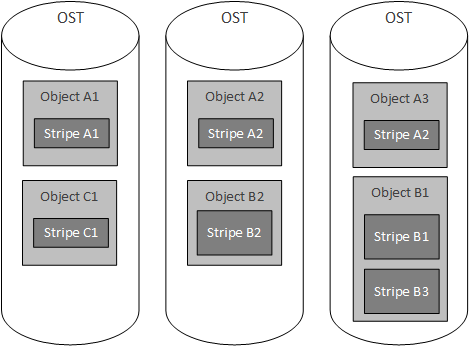
In a Lustre file system, striping occurs in a round-robin fashion, where strips are written according to a defined stripe size. When a file is stored, bits are written to an object on an OST until the stripe size is reached, whereupon bits are written to an object on the next OST. This process continues 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 stripe within the object of the first OST [1]. This process is illustrated below in **Figure 3**.



**Figure 3.** Parts of a file exist on an OST as an object, which contains 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)[[4]](#footnote-4). In **Figure 3**, 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].

Note that multiple objects, storing data for different files, exist on a single OST, each with possibly varying stripe counts and stripe sizes. **Figure 4** illustrates a more complex storage configuration, where three files are stored with 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.



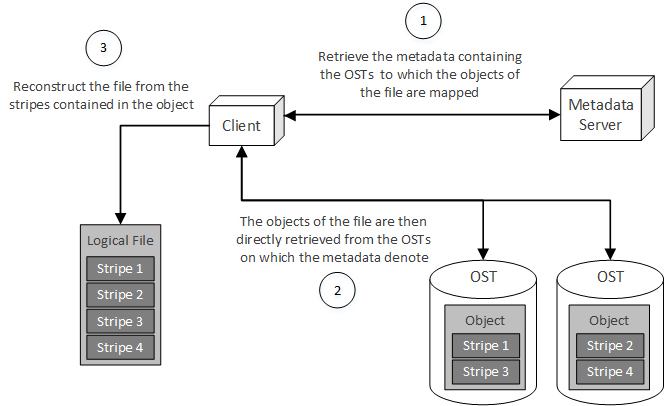
**Figure 4.** Various objects, representing parts of files, concurrently exist on a single OST, each with possibly varying stripe count and strip size.

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. Upon retrieving the objects, object 1 is read for bits, where is equal to the stripe size. Once bits are read from the first object (and it known that there are more bits in the file, such as with knowledge of the file size), then bits are read from the second object. If there are more bits to read, bits are read from first object, starting at bit , and this process continues until the entire file is read.

Note that the last read from an object may not be equal to bits. Instead, the last number of bits, , to read from the object is bound by and is equal to

where is equal to size of the file in bits, is the number of stripes read so far, is the stripe size, and is the integer floor of (truncation of the decimal value for a given number ). The resulting process of obtaining the object locations from the MDS and reconstructing a file is illustrated in **Figure 5**. This figure is an expansion of **Figure 2**, and includes the details described in this section.

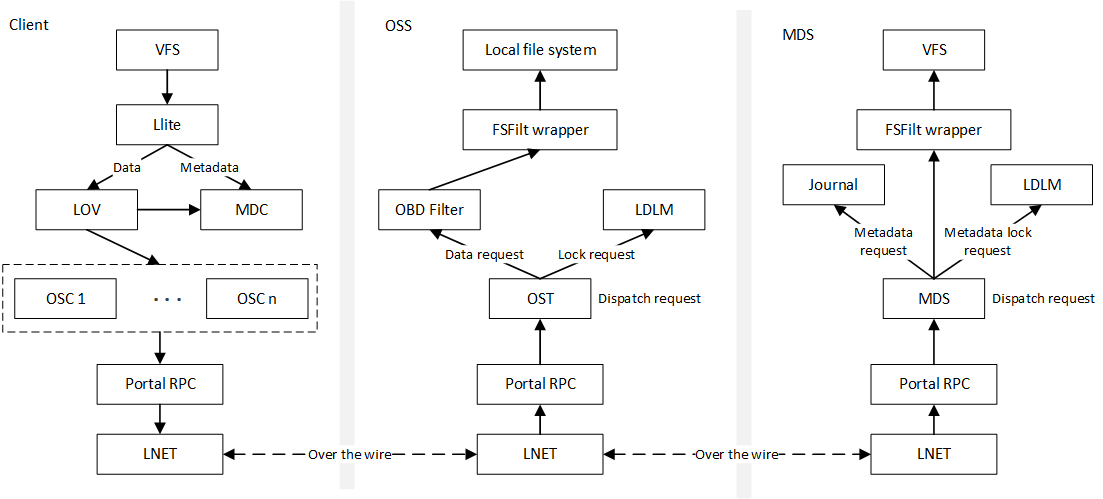
With a foundational understanding of how a file is stored and reconstructed in a Lustre file system, it is important to understand how clients present a seamless interface to the end-user. This discussion includes the layered architecture of a Lustre file system, a description of the Linux Virtual File 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 5.** 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.

