Metadata Management in Distributed File Systems

University of Oulu
Faculty of Information Technology and Electrical Engineering
Bachelor's Thesis
Joni Laitala
6.9.2017

Abstract

The purpose of this research has been to study the architectures of popular distributed file systems used in cloud computing, with a focus on their metadata management, in order to identify differences between and issues within varying designs from the metadata perspective.

File system and metadata concepts are briefly introduced before the comparisons are made.

Keywords
big data, distributed file system, distributed storage system, metadata management

Supervisor
Dr., University Lecturer Ari Vesanen

Contents

Abstract 2
Contents 3
1. Introduction 4
2. File Systems 5
    2.1 Disk File Systems 5
    2.2 Cloud Distributed File Systems 5
    2.3 Metadata 6
3. Examples of Distributed File Systems 7
    3.1 Apache Hadoop and Hadoop Distributed Filesystem (HDFS) 7
    3.2 Lustre 7
    3.3 OrangeFS 8
    3.4 GlusterFS 8
4. Metadata Handling 9
    4.1 Distributed File System Architectures 9
    4.2 HDFS 9
    4.3 Lustre 11
    4.4 OrangeFS 13
    4.5 GlusterFS 14
5. Comparison of Architectures 15
    5.1 Common Issues and Solutions 15
    5.1.1 Centralized Metadata 15
    5.1.2 Directory Distribution 15
    5.1.3 Metadata Allocation in Metadata Server Clusters 16
    5.2 Summary of Reviewed Distributed File Systems 17
6. Conclusions 18
7. References 19

1. Introduction

Cloud computing and storage have evolved to become almost an industry standard for growing online services and high performance computing (HPC) over the last 10 years. Any growing web service that hosts and streams media, files or software benefits from this so called horizontal scalability that cloud storage brings, as they need only as much hardware as is needed for processing and storage, and more can be added without disturbing availability. A combining challenge for both HPC and service is the need for efficient reading and writing of data when the storage devices are scattered across computer nodes (Chen & Liu, 2016). Cloud storage can be hosted on off-the-shelf hardware or rented virtual servers. Today anyone could create their own fully functioning "cloud" with open-source software, and a handful of computers with ordinary hardware.

In order to store files across computers, a cloud computing platform typically utilizes some form of distributed file system (DFS). Distributed file systems combine the parallel data processing power of multiple computer nodes into one logical abstraction of the namespace spanning across the cluster. A question that rises from the use of distributed file systems is how the metadata of the stored data is processed and stored in the cluster, when the data is spread across a cluster and any node may disconnect at any time.

This thesis will focus on the way metadata is handled in distributed file systems. This will be done by comparing the metadata handling techniques of four popular distributed file systems with slightly differing intended uses; Lustre, the Hadoop Distributed Filesystem, OrangeFS and GlusterFS. The way metadata is stored, and their location in the cluster and how the integrity of metadata is ensured in the different DFSs will be compared.

In Chapter 2 the basic concepts of file system, metadata and distributed file system will be introduced. Chapter 3 introduces the distributed file systems that are reviewed in the thesis in more detail. The metadata handling of the four example DFSs are reviewed in Chapter 4. Finally a comparison and the conclusions are made in Chapter 5, common metadata handling issues are also reviewed in the fifth chapter before conclusions.

2. File Systems

File systems are a tool to handle files stored on memory. They define how data is arranged and handled on a storage media. Different file systems exist for various purposes, for example optical disks have their own standard that is specifically engineered for them to ensure optimal performance (Volume and file structure of CDROM for information interchange, 1987), and flash memory devices have their own file system to consider the properties of flash storage. A file system may even be temporary with short-term needs and exist in a system's main memory, as is the case with tmpfs (tmpfs, 2017). The nature of the storage device affects how the file system is implemented. Many storage methods have multiple file system options available, where each are tailored for a specific purpose, bringing advantages over others in more defined environments.

2.1 Disk File Systems

Operating systems utilize their own file systems for handling local data on the computer's mass storage, be it hard disk drives (HDD), solid-state drives (SSD) or other media. It is good to know the terminology; a single computer could have multiple file systems in use at once for each disk partition, but the type of these file systems may or may not be different.

