Distributed File Recovery on the Lustre Distributed File System

J. Albano, R. Seker, R. Babiceanu, S. Oral*

albano.justin@gmail.com, {sekerr|babicear@erau.edu}, oralhs@ornl.gov

Department of Electrical, Computer, Software & Systems Engineering
Embry-Riddle Aeronautical University – Daytona Beach Campus, Daytona Beach, FL
**Oak Ridge National Laboratory, Oak Ridge, TN*

Abstract

With the advancement of cloud-computing technologies and the growth in distributed software applications, a great deal of research that has been focused on the concepts and implementations of distributed file systems to support these application. Since its inception in 1999 by Peter Braam at Carnegie Mellon University, the Lustre distributed file system has gained both the technical, as well as financial interest of some of the largest technology entities, including Oracle, Seagate, Intel, Oak Ridge National Laboratory, and OpenSFS. With this immense backing, Lustre has been incorporated in over 60% of the TOP100 high performance computers in the world and is slated to significantly increase this market share. Although the Lustre file system itself has seen a sharp increase in research since its infancy, support for many of the fields surrounding the file system has been greatly lacking. Primary among these deficiencies is file recovery on the Lustre file system. This paper attempts to fill this gap and provides a simplified solution which is then developed into a distributed solution that can scale to meet the needs and requirements of various sizes of Lustre file system deployments. While this paper focuses on the Lustre file system, the concepts and solution provided in this paper can be used on any similar metadata-based distributed file system. Although this paper does not provide an implementation of this solution, a complete solution architecture is provided, enabling further research and implementation.

1 Introduction

In the past two decades, the software industry has experienced an explosive growth in cloud computing and distributed software technology, and with this expanse, there has been a subsequent need to research and implement supporting infrastructure technologies, particularly file systems. While networked file systems have been a part of software infrastructure since the inception of the File Access Listener (FAL) in 1976, which would serve as the forerunner for the Network File System (NFS), these file systems fail to meet the growing need of distributed software to provide the vast throughput needed by these large-scale systems [1]. In essence, the file system has become to the distributed software application what the hard drive was during the age of hard disk drives: The underlying bottleneck that dictates a large portion of the Input/Output (I/O) performance characteristics of the system. In order to resolve this issue, a file system based on distributed technologies and principles was needed.

This need led to a plethora of distributed file systems, foremost among them, the Lustre Distributed File system. Created in 1999 by Peter Braam of Carnegie Mellon University, Lustre has soared in popularity in its nearly two-decade lifespan, experiencing interest by Sun Microsystems, Intel, Seagate, Oak Ridge National Laboratories, and a host of other large sponsors during this time. Leveraging this popularity, Lustre has now achieved more than a 60% market share in the TOP100 high performance computers (HPCs) and is slated to increase its growth in both the military and commercial software sectors in the coming years [2].

Even with this enormous growth, research is still lacking in supporting concepts, including file system forensics. In particular, the research surrounding the recovery of deleted files from the Lustre distributed file system, and metadata-based file systems in general, is sparse. Therefore, the research contained within this paper is intended to serve the purpose of filling this gap and provide a solution architecture that can be implemented to solve the problem of distributed file recovery.

Section 2 provides an introduction to the technical aspects that make up the Lustre file system, as well as background knowledge about the concepts (such as metadata-based distributed file systems) that will be essential to the understanding of the solution presented in this paper. Likewise, a brief history of the Lustre file system is also included to provide the historical and technological context of Lustre.

Section 3 presents the problem context for file recovery on the Lustre file system and describes a basic, conceptual solution that will later be used as the stepping stone for the distributed solution to the same problem. This solution leverages the existing technologies available for localized file recovery and treats the problem of distributed file recovery as a composite of multiple, localized file recoveries.

Section 4 leverages this conceptual model to create a distributed solution to the file recovery problem on Lustre, allowing the solution to scale to meet the needs of the particular Lustre cluster on which the recovery is being performed. This distributed solution also resolves many of the disadvantages of the conceptual solution.

The conclusion section provides an overarching description of the limitations and applicability of the solution described in section 4 and provides additional resources such as tested code examples that support the distributed solution presented in this paper.

2 Lustre File System

The Lustre file system is a metadata, object-based distributed file system that is capable of providing petabytes of storage and terabytes of aggregate I/O over a disperse network [3]. After the creation of Lustre in 1999, Peter Braam acquired Lustre under his company, Cluster File Systems (CFS); CFS was later purchased by Sun Microsystems in September of 2007 [4]. In April 2010, Oracle bought out Sun Microsystems, and thus acquired the Lustre file system [5]. In December of the same year, Oracle announced that it would discontinue long-term support for Lustre, and in response, Whamcloud and Open Scalable File Systems (OpenSFS) were created in order to continue to the development of Lustre [6], [7]. Supporting this initiative, Xyratex Ltd., a Seagate affiliate of whom Peter Braam joined in late 2010, purchased the intellectual property for Lustre and later donated the `lustre.org` domain name and logo to the open source community [8].