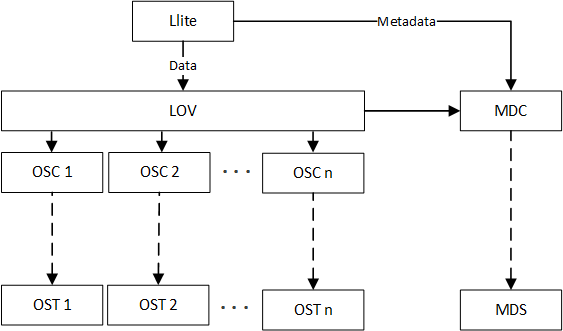
## Client Interface to a Lustre File System

Based on an understanding of the scheme presented in **Figure 5**, 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 6**.



**Figure 6.** Lustre uses a layered architecture to communicate between client, OSS, and MDS [10].

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 is to be accessed (as in the case of retrieving an object from an OST) or the Metadata Client (MDC) when metadata is 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 also 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 an OST [10]. A logical illustration of the MDC and OSCs of a client are depicted in **Figure 7**.



**Figure 7.** 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 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 [10].

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 Block Device (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[[5]](#footnote-5) 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[[6]](#footnote-6).

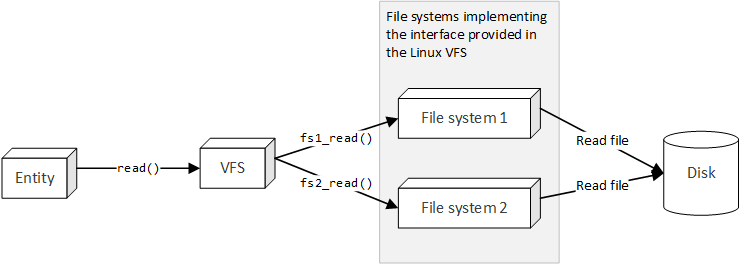
In the case of lock requests, the request is forwarded to the Lustre Distributed Lock Manager (LDLM). The LDLM is responsible for maintaining global consistency for file locking among all of the nodes in a Lustre File System. This module of the Lustre file system is explored in greater detail in **Chapter 4** of [10].

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 [10].

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.

## Linux VFS & Lustre

The Linux VFS[[7]](#footnote-7) 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 8**.

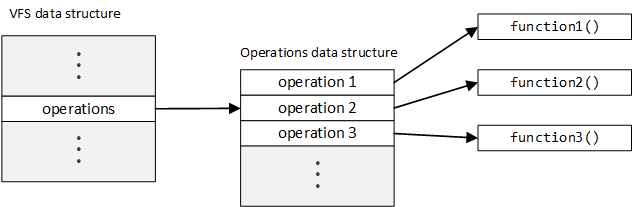


**Figure 8.** The Linux VFS provides a level of indirection, ensuring that an entity can interact with a mounted file system without knowing the internal details of the file system.

This scheme is very similar to the use of software interfaces in Object-Oriented Programming (OOP) to abstract the implementation details of a class, and as will follow 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 ([11])

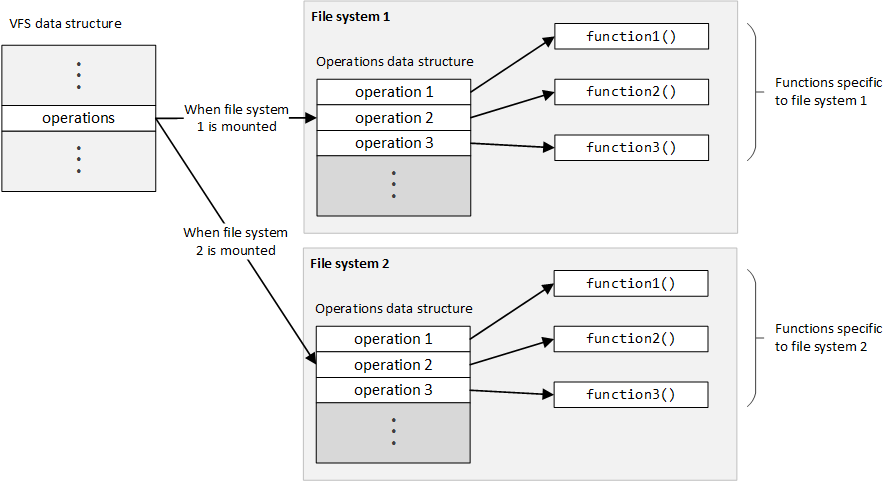
1. Superblock: the root of the tree, representing a mounted file system
2. Inode: represents a single file in the file system
3. Dentry: represents a single directory entry, or a component in a path
4. File: represents an open file associated with a process in the Linux kernel[[8]](#footnote-8)

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 9**.