A file system is created with a specialized application that prepares a disk for the file system (Wirzenius, Oja, Stafford, & Weeks, 2003). File systems are typically operating system-specific. Linux has more than a dozen file systems available to it. Windows machines also have their own file systems, the new technology file system (NTFS) being the most used.

However, on large servers that handle big data, ordinary file systems are not very efficient. In these cases, there system can usually benefit from an abstraction of multiple machines' file systems into a single distributed file system (DFS), where the interface can be accessed from a single interface (Thanh, Mohan, Choi, Kim, & Kim, 2008).

2.2 Cloud Distributed File Systems

A distributed file system (DFS) is a storage system spread across multiple storage devices. It essentially unifies the local file systems of computer nodes, creating a single large file system with a unified namespace. The local disk file systems of each computer lay underneath the DFS, and file data is spread across each node, each with its own storage. A DFS is logically similar to any other file system, its interface is comparable to the interface of an ordinary disk file system (Shvachko, Kuang, Radia, & Chansler, 2010). What happens in the back-end does not necessarily concern the user, for file operations they can simply use the file system through a UI of their choice or through a standard command-line interface with commands familiar from UNIX. This property is called network transparency (Thanh et al., 2008).

In Network File Systems (NFS) it is common to have multiple servers for hosting large amounts of data and their backups (Sandberg, 1986). However, in this older protocol the data gets fragmented into separate "silos" that are not linked together efficiently (Thanh et al., 2008). Performance and scaling is limited by single server bandwidth, creating a bottleneck. This is where the newer DFSs can prove to be effective. Numerous projects

exist that implement DFSs for different platforms, such as search engines, grids, computing clusters, et cetera (Thanh et al., 2008). Many of them are open-source software, and can be set up with relative ease.

One of the impressive capabilities brought by DFS is the abstraction of multiple local file systems into a single large file system. The abstraction happens over multiple server nodes that can be spread over different physical locations and devices (Vadiya & Deshpande, 2016). The DFS interface can look and act logically like a conventional file system when observed from outside, save for the latency from the overheads caused by networking (Thanh et al., 2008). In Hadoop for example, the users can browse the UNIX-like file system from a web-based interface if they wish to do so, and do not need to understand the complexity of the underlying network of computer nodes. When a user copies a file into a DFS, in the background the file is split into blocks and replicated across the cluster. This means that a single file is spread across multiple servers, enabling the read/write processes to happen in parallel, resulting in an efficient system for processing bulk data.

Often files are also replicated a number of times to ensure availability if one of the nodes falls offline unexpectedly (Buyya, Calheiros, & Dastjerdi, 2016; Verma, Mansuri, & Jain, 2016). This could be compared to the RAID 10 protocol, where data is not only striped on different partitions to maximize read and write speeds, but also replicated a number of times to improve availability, but on a cluster level. Thanks to efficient replication and metadata handling, some DFSs can even retain complete availability and integrity of all files and services if an entire server rack falls offline unexpectedly, and automatically restore and re-spread the file blocks once the missing nodes are back online (Shvachko et al., 2010; Singh & Kaur, 2014).

2.3 Metadata

Metadata is information about other data (Riley, J. 2017). It is included almost anywhere where information is archived. For example, retailers have information about their customers, libraries about the books they store and warehouses about their inventory. Metadata makes it more efficient to manage and quickly obtain information about data, without having to go through all the actual data (Riley, 2017). Metadata also may include additional information about the data, which the data itself does not contain, such as ownership information or creation date.

There are two kinds of file metadata; structural metadata and descriptive metadata (Chen & Liu, 2016). Structural metadata is information about the data containers. Descriptive on the other hand describes the data content itself, file locations, creation date, file permissions and so on. Most importantly metadata helps locate the files and their parts across nodes (Chen & Liu, 2016). The metadata managed in local file systems and DFS differs a bit. In DFSs it is often necessary to store additional DFS related information, such as the location of file chunks in the cluster.

A DFS needs an efficient way of handling its metadata due to the limitations of parallel computing and network messaging. This means that one or more, or even all the nodes in the cluster are responsible for containing file metadata and keeping it up-to-date. Exactly what and how the metadata is stored varies between solutions, and will be examined more closely in the next chapters.