Although Lustre has switched ownership numerous times over its lifetime, the foundational concepts of this distributed file system have remained largely constant. Lustre is an object-based file system that divides a single, contiguous file into multiple portions called objects. These objects are then stored on various storage nodes across a Lustre cluster, allowing the once-singular file to be accessed and modified in a parallel manner. For example, if a file is divided into three parts, all three parts can be read in parallel to reconstruct the file, analogous to the technique used in Redundant Array of Independent Disks (RAID) 0 configuration.

In order to record the location of these objects in a Lustre cluster, there are two major techniques available to date: (1) a functionally deterministic algorithm or (2) a metadata service. Although deterministic functions for mapping objects of a file to storage nodes, such as Controlled Replication under Scalable Hashing (CRUSH), are a well-researched means of achieving this mapping, Lustre implements a persistent record of object mappings using a metadata service. This metadata service simply records the mapping of each object to its associated object storage node, where the key of the mapping is the object identifier and the value is the identifier of the corresponding object storage node.

The components, and there corresponding points of interaction, are illustrated below in **Figure 1**.

The client is the component that interfaces with the end-user of the Lustre file system and implements the

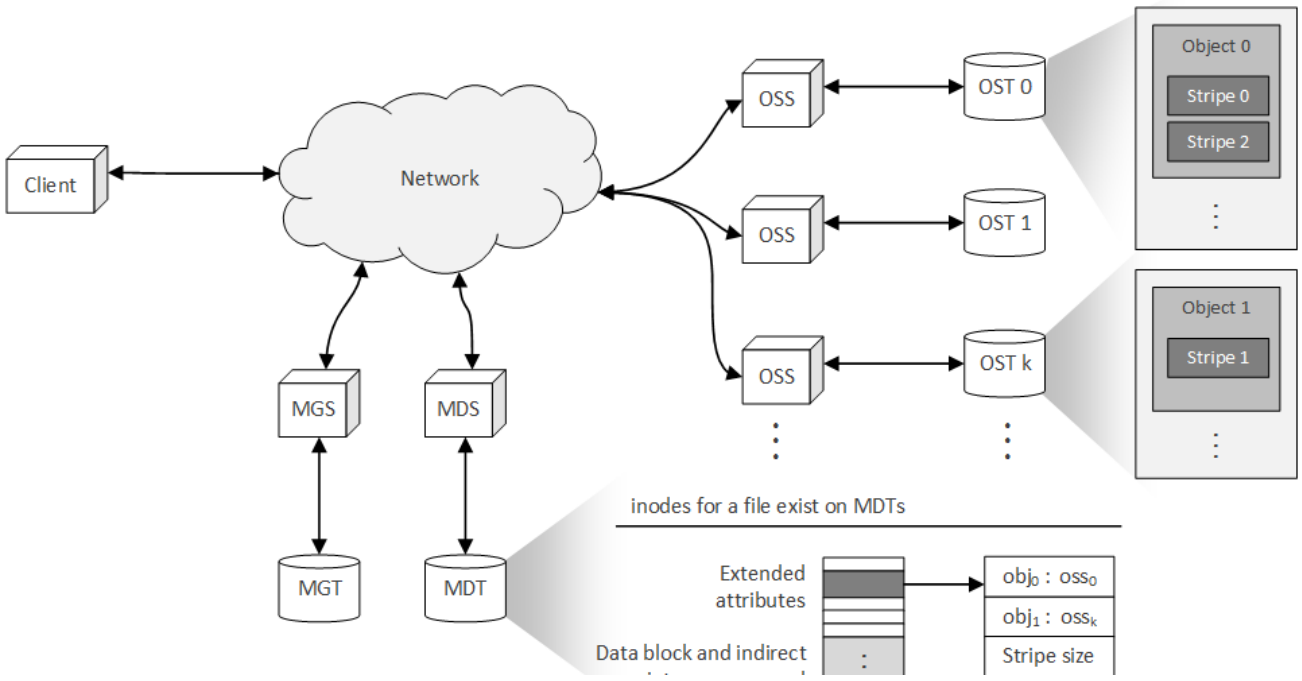

Figure 1. A Lustre cluster is divided into clients, which interface with the end-user, MGS/MGT pairs that store configuration data, MDS/MDT pairs that store metadata, and OSS/OST pairs that store the objects that compose files.

Linux Virtual File System (VFS) interface in order to provide this abstraction (this implementation is referred to as Lustre Lite, or llite). The Management Server (MGS) and Management Target (MGT) pair stores the configuration data about a Lustre cluster, and although this pair is important in the context of establishing a Lustre cluster, this paper does not focus on this pair. The Metadata Server (MDS) and Metadata Target (MDT) manage and store the metadata for the Lustre file system, respectively. The Object Storage Server (OSS) manages a series of Object Storage Targets (OSTs), where the objects associated with a file are striped across one or more OSTs. Note that each OSS can manage multiple OSTs, and that all interactions with the OSTs are performed through its corresponding OSS (the OSS serves as a proxy for its associated OSTs).

In order to store the metadata for a file, the MDS stores a shell of an inode on the MDT, where the direct and indirect block references of the inode are unused (since the data for the file does not exist locally on the MDT, but rather, on the OSTs) and the extended attributes of the inode are used to store the metadata for a file in the Lustre file system. The exact metadata stored in these Layout Extended Attributes (Layout EAs) are better understood in the context of the striping algorithm implemented by the Lustre file system.