**Figure 9.** 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 10**.



**Figure 10.** 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,

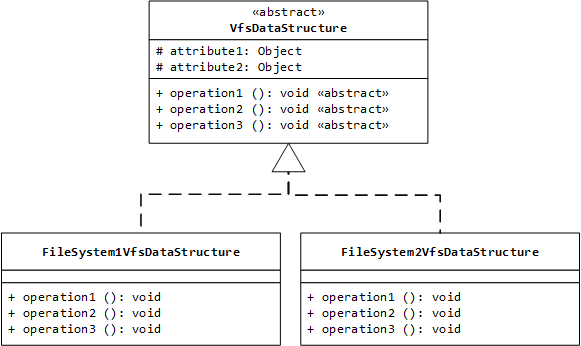
vfs\_data\_struct->operations->operation1();

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 10**). 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) [12]. A class diagram representing the OOP equivalent is illustrated below in **Figure 11**.



**Figure 11.** 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:

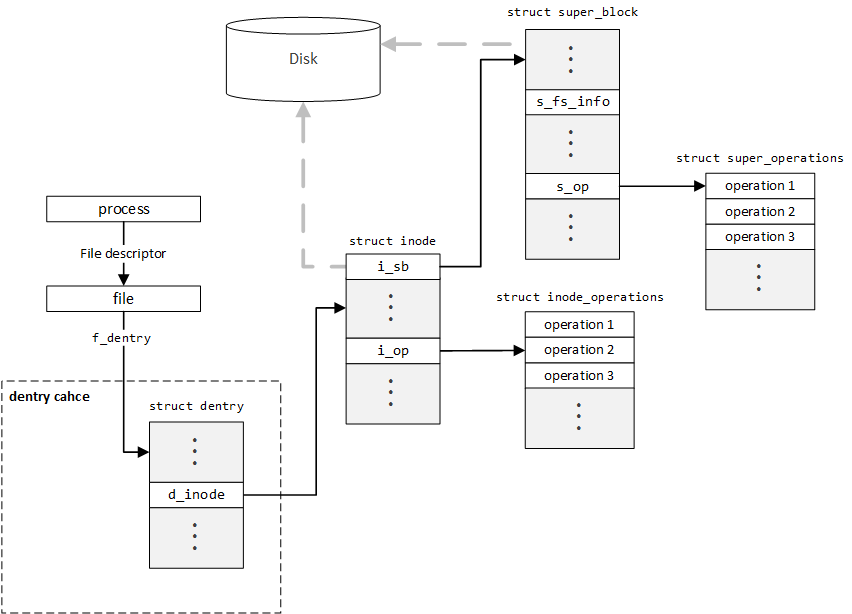
// Obtain a VfsDataStructure object from the factory

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 12**.

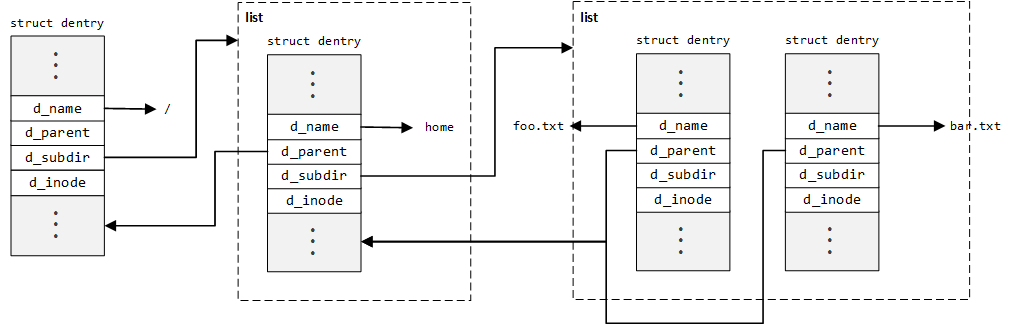


**Figure 12.** The VFS data structures point to one another, creating a tree structure that represents the file system in memory [10], [13], [16].

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, if 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 represent / 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 13**.

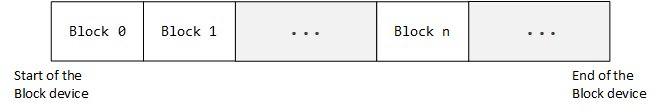


**Figure 13.** The chaining of dentry structures to represent a path creates a double-linked tree, where each element has a reference to a list of sub-elements as well as a reference to its parent element.