3. Examples of Distributed File Systems

Some example implementations of Distributed File Systems will be provided to further explain the concept. Many solutions exist in the market due to the fact that one file system architecture cannot satisfy the needs of every use-case scenario. Some DFS are designed to be hosted on commodity hardware, like the traditional spinning disk drives, and typically do not receive cost-effective benefit from using Solid State Disks or other fast flash media. Others are more suited for High Performance Computing (HPC) where the hardware can be much more efficient (Chen & Liu, 2016). The commercial purpose of the system also matters. HPC systems do not usually provide services, but big data platforms are often built for the purpose of serving data to clients and also receiving enormous amounts of streaming data through the network, only limited by the size of the cluster. A HPC system might not even be connected to the internet. These differences can naturally affect the DFS architecture.

3.1 Apache Hadoop and Hadoop Distributed Filesystem (HDFS)

Hadoop is an Apache open-source project. Hadoop is very modular, with dozens of compatible open source modules available. Its modules were designed with the assumption that failures are common and should be automatically handled by the framework (Chen & Liu, 2016). Hadoop is able to handle massive I/O streams (Vadiya & Deshpande, 2016). It has gained popularity in Big Data solutions and has spawned several commercial distributions that provide easier implementation and subservice version compatibility with more automated installations. Some of its supported features include dynamically scalable clusters, machine learning and handling bulk streaming data.

At Hadoop's core is its native file system, the Hadoop Distributed Filesystem (HDFS), which is designed to store and process tens of petabytes of data reliably on commodity hardware, as well as streaming the data for various user applications (Shvachko et al., 2010). HDFS is Java-based unlike the majority of other DFSs that are based on C/C++. A typical fully distributed HDFS platform has at least 3 nodes, but can scale to tens of thousands of nodes (Shvachko et al., 2010). Files are typically split into 128Mb chunks, but this is user customizable even on individual file level. By default, each file block is replicated three times by the framework on different storage devices, but this is also customizable (Shvachko et al., 2010). The next version of Hadoop will support Erasure Coding as an option to replication. Erasure Coding reduces the storage cost roughly by half compared to 3 times replication. HDFS originates from the Google File System (GFS) (HDFS Architecture Guide, 2017).

3.2 Lustre

Lustre is an open-source file system designed for High Performance Computing platforms i.e. supercomputers (Vadiya & Deshpande, 2016). It is one of the most successful HPC file systems, with multiple machines in the top 10 most powerful supercomputer clusters using it. Lustre has also been used in datacenters as a back-end file system, some examples in the Lustre Filesystem White Paper (Sun Microsystems Inc., 2007) include Internet Service Providers and large financial institutions.

The amount of Lustre clients in a cluster can vary from 1 to more than 100,000. Lustre is able to handle tens of petabytes of storage (Chen & Liu, 2016). A cluster's aggregate input-output can exceed a terabyte per second. As is typical to most DFS's, I/O throughput increases dynamically as new servers are added into the cluster. Lustre computing clusters can be combined to form even larger clusters (Lustre Software Release 2.x Operations Manual, 2017).

3.3 OrangeFS

OrangeFS is an open source distributed file system that focuses on cluster computing. It was split into another project from an older Parallel Virtual File System (PVFS2) in 2011, but PVFS is still being developed next to OrangeFS as separate projects (Yang, Ligon III, & Quarles, 2011). OrangeFS adds more support for interfaces and aims for a larger audience than PVFS (Moore et al., 2011). It seems that OrangeFS is best compared to Lustre in that it focuses on high performance computing, but the modularity and wider range of potential use cases and interfacing options resemble the HDFS file system. It could be said that its customer base lie somewhere between the ones of HDFS and Lustre.

OrangeFS enables parallel computing in that it allows file objects to be read by multiple machines in parallel. Like Hadoop, it allows larger computational problems to be split into smaller ones that can be solved by multiple machines in parallel (OrangeFS Documentation, 2017).

Some of the unique features of OrangeFS are its distribution of metadata to storage servers, and its storage and transmission of files as objects across many servers.

3.4 GlusterFS

GlusterFS is also an open source scalable distributed file system. Its applications are mainly in cloud computing, content delivery and media streaming (Chen & Liu, 2016). Its maximum capacity reaches petabyte level. GlusterFS supports Ethernet and the very fast Remote Direct Memory Access for node-to-node communications.