A. Striping a File

The striping in a Lustre cluster is accomplished using a round-robin algorithm parameterized by the number of OSTs to stripe across (stripe count[1]) and the number of bytes written in each stripe (stripe size) [9]. Starting with the first OST, n bytes are written to the object corresponding to the file to be written on the OST, where n is equal to the stripe size. Once completed, n bytes are written to the object on the next OST. This process continues until the all bytes of the file have been written, or each OST has been written to exactly once; in the latter case, writing continues from the first OST and continues using the same algorithm previously described. Note that only one object exists on each OST for a given file and the object exists on each OST as a file on the local file system of the OST (sometimes referred to as the backing file system).

More succinctly, there is a one-to-one mapping between the objects that make up a file and the OSTs on which the objects are stored. Each object contains stripes $i, i+k, i+2k, \ldots$, where i is the index of object and k is the stripe count. For example, if a $40\,MB$ file is striped with a stripe size of $5\,MB$ and a stripe count of 3, the resulting striping scheme would resemble the following:

$$object_0 \rightarrow \{stripe_0, stripe_3, stripe_6\}$$
$$object_1 \rightarrow \{stripe_1, stripe_4, stripe_7\}$$
$$object_2 \rightarrow \{stripe_2, stripe_5\}$$

[1] Note that stripe count does not refer to the number of stripes, but rather, the number of OSTs that a file is striped across.

Note that each object does not necessarily contain the same number of stripes, and in general, each file in a Lustre file system can have a different stripe size and stripe count than the other files in the same file system.

B. Accessing a File

Using this scheme, there are three main steps that a client must perform in order to access a file [9], [10]:

1. **Obtain file metadata**: the client obtains the metadata for the file from the MDS
2. **Retrieve each object**: using this metadata, the client retrieves each object that makes up the file from the OSTs storing these objects (this retrieval is performed through the OSSs associated with each of the OSTs on which an object is stored)
3. **Reconstruct file**: using the metadata and the objects retrieved from the OSTs, the client can reconstruct the singular, contiguous file from its constituent objects

Using the these steps, the client component is capable of providing an end-user with the façade of a contiguous file, while in reality, the file is actually distributed in the form of objects across multiple OSTs in the Lustre cluster. While this basic procedure is the most common use case in the normal operation of Lustre, this paper also focuses on the use case performed when a file is deleted.

C. Deleting a File

Deletion, or unlinking, of a file in the Lustre file system requires two main steps [10]:

1. **Delete each object**: the object files containing the stripes of the file to be deleted are deleted from the local file system of their corresponding OSTs
2. **Delete metadata**: the inode containing the metadata associated with the file to be deleted is deleted from the MDT storing the inode

Once the object files and metadata inode for a Lustre file are deleted from the OSTs and MDT, respectively, the file considered to be deleted from the Lustre file system. Although it appears as though the objects and metadata inode are permanently deleted, they are only deleted in the sense that a file is deleted from a local file system in a localized (non-distributed) system. Using this knowledge, the recovery of a file on the Lustre distributed file system can be viewed as multiple recoveries of local files from a local file system and therefore, existing localized file recovery techniques can be used to recover and reconstruct a deleted file in the Lustre file system.

3 Conceptual Solution

Based on the steps in the deletion use case described in the previous section, recovery of a file from the Lustre file system entails the recovery of the metadata associated with the Lustre file, and the subsequent recovery of each of the objects associated with the file. Once the metadata and objects have been recovered, the once-deleted file can be reconstructed, resulting in the complete, recovered file. Viewed differently, the recovery of a file mimics the use case for accessing a file in the Lustre file system, but before the metadata and objects are retrieved, they must be recovered from the MDT and OSTs, respectively.

Therefore, the steps involved in recovering an unlinked file from the Lustre file system are as follows:

1. **Recover and obtain file metadata**: the agent recovering the file must first recover the metadata from the MDT and then store the recovered metadata for later use
2. **Recover and retrieve each object**: utilizing the recovered metadata, the OSTs on which the objects for the unlinked files exist can be discovered; the objects associated with the unlinked file are then recovered from their respective OSTs and stored on the recovery agent
3. **Reconstruct file**: using the recovered metadata and objects, the unlinked file can be reconstructed, thus recovering the file

Due to the trinary nature of this solution, it is termed the Three-Step Recovery Solution. Although the details of each steps in the Three-Step Solution have not yet been elaborated, logic already exists for the implementation of a component capable of performing each task.

In the case of the recovery of the metadata from the MDT, the metadata stored on the MDT exists in the form of an inode on the local file system of the MDT, and therefore, the recovery of the metadata from the MDT amounts to the recovery of an inode from a local file system. Numerous solutions to this problem already exist in digital forensics and the particulars of local inode recovery are not discussed further; rather, the logic capable of recovering the local inode is abstracted into a component called the Abstract Metadata Recovery Tool (AMRT), which can be used by the recovery agent to recover the metadata for a file based on a supplied, unique identifier for the file[2].