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 13**. 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[[9]](#footnote-9) [17]. 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[[10]](#footnote-10). 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.[[11]](#footnote-11)

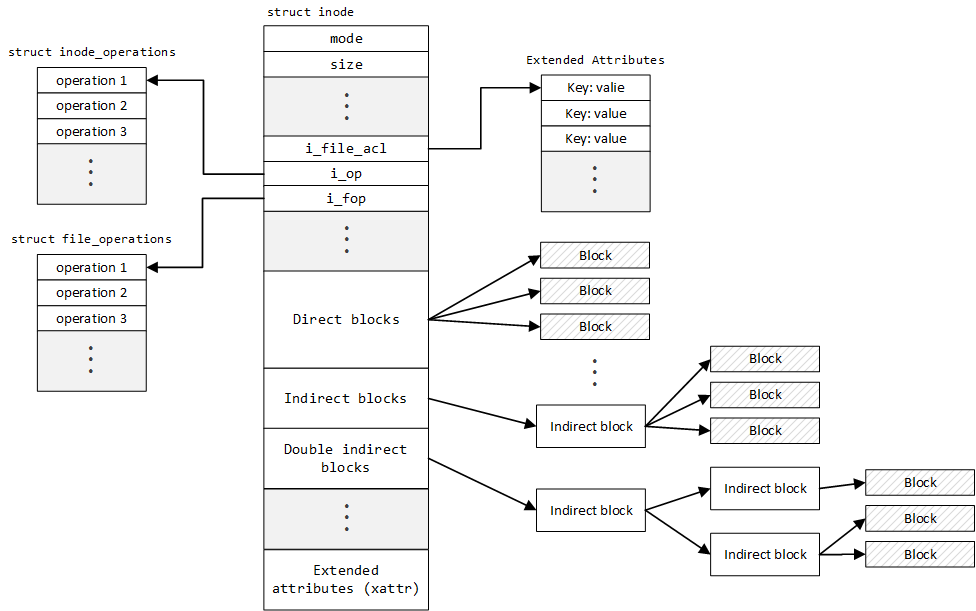
With a foundational understanding in the dentry structure established, it is important to explore the inode structure in greater detail. An inode is created each time a file is accessed and stores information about a file, including the permissions of the file, the access times, and, in the case of a 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 [15]. This concept is illustrated below in **Figure 14**.



**Figure 14.** 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.

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 15[[12]](#footnote-12)**.



**Figure 15.** 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 [10], [11], [15], [18].

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[[13]](#footnote-13)). 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 much the same scheme as double indirect blocks, but including one more level of indirection [15][[14]](#footnote-14).

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 node, such security data or other non-essential inode data [15]. 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 [15].

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 create 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 [11]. 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 [11], [13], [15], [18], [19], and [20].

### 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 below in **Figure 16**.

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)[[15]](#footnote-15)

function with the following logic:

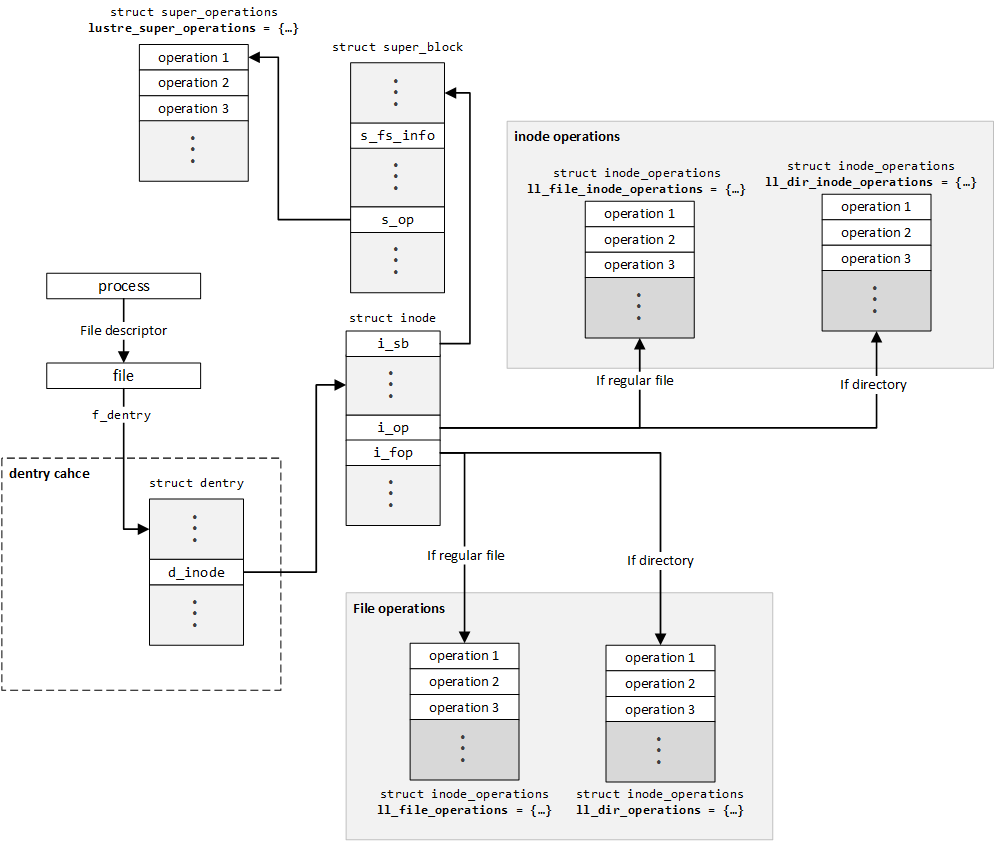
sb->s\_op = &lustre\_super\_operations[[16]](#footnote-16)