Chen and Liu (2016) point out that interestingly, GlusterFS is not using metadata in file saving. Instead it uses a Hash algorithm for choosing a location to save to, an additional query index is not needed as a result. GlusterFS supports mirroring of data for integrity, and node-parallel distribution and striping of data for efficiency. A GlusterFS cluster can balance loads by writing data on relatively idle servers.

4. Metadata Handling

The way metadata is managed in DFSs varies depending on the architecture they are implemented on. We will take a look at what kind of architectures are common, and then look at how each DFS handles the storage and management of metadata in practice.

4.1 Distributed File System Architectures

Thanh et. al. (2008) describe different DFS architectures in their "Taxonomy and Survey on Distributed File Systems". The client-server architecture provides a standardized view of the local file system, allowing different operating systems to access the same file system. It is not a DFS in itself, but may be part of one. A cluster-based distributed file system is a multiple-node DFS with a master node and up to hundreds of chunk servers under one master node. In it the master node handles the metadata of the cluster. Here the Metadata is decoupled from the data. One such example would be the HDFS (Shvachko et al., 2010). The symmetric architecture is peer-to-peer based. In it the clients also host the metadata, so each client understands the disk structures. In the asymmetric architecture, for example Lustre, there are one or more dedicated metadata managers.

4.2 HDFS

The architecture of a HDFS cluster is relatively simple. It is often described as the Master/Slave architecture, so it fits in the cluster-based category. Important components include the NameNode, possible secondary NameNode and two or more DataNodes (HDFS Architecture Guide, 2017). NameNode and DataNode are Java-based applications typically running on dedicated machines. Files and directories are stored on the NameNode machine and include attributes such as the namespace, modification times and file permissions. NameNode can be considered to be the server on a client-server architecture. The data itself is stored on DataNodes, often called Slave nodes. The minimal fully functional configuration of Hadoop has at least three nodes, one NameNode and two DataNodes. A NameNode can in practice also take the roles of a DataNode, but this would not be a truly efficient distributed setup.

The namespace and all of the file metadata is stored on the NameNode, so this component will be the focus when we evaluate metadata on HDFS. Each file is split into chunks, and each chunk is given a unique handle that is stored on the NameNode. In order to retrieve a complete file, a client must refer to the NameNode first to know which DataNodes contain the chunks.