In much the same way, the objects required to reconstruct the recovered file exist on the local file systems of the OSTs. Therefore, the recovery of each object from its corresponding OST amounts to the recovery of a file from the local file system of the OST. In the same manner as this problem has already been solved for the recovery of an inode from the local file system of the MDT, a great deal of research (and implemented tools) has been devised for the recovery of a file from a local file system. Therefore, this logic is abstracted into the Abstract Object File Recovery Tool (AOFRT), which can be used by the recovery agent to recover the objects for the unlinked file.

Lastly, once the metadata and objects have been recovered through the AMRT and AOFRT, respectively, they can be reconstructed. The process of reconstructing a Lustre file from its constituent metadata and objects already exists in the Lustre code base in the form of the implementation of the standard file access use case previously discovered. Therefore, it is superfluous to create such a solution. Instead, this existing logic is reused and abstracted into the Abstract File Reconstruction Tool (AFRT), which consumes the metadata and objects for a file and returns the reconstructed file. An illustration of this Three-Step Recovery Solution, utilizing the AMRT, AOFRT, and AFRT, is depicted in **Figure 2**.

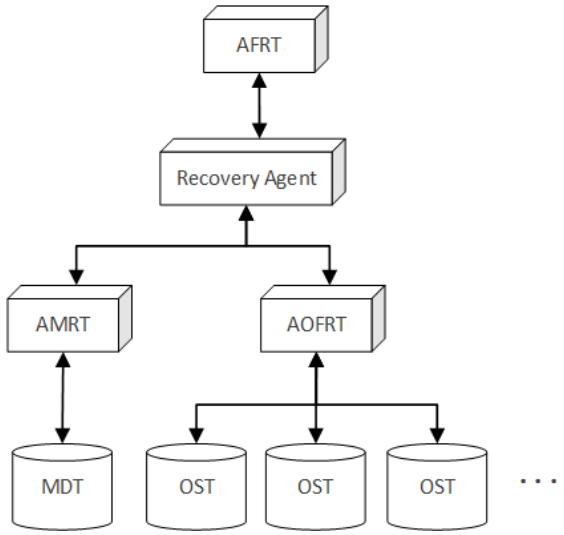

Figure 2. Using the AMRT, AOFRT, and AFRT, the metadata and objects for an unlinked file can be recovered and these constituent parts of the unlinked file can be reconstructed, resulting in the recovered file.

Although this conceptual solution suffices to recover an unlinked file, it has numerous hindrances and disadvantages in the context of a distributed environment, including: (1) this solution is localized in nature, where the recovery agent is responsible for performing each of the recovery and reconstruction actions, (2) the OSTs on which the objects of interest exist must be mounted directly to the single AOFRT, and (3) the solution does not scale to the possibly numerous OSTs on which an unlinked file exists. Bearing these issues in mind, an improved, distributed solution to this distributed problem can be created.

[2] File Identifiers (FIDs) are used in the Lustre file system to uniquely identify files at a global level in the file system (uniquely between all OSTs in a Lustre file system) and therefore, these FIDs are a natural fit for the unique means of identifying the file to recover [9].

4 Distributed Solution

In order to alleviate these localized issues, distributed software technologies can be used to create a solution that both scales to meet the size of the Lustre file system, as well as decouples the process from the central recovery agent. In the case of the Three-Step Recovery Solution, MapReduce can be used to distribute the recovery of an unlinked file [11]. The driving force behind the adoption of MapReduce for distributed recovery solution is the ability of the MapReduce architecture to map the constituent parts of a file and reduce these parts into a single, contiguous file. As it stands, the smallest portion of a file that can be recovered using the Three-Step Recovery Solution is an object (recovered from an OST using the AOFRT), but this does not provide fine enough granularity. For example, the objects of a file cannot simply be serially rearranged and ordered to reconstruct the file, since each object does not contain sequential stripes for a file.

Instead, a finer level of granularity is needed: The stripes themselves. If the stripes for a file can be obtained from the recovered objects, these stripes could in turn be keyed (mapped, in MapReduce vernacular) by the stripe index and then reduced. In essence, the reduction step of the MapReduce process would gather the stripes provided to it and reorder these strips based on the stripe index (the key for each stripe). It is important to note that the objects recovered from an OST using the AOFRT contain only a non-sequential subset of the stripes for a file. For example, in the **Striping a File** portion of Section 2, $object_0$ contains only $\{stripe_0, stripe_3, stripe_6\}$. Therefore, at each step of the reduce process, only a partial, ordered subset of the total stripes for a file are obtained. During the final reduction step, the complete, ordered set of all strips is obtained. This concept is illustrated in **Figure 3**.