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)[[17]](#footnote-17)

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

lustre\_register\_client\_fill\_super[[18]](#footnote-18)

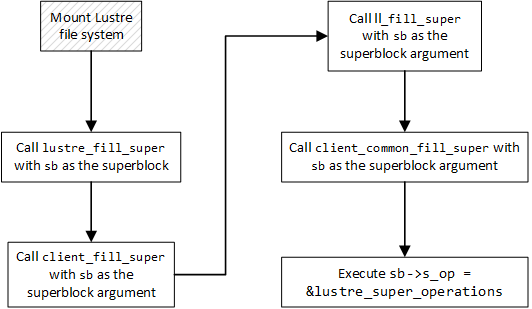


**Figure 16.** By replacing overriding the operations structures in the Linux VFS, Llite is able to override the functionality of the VFS and provide a coherent file system to the end-user [10].

This function simply sets the client\_fill\_super[[19]](#footnote-19) 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)[[20]](#footnote-20)

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 17**.



**Figure 17.** 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 17** 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()[[21]](#footnote-21) function, which is responsible for initializing inodes [10]. The logic used to initialize these operations structures in the inode is[[22]](#footnote-22)

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 (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.

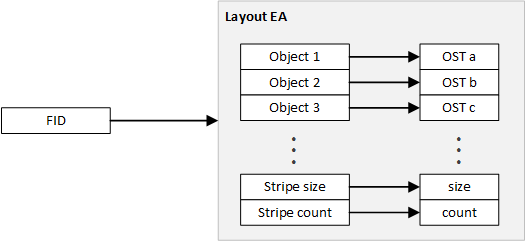
While the specifics of how the correct operation structures are set in the inode and superblock differ, since the superblock and inode are initialized at differing periods in the execution of the file system logic, a simple rule can be used to sum up who is responsible for setting the correct operations structures: The party or function that is creating the new superblock or inode is responsible for setting its correct operations structures [10]. 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.

### 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.

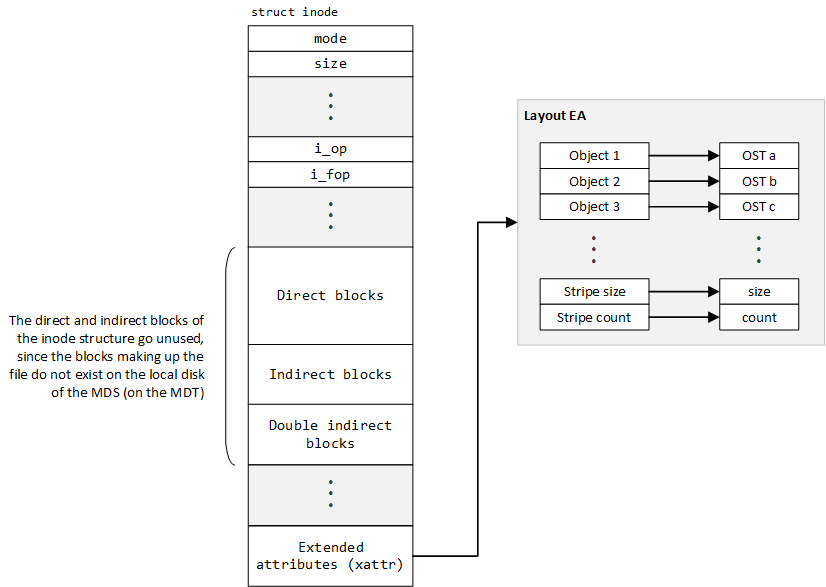
By shadow file system, we mean to say that the MDS maintains the structure for the file system, just as would a local file system (for example, maintaining an inode for each of the files in the file system, a superblock for the entirety of the file system), but instead of maintaining pointers to the 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[[23]](#footnote-23)), 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 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 [1]. 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[[24]](#footnote-24) [1], [10], as illustrated below in **Figure 18**.