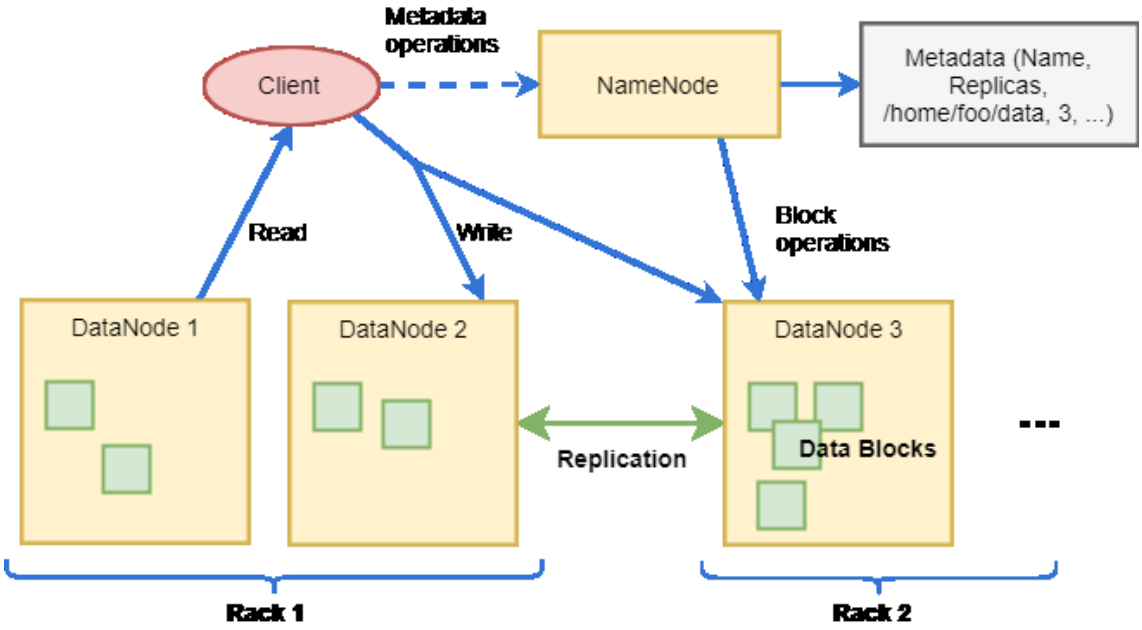

Figure 1. HDFS cluster architecture and file operations (HDFS Architecture Guide, 2017)

In Figure 1 we can see the architecture of a HDFS cluster. There is one NameNode handling the metadata, and a number of DataNodes storing data blocks. A client program accessing HDFS first consults NameNode on the metadata, and then accesses the DataNodes directly to get information. No actual file data flows through the NameNode itself, reading and writing is done on the DataNodes only. HDFS is also rack-aware, meaning that replications will be done on different server racks automatically if the rack layout is defined (HDFS Architecture Guide, 2017). In the event of rack failure in a multi-rack cluster with file replication, at least one full set of file blocks should always survive, maintaining full availability.

Every metadata change is recorded on a transaction log called EditLog. The HDFS official documentation (2017) gives an example; when a new file is created in HDFS, the NameNode inserts a record in the EditLog. Also, changing the replication factor of a file does the same. The EditLog is simply a file in the NameNode's local file system, not in the DFS. The entire namespace, mapping of blocks and files and the properties of HDFS is stored in another file called FsImage. The FsImage file is also stored in the local file system of the NameNode.

It would not be efficient to have the file system's namespace only on the disk. The active namespace and file Blockmap is also loaded in the NameNode's system memory, which is much faster to use. It is stated that 4Gb of RAM is more than enough to host a "huge" amount of file metadata (HDFS Architecture Guide, 2017). This is due to the design principle that Hadoop is designed for commodity hardware. The reason for storing the file system image also on the disk, is because the RAM is volatile memory. If power is lost on the system or a reboot is done, all the memory stored on RAM is lost. Disk drives instead have permanent memory. On NameNode system startup, the data in FsImage are loaded into memory. Then the EditLog is read, and all the changes in it are applied to the RAM representation of FsImage. After the process is complete, the in-memory FsImage will be written over the file version of the FsImage. This is called checkpointing. Periodic checkpointing is also done occasionally while the system is running.

The FsImage and EditLog are very crucial to the Hadoop Distributed Filesystem. If any of them is corrupted, the entire DFS may become non-functional. To counter this, the NameNode can be configured to maintain copies of the two files, synchronously updating them as the actual files are updated. Maintaining multiple copies might affect the NameNode's metadata handling efficiency, but the official HDFS architecture documentation (2017) states that this is unlikely to be a real problem, as HDFS applications are not very metadata intensive, merely data intensive. If copies are used, the NameNode chooses the most up-to-date consistent FsImage and EditLog to use.

The NameNode is a single point of failure unless a backup NameNode is designated (Weets, Kakhani, & Kumar, 2015). During normal operation, the backup NameNode is mirrored alongside the actual NameNode, and when failure happens, the cluster switches to using the backup NameNode. If no backup NameNode is used, the problem has to be fixed manually and the cluster is non-functional until the NameNode is back running.

The DataNodes store the actual data of the files. Data is stored as data objects called blocks, and each block is an individual file in its own local file system (HDFS Architecture Guide, 2017). An individual DataNode does not know of the actual files. Storing all files in a single directory would not be efficient due to the nature of local file systems. Instead, the DataNode determines the optimal file directory structure, such as the number of files in a directory, and creates subdirectories as needed based on a heuristic. Upon startup the DataNode creates what is called a Blockreport. It scans its local file system, collecting information on each data block it finds into a list, and then sends the list to the NameNode (HDFS Architecture Guide, 2017).