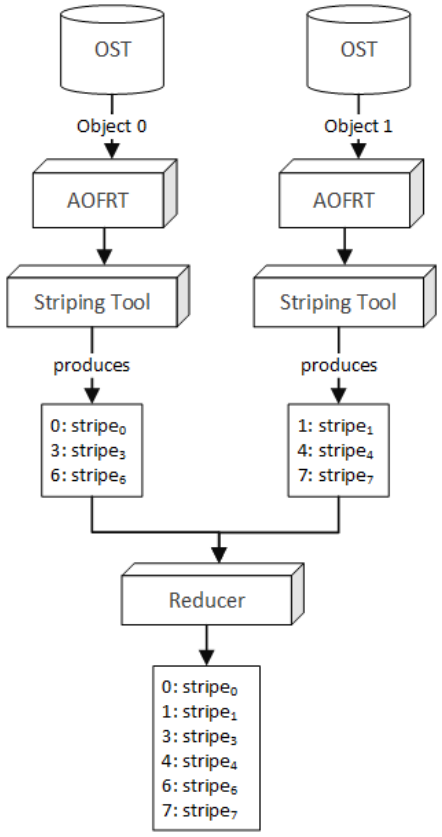

Figure 3. By dividing each recovered object into its constituent stripes, keyed by the stripe index, the partial stripe subsets can be combined and ordered until the complete set of stripes is obtained.

A. Partial Striping Component

The keen reader will notice that in order for this solution to be complete, some striping tool must be devised that consumes an object, along with the metadata for the file to be recovered, and produces a mapping of stripe indices to the stripe data for each stripe. In the context of the previous discussion of partially ordered stripes, this tool is called the Partial Striping Component (PSC). Note that the PSC acts as the mapping component in the MapReduce architecture. As will be seen shortly, the mapping algorithm for the distributed solution essentially delegates the mapping logic to the PSC, which produces the keyed stripes for an object recovered from the AOFRT. The algorithm used by the PSC for obtaining the map of stripe indices to stripe data for each stripe is enumerated in **Listing 1** (on the following page).

This algorithm presumes that the stripes of a file are conceptualized in a tabular format, where the columns of the table represent each object and each row represents one pass of the round-robin striping algorithm. This visualization, using the example data in Section 2, is illustrated in **Figure 4**.

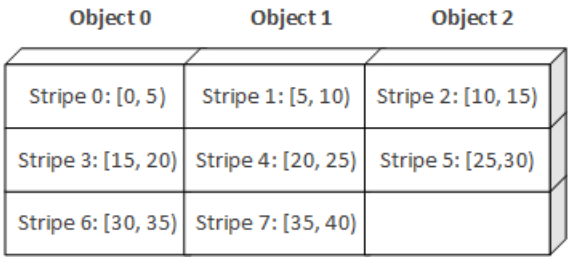

Figure 4. Striping in the Lustre file system can be viewed as a table, where columns represent the objects across which the file is striped, and rows represent a single pass of the round-robin striping algorithm.

Note that the notation $[a, b)$ is used for each stripe to denote that the data for the stripe ranges from byte a to byte b of the file being striped (where the lower bound is inclusive and the upper bound is exclusive). It is important to also note that there are two distinct categories of bounds: (1) file bounds and (2) object bounds. File bounds are synonymous with the bounds illustrated using the $[a, b)$ notation of **Figure 4** and represent the bounds seen from the perspective of the complete file (e.g. if the complete is f bytes in size, the upper bound can vary within the range

$[0, f)$). Object bounds, on the other hand, represents the bounds of the object file, rather than the complete file. For example, although the complete file in **Figure 4** is $40\ bytes$ in length, $object_0$ is only $15\ bytes$ in length, since it contains bytes [0,5), [15,20), and [30,35) from the complete file (aggregately, only $15\ bytes$). Continuing this example, the following mapping of file bounds to object bounds for $object_0$ can be computed:

$$file[0,5) \rightarrow object_0[0,5)$$
$$file[15,20) \rightarrow object_0[5,10)$$
$$file[30,35) \rightarrow object_0[10,15)$$

Although this distinction in file and object bounds may appear to be subtle, a clear delineation of these two definitions is essential to the understanding of the PSC algorithm described in **Listing 1**.

```
get_stripes (idx_obj, cnt_stripe, file_obj, size_stripe, size_file):
  is_completed := false
  idx_row := 0
  stripes := new Map
  if size_stripe = 0 or size_file = 0:
    is_completed := true
  while not is_completed:
    idx_stripe := (cnt_stripe × idx_row) + idx_obj
    bnd_low,file := idx_stripe × size_stripe
    if bnd_low,file < size_file:
      bnd_low,obj = idx_row × size_stripe
      if bnd_low,obj + size_stripe > size_file:
        bnd_up,obj := bnd_low,obj +
              (size_file − bnd_low,file)
      else:
        bnd_up,obj := bnd_low,obj + size_stripe
      end
      data := file_obj.read (bnd_low,obj, bnd_up,obj)
      stripes.add (idx_stripe, data)
      idx_row := idx_row + 1
    else:
      is_completed := true
    end
  end
  return stripes
end
```

Listing 1. Using the metadata for a file, a list of stripes, keyed by the index of the stripe, can be obtained algorithmically.

Leveraging the conceptualization of striping illustrated in **Figure 4**, the PSC algorithm requires the index of object from which the stripe are being retrieved, idx_{obj}, which in essence, represents the column index of the striping table depicted in **Figure 4**. Note that all indices using in the PSC start at 0, not 1. The algorithm also requires the following parameters: the stripe count, cnt_{stripe}; file representing the object, as recovered by the AOFRT, $file_{obj}$, which contains the data for the object and therefore the stripe data; the size of each stripe, $size_{stripe}$, in bytes; and the size of the file, $size_{file}$, in bytes.