**Figure 18.** The mapping of objects to their respective OST is maintained in the layout EA object stored on the MDT, as referenced by an FID.

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 19**.

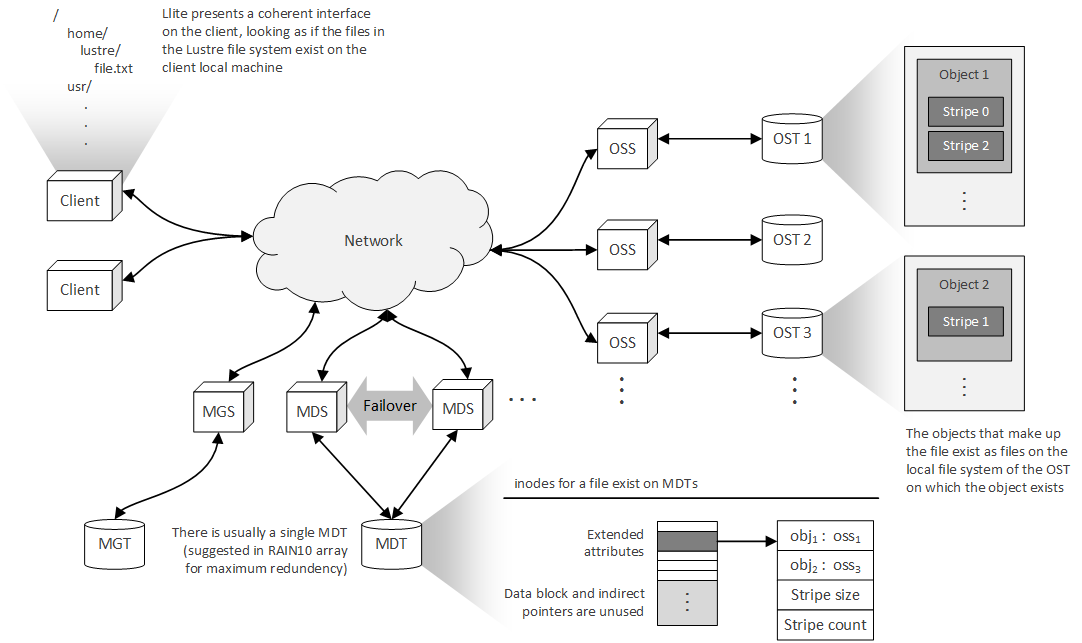


**Figure 19.** The layout EA can be thought of as a logical part of the inode structure, where the layout EA replaces the need for the direct and indirect block references of the inode.

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 a file system local 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. 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: While 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.

## 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 20** 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 20.** 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, an 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 width and the stripe size. 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 width and stripe size to reconstruct the file. With the completion of the reconstruction of this object, the file now exists on the local machine of the client and can be read by the user[[25]](#footnote-25).

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.

# 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 [1] 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 [1].

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 section 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 be successfully connect to one another. Therefore, the sections pertaining to this shortfall contain detailed description of the errors encountered and information relating to possible solutions and the solution approach taken thus far to mitigate or solve any of these issues.

## Lustre Server Installation & Configuration Procedures

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).

### Creating Virtual Machine Image

The VM used to run the Lustre software was created and executing using VMWare Player 7[[26]](#footnote-26) using the auto-installer for CentOS 6.6[[27]](#footnote-27) 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
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. 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.

### 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

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 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. 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:

|  |  |
| --- | --- |
| # | 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 [22] and [23].

Upon executing this command, the installation process with 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 [24]. 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.

### 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 Internet Protocol (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).[[28]](#footnote-28)

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"  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 [26]).

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.

### 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
2. Select the **Power** option
3. 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[[29]](#footnote-29)
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.

## Configuring Server Nodes

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.

### 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, will suffice)[[30]](#footnote-30)
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 must be 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  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 [1][[31]](#footnote-31):

|  |
| --- |
| 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  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.

### 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, they will have the same static IP configuration. 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 |

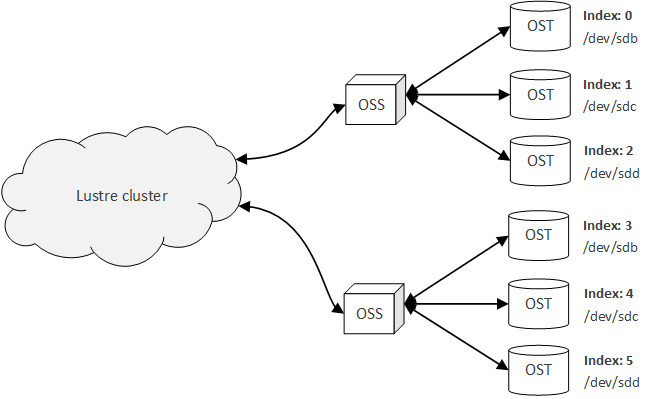
Once the OST is mounted, the creation and configuration of the server-side portion of the Lustre cluster is completed. The resulting Lustre cluster contains a single MGS/MDS node, with an accompanying MGT/MDT block device, and a single OSS, with an accompanying OST block device. Note that this is essentially 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 descried about.