4.3 Lustre

There are three key server components in the Lustre architecture; metadata servers (MDS), the object storage servers (OSS) and the Management Servers (MGS). (Lustre Software Release 2.x Operations Manual, 2017). Object Storage Servers handle the actual object data storage, and the metadata servers process the metadata storage. Management servers (MGT) contain file system registries and configuration information, which provide this information to other Lustre components as needed. The disk storage for actual data are referred to as object storage targets (OST) and the metadata equivalents as metadata targets (MDT). Object storage and metadata storage servers are grouped into failover pairs in Lustre. If one fails, the other will take its place (Sun Microsystems Inc., 2007). The scalability of Lustre is achieved by adding new pairs of object storage servers with their own Object Storage Targets (Lustre Software Release 2.x Operations Manual, 2017). Up to four metadata targets can usually be added for metadata Servers. A single MDS is known to handle 3 billion files in practice, according to official Lustre documentation (2017).

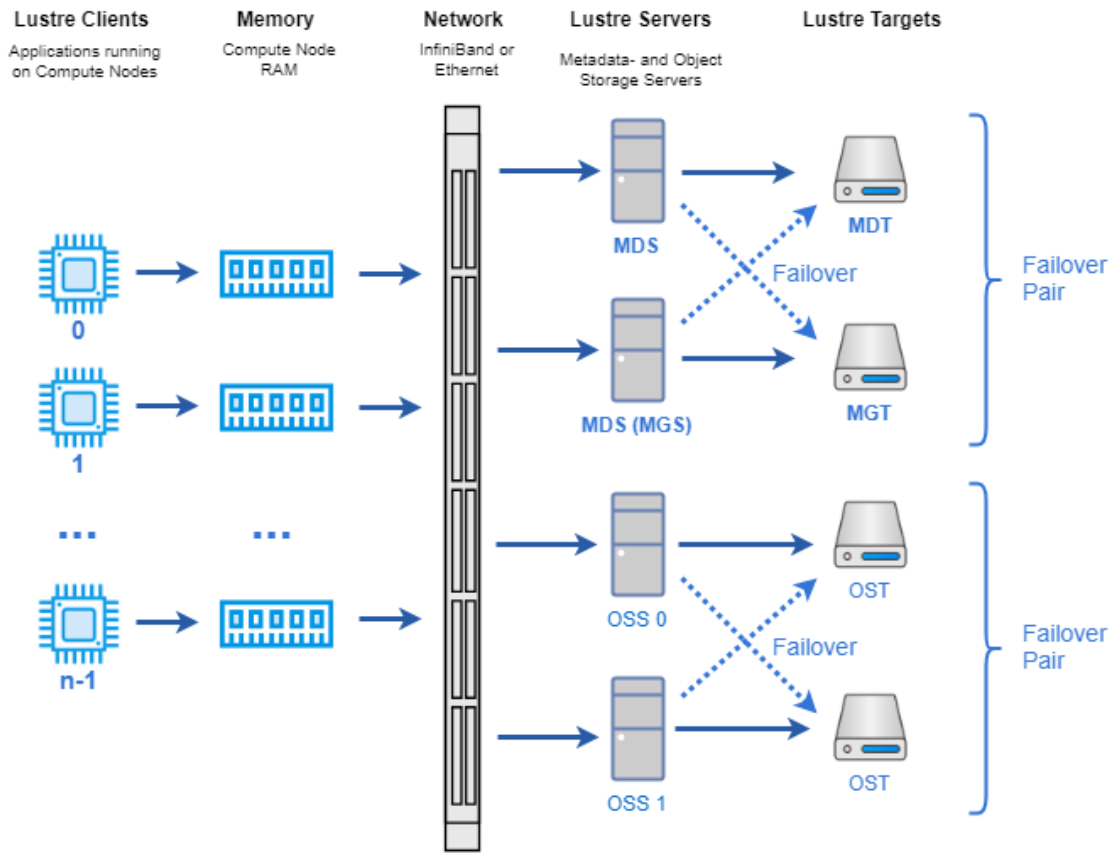

Figure 2. Lustre basic high availability configuration (Lustre Software Release 2.x Operations Manual, 2017)

The flow of data from application to disk is shown on Figure 2. The metadata handling is distributed onto two metadata servers that have different roles, one is the Metadata Server (MDS) with a Metadata Target (MDT) for storage, the other is a Metadata Management Server with a Management Target (MGT) for storage. Both servers can access each other's storage as a failover. We can also see the scalability points of the cluster. Each Object Storage Server may have multiple Object Storage Targets and new ones can be dynamically added. Optimally the OSTs will be evenly spread to the Object Storage Servers. The Object Storage Servers will be hosted in a failover configuration as well to provide high availability. The very minimum configuration could have one Metadata Server with combined Metadata Management Server and a single Metadata. To avoid single point of failure, in practice there would be at least one pair of MDSs so that in the case of a failure the other will be used as a backup automatically as depicted in Figure 2.