Using this input data, the PSC algorithm then iterates until all stripes from an object have been extracted. In order to determine that all stripes from an object have been extracted, the lower bound of the stripe data, from the perspective of the file to be recovered $bnd_{low,file}$, is computed by taking the product of the stripe index and size of the stripe. For example, if the stripe index is 3, the file lower bound will be 15, as illustrated in **Figure 4**. In order to compute the current index of the stripe using the tabular conceptualization, the index of the first stripe for each row is calculated by taking the product of the row index and the total number of columns, equal to the count of objects, shifted by the column index of the object (by adding the column index).

For example, to find the index of the second stripe for $object_1$ is calculated as follows: The index of the first stripe on the second row is equal to 1 (the row index) multiplied by 3 (the total number objects over which the file was striped, or the stripe count), or 3. This initial index is then shifted by 1 (the index of the object, or column index), which results in 4, the index of the second stripe contained in $object_1$.

If the file lower bound is greater than the size of the file (the complete file, not the object file), then all stripes for this object have been extracted and the algorithm completes. For example, the index of the third stripe of $object_2$ (which does not exist) is 8, and therefore, the file lower bound for this stripe is equal to 40. Since this lower bound is inclusive, an the file only contains bytes [0,40), there does not exist any bytes to read from the third stripe of $object_2$, and therefore, all stripe data has been extracted.

If there still remains stripe data to extract (not all stripes have been extracted from the object file), then the lower and upper bounds of the stripe, from the perspective of the object file, $bnd_{low,obj}$ and $bnd_{up,obj}$, respectively, are recalculated. If number of the bytes remaining in the file is less than the sum of the file lower bound and the stripe size, then this stripe is the last stripe in the file and contains less bytes than the stripe size. In this case, the object upper bound is equal to the object lower bound, plus the number of bytes remaining the file (the difference of the file size and the file lower bound).

If the file size is greater than the sum of the file lower bound and the stripe size, there are more bytes remaining in the file than the stripe size, and thus, this stripe contains exactly $size_{stripe}$ number of bytes. Note that even though

the size of this stripe is equal to the stripe size, it may also be the last stripe, where the size of the file being striped is wholly divisible by the stripe size. This occurs when the file size meets the criteria enumerated in equation (1) below:

$$(size_{file})\ mod\ size_{stripe} = 0 \quad (1)$$

where mod is the modulus operator.

With the stripe index and object lower and upper bounds computed, the bytes from the object file can be read using the $read(..)$ function. This function takes the inclusive byte index of the first byte to read from the file as its first argument, followed by the exclusive byte index of the last byte to read from the file. More concisely, this method reads from the object file with the following bounds: $[bound_{low}, bound_{up})$. After obtaining this stripe data, this data, along with its corresponding stripe index, is added to a map with the following format,

$$idx_{stripe} \rightarrow data_{stripe}$$

and the index of the row is incremented. When the algorithm completes, this map is returned. Thus, using this algorithm, the PSC can consume the recovered object file, $file_{obj}$, and the recovered metadata, which contains idx_{obj} (since the column index is equal to index of the object), cnt_{obj} (since this information can be implicitly obtained by counting the number of $object \rightarrow OST$ map entries in the metadata for a file), $size_{stripe}$, and $size_{file}$, and produce a map containing the data for each stripe contained in the supplied object, keyed by the index of said stripe.

It is important to note that the conditional statement

$$size_{stripe} = 0\ \boldsymbol{or}\ size_{file} = 0$$

at the preamble of the algorithm in **Listing 1** is an optimization that shortcuts the algorithm if either the stripe size or file size is 0. In either case, the algorithm should conclude immediately and an empty mapping of stripes should be returned. By adding this optimization in the preamble of the algorithm, the remaining portion of the algorithm can assume, using De Morgan's Law, that the following condition holds,

$$size_{stripe} \neq 0\ \boldsymbol{and}\ size_{file} \neq 0$$

simplifying the core of the striping algorithm.

B. Solution Architecture

The main disadvantage of the conceptual solution presented in section 3 is that it is centralized about the recovery agent, and therefore, all persistent targets (the MDT and the OSTs) must be directly connected to the agent, or to the AMRT and AOFRTs, which are in turn connected to the agent. As the number of OSTs on which a file is striped increases, this solution degrades and becomes practically infeasible. The improved, distributed solution architecture is illustrated in **Figure 5**.