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 20**.



**Figure 21.** 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 f 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 document in the **Failure to Connect OSS to MGS/MDS Node** section of this document.

# Problem Statement

# Solution

# 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 his 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 a possible solution, 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 ). The results of the ping command from the OSS VM to the MGS/MDS VM were

|  |
| --- |
| 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[[32]](#footnote-32):

|  |  |
| --- | --- |
| $ | 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 |

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 has timed out. This is likely caused by the inability of the OSS to connect to the MGS/MDS, with respect to 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 to the mkfs.lustre command.[[33]](#footnote-33) 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 been found to experience the same problems. In particular, [29] and [30] 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 [28] were also tried, but likewise, did not result in a solution to this issue.

Approaching the problem from a different perspective, both [31] and [32] 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[[34]](#footnote-34):

|  |  |
| --- | --- |
| $ | 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). At the time of writing, this issue still remains unresolved.

# Glossary

|  |  |  |
| --- | --- | --- |
| **Entry** | **Definition** | **Aliases** |
|  |  |  |

# Acronyms & Abbreviations

|  |  |
| --- | --- |
| **Entry** | **Expanded Phrase** |
| API | Application Programming Interface |
| CRUSH | Controlled Replication Under Scalable Hashing |
| EA | Extended Attribute (see also xattr) |
| Ext4 | Fourth Extended File System |
| FID | File Identifier (ID) |
| FSFilt | File System Filter |
| GPG | GNU Privacy Guard |
| GUI | Graphical User Interface |
| I/O | Input/Output |
| IP | Internet Protocol |
| ISO | International Organization for Standardization |
| Ldiskfs | Luster Disk File System |
| Llite | Lustre Lite |
| LNET | Lustre Network (or Lustre Networking) |
| LOV | Logical Object Volume |
| MDC | Metadata Client |
| MDS | Metadata Server |
| MDT | Metadata Target |
| MDT | Management Target |
| MGS | Management Server |
| OBD | Object Block Device |
| OID | Object Identifier (ID) |
| OOP | Object-Oriented Programming |
| OSS | Object Storage Server |
| OST | Object Storage Target |
| POSIX | Portable Operating System Interface |
| RAID | Redundant Array of Independent Disks (or Redundant Array of Inexpensive Disks) |
| RPC | Remote Procedure Call |
| RPM | RedHat Package Manager |
| SELinux | Security-Enhanced Linux |
| symlink | Symbolic Link |
| TCP | Transmission Control Protocol |
| VFS | [Linux] Virtual File System |
| VM | Virtual Machine |
| xattr | Extended Attributes |
| ZFS | Z File System |

# References

1. "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>.
2. 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>.
3. 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>.
4. 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>.
5. Stallings, William. *Computer Organization & Architecture: Designing for Performance* *(9th Edition)*. Upper Saddle River, NJ: Prentice Hall, 2013. Print.
6. "Project 1: Layout Enhancement Design." Scope of Work: n. pag. OpenSFS: The Lustre File System Community. OpenFSF. Web. 26 Mar. 2015.
7. "Layout Enhancement Scope Statement." (n.d.): n. pag. OpenSFS: The Lustre File System Community. OpenFSF, 10 Oct. 2013. Web. 26 Mar. 2015.
8. “Layout Enhancement Solution Architecture.” OpenSFS: The Lustre File System Community. Open Scalable File Systems, Inc., 20 Dec. 2013. Web. 26 Mar. 2015.
9. 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.
10. 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 Labratory, Apr. 2009. Web. 27 Mar. 2015. <http://users.nccs.gov/~fwang2/papers/lustre\_report.pdf>.
11. 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>.
12. Meyer, Bertrand. *Object-Oriented Software Construction*, 2nd Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1997. Print.
13. Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel*. Beijing: O'Reilly, 2006. Print.
14. "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>.
15. 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>.
16. 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>.
17. "Dentries and Inodes." Fieldses.org, n.d. Web. 28 Mar. 2015. <http://www.fieldses.org/~bfields/kernel/vfs.txt>.
18. 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>.
19. 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/>.
20. 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>.
21. Lustre Release Git. N.p., n.d. Web. 28 Mar. 2015. <http://git.whamcloud.com/fs/lustre-release.git>.
22. “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/>.
23. “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/>.
24. 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>.
25. 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>.
26. 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>.
27. "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>.
28. "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>.
29. 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>.
30. 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>.
31. 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>.
32. 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>.
33. 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 input/output (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 [4] 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.