Consider the scenario where a Client (here Lustre Client) requests file access or creation on the DFS. The client first contacts the Metadata Storage Server (MDS) for the locations in the Object Storage Servers (OSS). In read operations, the data then flows from OSTs to Client memory in parallel (Lustre Software Release 2.x Operations Manual, 2017). This process is very similar to HDFS in its logic.

4.4 OrangeFS

OrangeFS distributes its metadata to ensure efficiency in computations. In HPC environments, an OrangeFS storage node typically has two services on it; the metadata service that handles the receiving and sending of file and directory information, and the data service which sends and receives the actual data of the objects on that specific node (OrangeFS Documentation, 2017). A storage node can be configured also to only provide one of these services, but usually both are enabled.

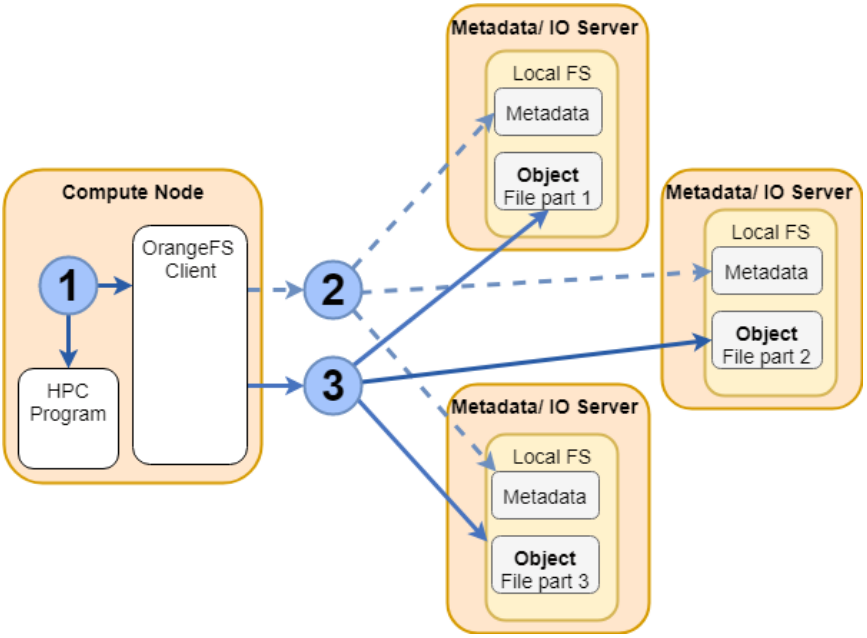

Figure 3. OrangeFS file retrieval process (OrangeFS Documentation, 2017)

The process of fetching a file and the rough architecture of an OrangeFS cluster is shown on Figure 3 as it is depicted in the official OrangeFS documentation (2017). Assume we have an HPC application that wants to retrieve a file from an OrangeFS cluster. On the machine that wants to do the request, there would also be an OrangeFS client application that can contact the DFS. First the HPC application makes the request for a file to the client. Next the client will look up the file's object locations by asking it from the Metadata/ IO Servers' metadata services. Once it gets the answers, the client will retrieve the file parts in parallel from the nodes. So, a client would have to contact many storage nodes just to ask for the metadata and figure out where the file parts are located.

OrangeFS handles files and directories as objects. Each instance of a file or directory contains two objects, the data object and the metadata object. Objects could also contain both data and metadata, if that is needed from them. The metadata of each individual file is stored in a randomly selected metadata server. The Object based file system supposedly provides better scalability in both capacity and performance (OrangeFS Documentation, 2017).