Instead of connecting each OST to the recovery agent,

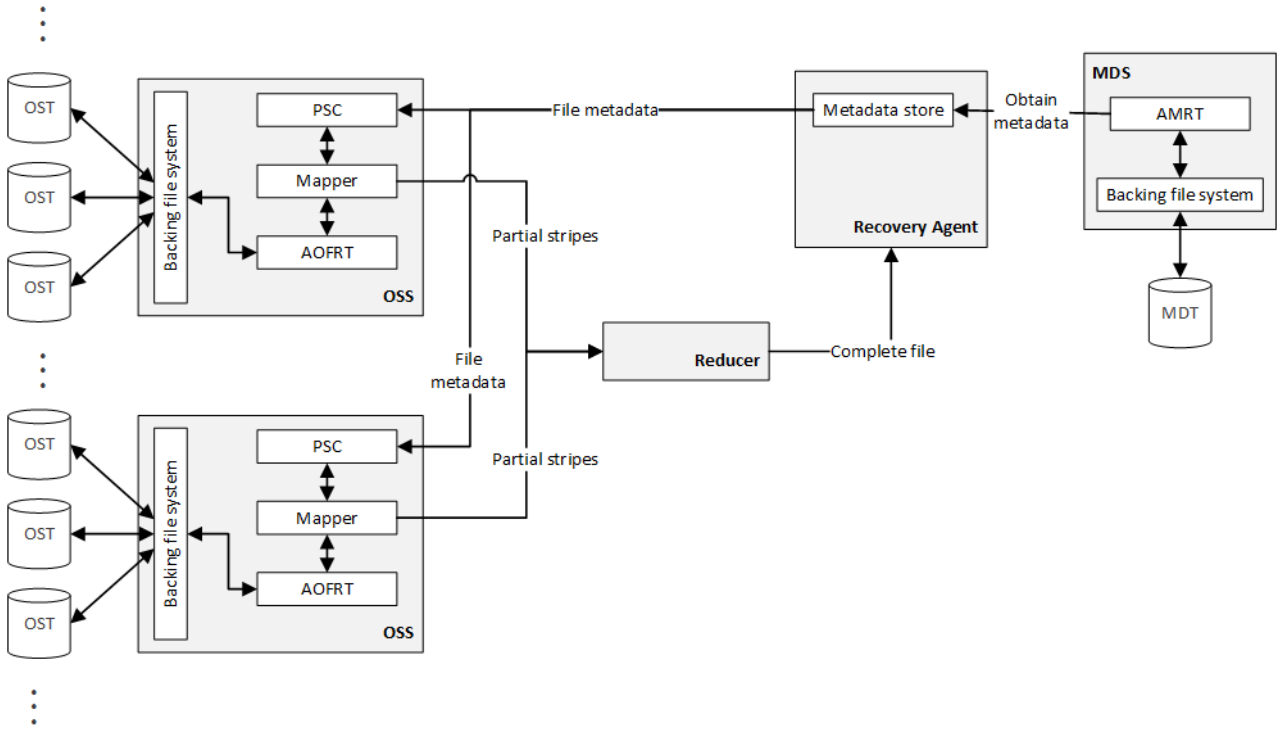

Figure 5. Using the MapReduce architecture, the stripes that constitute the recovered file flow from each OSS, managing an OST on which the objects reside, to the recovery agent through a series of reductions, ultimately resulting in the complete, recovered file.

a PSC is placed on each of the OSSs, which decouples the agent from direct contact with each of the OSTs containing objects of interest. When a recovery is initiated, the recovery agent recovers the metadata for the file of interest and stores it, keyed by the identifier for the file to recover, in a metadata store that is accessible by the PSC on each OSS. The recovery agent then informs the PSC on each of the OSSs (all PSCs) that a recovery has been initiated, supplying the identifier for the file to recover. The PSCs then obtain the metadata for this file and provide it to the Mapper component. The Mapper component then uses the metadata to decide if the objects for the file are contained (or, were contained prior to unlinking) on any of the OSTs managed by the OSS on which the Mapper resides.

If the OSS on which the Mapper resides does not manage any OSTs containing the objects for the file to be recovered, the recovery request completes on that OSS. If, instead, the Mapper discovers that the OSTs managed by the OSS on which the Mapper resides contain objects that make up the file, the Mapper then uses the AOFRT on each OSS to recover these objects. Once all the pertinent objects have been recovered, the objects, along with the metadata for the file to be recovered, are supplied to the PSC, which produces a map, where the keys are the stripe indices for the stripes contained in the supplied object and the value for each key is the stripe data corresponding to the stripe index. The PSC then returns this map (for each object it is supplied) to the Mapper, which then forwards these maps onto the reducer.

The reducer then combines and sorts these maps to produce a complete set of all stripes for the file to be recovered. This complete stripe set is then returned to the recovery agent that initiated the recovery process. The recovery agent can then obtain the data for the file by reading the value for each map entry, in order of the index. For example, the value for index 0 is read, then the value for index 1 is read, and so forth. At the completion of this process, the recovered file resides on the recovery agent in accordance with the purpose of the recovery process.

C. Notes & Improvements

There are some important notes to consider in this solution architecture. First, the PSC has two main responsibilities: (1) to act as the point-of-contact between the OSS and the recovery agent, and (2) to extract and map the stripes obtained from the objects recovered by the AOFRT. These responsibilities are disparate beyond the fact they both pertain to the recovery of a file, and therefore, the former responsibility can be divided into a secondary component, called the Point-of-Contact Component (PCC). This PCC is then responsible for obtaining the metadata from the metadata store on the recovery agent and passing this metadata to the Mapper component. Using this scheme, the Mapper only uses the PSC as a delegate from which the mapped stripes for the recovered objects are obtained. This improved decoupling is illustrated in **Figure 6**[3].