# Index

airport staff 1

BAS Development Team 1

BASDT 2

Birdinator Autonomous System 1

Electrical, Computer, Software & Systems Engineering i, 2, 23, 24

ECSSE 2, 23

Embry-Riddle Aeronautical University i, 2, 23, 24

ERAU 2, 23

European Union 4, 23

event 5, 15, 17, 21

Global Positioning System 4, 21, 23

GPS 4, 8, 9, 10, 21, 23

IEEE Std. 830-1998 i, 1, 24

latitude 9, 21

longitude 9, 21

use case 5, 6, 8, 11, 12

diagram 5

vehicle parameters 21

waypoint 21

Waypoint Navigation 8

World Geodetic System 1984 18, 21, 23, 24

WGS84 18, 21, 23

1. For more information on CRUSH file systems, see [3] and [4]. [↑](#footnote-ref-1)
2. Within a single Lustre file system, not per OSS [↑](#footnote-ref-2)
3. For more information on the future plans for supporting other types of logical RAID configurations, refer to Lustre Contract SFS-DEV-003. See also [6], [7], [8], and [9]. [↑](#footnote-ref-3)
4. Note that the stripe size 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 [1]. [↑](#footnote-ref-4)
5. 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). [↑](#footnote-ref-5)
6. This information was published in 2009, and at the time of writing of [10], [10] 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 [1] and [2], as well as through personal communication with a researcher at Oak Ridge National Laboratories on March 6, 2015. [↑](#footnote-ref-6)
7. The Linux VFS is sometimes defined as the Linux Virtual File Switch, rather than the Linux Virtual File System [11]. [↑](#footnote-ref-7)
8. 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, there are no specific structures for directories in the VFS. Likewise, a dentry structure does not represent a file, but rather, a component in a path [11]. [↑](#footnote-ref-8)
9. 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 [11]. [↑](#footnote-ref-9)
10. For more information on the dentry cache, see [11], [13], and [14]. [↑](#footnote-ref-10)
11. For more information on walking a path and the involvement of dentries in this process, see [13]. [↑](#footnote-ref-11)
12. This figure is also supported by referencing the source code for Linux kernel 3.19 under fs/ext4/xattr.c. [↑](#footnote-ref-12)
13. While inodes use an inode number, denoted by the field i\_ino in the inode structure, to uniquely identify an inode, Lustre used 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**. [↑](#footnote-ref-13)
14. These triple indirect blocks are not shown in **Figure 15** for the sake of brevity. [↑](#footnote-ref-14)
15. Revision 355a283fce6998f5b5621adc9697d98d0fb72dfe, /lustre/llite/llite\_lib.c, line 169 of [21] [↑](#footnote-ref-15)
16. Ibid., line 490 [↑](#footnote-ref-16)
17. Ibid., line 1007 [↑](#footnote-ref-17)
18. Ibid., /lustre/llite/super25.c, line 183 [↑](#footnote-ref-18)
19. Revision 355a283fce6998f5b5621adc9697d98d0fb72dfe, /lustre/obdclass/obd\_mount.c, line 1357 of [21] [↑](#footnote-ref-19)
20. Ibid., line 1268 [↑](#footnote-ref-20)
21. Revision 355a283fce6998f5b5621adc9697d98d0fb72dfe, /lustre/llite/llite\_lib.c, line 2004 of [21] [↑](#footnote-ref-21)
22. Ibid., line 2034 to line 2054 [↑](#footnote-ref-22)
23. 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 xattr. 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. [↑](#footnote-ref-23)
24. This information is important when reconstructing the file, where the stripes must be pulled from the objects in the OST and reassembled in the correct order. [↑](#footnote-ref-24)
25. 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 and ultimately to the destination, 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 [10]. [↑](#footnote-ref-25)
26. VMWare Player 7.1.0 build-2496824 [↑](#footnote-ref-26)
27. CentOS-6.6-x86\_64 [↑](#footnote-ref-27)
28. For more information on disabling SELinux see [25]. [↑](#footnote-ref-28)
29. 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 [27]. [↑](#footnote-ref-29)
30. 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]. [↑](#footnote-ref-30)
31. For more information on maintaining the Lustre file system, see section Lustre Maintenance of [1]. [↑](#footnote-ref-31)
32. This command was found in the walkthrough presented in [28]. [↑](#footnote-ref-32)
33. 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 [1] (chapter 13 and 31, respectively). [↑](#footnote-ref-33)
34. For more information on altering the IPTables, see [33]. [↑](#footnote-ref-34)