Apart from this improvement of the PSC, it is important to note that the number of reducers in this architecture is not fixed (or not simply one, as depicted in **Figure 5**). Instead, as is part of the MapReduce architecture, the number of reducers can scale to meet the needs of the recovery process. For example, if a small file to be recovered is striped across only a few OSTs, then only a few reducers will be needed. If instead, a very large file to be recovered is striped across a large number of OSTs, then many reducers can be used to distributed the reduction process and offload to required work on a large number of compute nodes. In a sense, the distributed

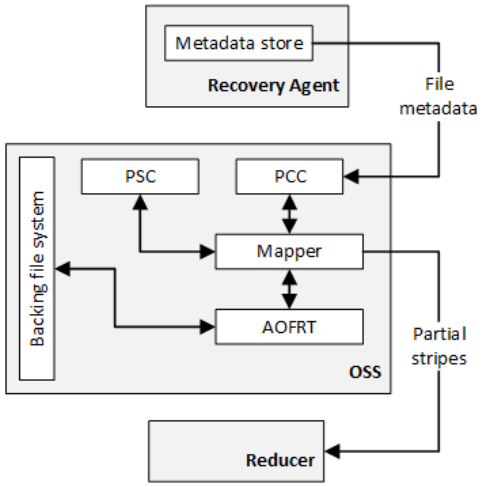

Figure 6. By dividing the two responsibilities of the PSC into the PCC and PSC, respectively, the PSC maintains higher cohesion and increases the singularity of its purpose.

Three-Step Recovery Solution can be scaled to meet the challenge of theoretically recovering any size file with any type of striping pattern (e.g. large number of objects or small number of objects).

5 Conclusion

Although a great deal of research has been completed in previous decades on distributed file systems, primarily the Lustre distributed file system, a gap has formed in the research of technologies that support these complex systems. Foremost among these research disparities is the forensics and file recovery techniques surrounding the Lustre file system. This paper serves the purpose of filling this gap and providing an implementable solution architecture that can be used to create a distributed resolution to this naturally distributed problem.

At the time of writing, the solution architecture described in this paper has not been implemented in its

[3] The solution architecture does not make this distinction, as the purpose of **Figure 5** is to provide a conceptualized, distributed solution, rather than design and optimize the intricacies of the solution.

entirety, but this does not detract from its practically and pragmatism. To the contrary, the main algorithm that makes this solution possible, as described in **Listing 1**, has been implemented in a simulated environment and tested using a suite of automated tests. The source code for the implementation of this algorithm, along with its battery of automated tests, can be found at `https://goo.gl/bl6mX6`.

As a supplemental note, the algorithms and solutions described in this paper are not strictly limited in scope to the Lustre distributed file system. Instead, these solutions are described in terms of the general concepts common among all metadata-based file systems and therefore can be extended to any metadata-based distributed file system by replacing the Lustre-specific terminology used in this paper with the respective terminology of the metadata-based distributed file system for which this solution is ported.

Acknowledgements

This research would not have been possible without the patient help and support of Dr. Remzi Seker at Department of Electrical, Computer, Software, and Systems Engineering at Embry-Riddle Aeronautical Universi;ty. I am deeply grateful for the technical knowledge and advisement you have provided throughout this endeavor. I would also like to thank Dr. Sarp Oral at Oak Ridge National Laboratory for his expertise on the Lustre file system. Although our correspondence was brief, the information you have provided me is invaluable and was directly responsible for making this research possible. All mistakes and errors in the paper are solely mine.

Joshua 24:15

—J.A.

References

[1] Jones, M. Tim. "Network File Systems and Linux." *Network File Systems and Linux*. IBM DeveloperWorks, 10 Nov. 2010. Web. 02 Apr. 2015.

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

[3] Rutman, Nathan. "Rock-Hard Lustre Trends in Scalability and Quality." (n.d.): n. pag. *OpenSFS: The Lustre File System Community*. Xyratex, 2011. Web. 2 May 2015.

[4] Petersen, Torben K. *Inside The Lustre File System* (n.d.): n. pag. Seagate. Seagate. Web. 26 Mar. 2015.

[5] Montalbano, Elizabeth. "Update: Oracle Agrees to Buy Sun for $7.4B." *InfoWorld*. N.p., 20 Apr. 2009. Web. 3 Apr. 2015.

[6] Brueckner, Rich. "Inside Track: Oracle Has Kicked Lustre to the Curb - InsideHPC." *InsideHPC*. N.p., 10 Jan. 2011. Web. 03 Apr. 2015.

[7] Gorda, Brent. "Whamcloud Aims to Make Sure Lustre Has a Future in HPC - InsideHPC." *InsideHPC*. N.p., 20 Aug. 2010. Web. 03 Apr. 2015.

[8] “Xyratex Advances Lustre® Initiative, Assumes Ownership of Related Assets.” *Seagate*. N.p., 19 Feb. 2013. Web. 03 Apr. 2015.

[9] "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>.

[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 Laboratory*, Apr. 2009. Web. 27 Mar. 2015. <http://users.nccs.gov/~fwang2/papers/lustre_report.pdf>.

[11] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." 6th Symposium on Operating Systems Design & Implementation (2004): n. pag. Google.com. Google, Inc., 2004. Web. 6 Apr. 2015.