An object in this case may contain three things; A handle for reaching it, a bytestream of arbitrary length, generally used to store the file data, and thirdly the key/value pairs that allow us to store and retrieve metadata and attributes based on the key. Key/value pairs can be searched by index. Either the key/value pairs or the bytestream can be used, or in special cases both.

In OrangeFS the metadata contains usual ownership and access permission information, and as expected it also contains additional information related to the distributed nature of the data of OrangeFS. The user has power over how the data is distributed, and the metadata reflect on these configurations.

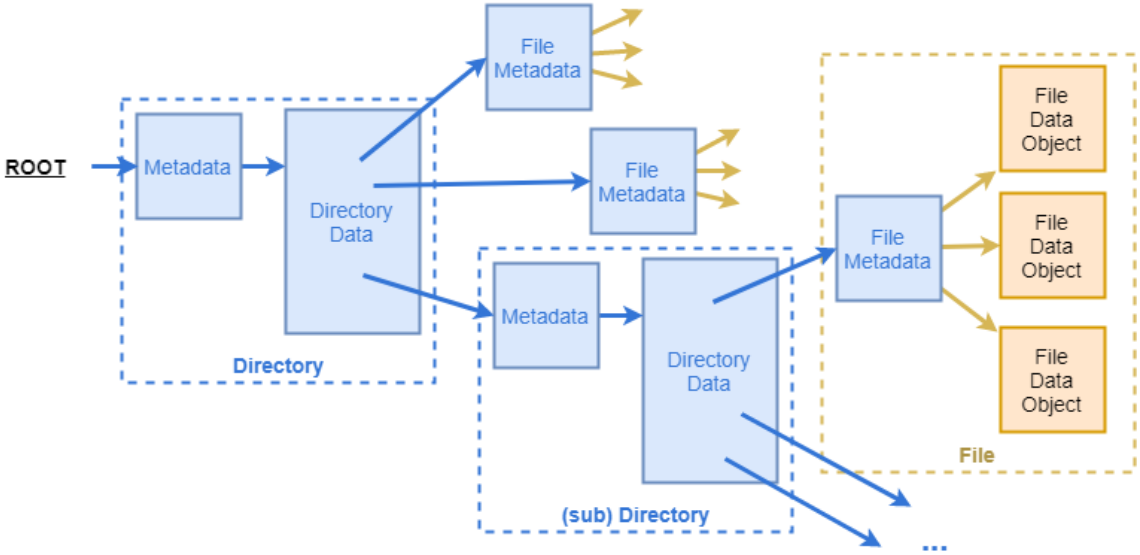

Figure 4. OrangeFS file system objects (OrangeFS Documentation, 2017)

The object based structure is visualized in figure 2. Every solid box is an object, and only the File Data Objects at the end are actual data. Everything else is metadata. The figure only shows the logical flow of metadata, not the physical flow of data between servers.

4.5 GlusterFS

GlusterFS is an exception among the example filesystems. It has a no-metadata server architecture, meaning it has no separate metadata servers to keep track of the metadata (GlusterFS Documentation, 2017). When a client makes a request in GlusterFS, it determines the correct server by an algorithm. According to Gluster the architecture enables linear scalability and reliability despite the lack of metadata. However, if file changes happen when a node is down in a GlusterFS cluster, and no replication is set, it may cause problems and require manual intervention (Vadiya & Deshpande, 2016). Donvito, Marzulli & Diacono (2014) also point out that Gluster has been suffering from some reliability issues. A so called split-brain issue may occur if the cluster experiences sudden outage during file operations. There two or more servers will have different versions of the same file, and they disagree on which is the most up-to-date. Solving this issue may require manual intervention. The next major version, GlusterFS 3, is aimed to solve these issues among others (GlusterFS Documentation, 2017).

The basic storage units in GlusterFS are called bricks. A brick is any designated local storage directory in a GlusterFS storage pool, basically a local file system (Davies & Orsaria, 2013). A logical collection of bricks is called a volume, and the volume is what is visible to the users of the file system (Donvito et al., 2014).

In GlusterFS data is stored by a hash algorithm. Data can be mirrored to have multiple copies, distributed across different nodes and files cut into stripes by a selectable pattern, of which there are seven to choose from (Chen & Liu, 2016).

5. Comparison of Architectures

We saw some radical differences in the way metadata was stored. Taking OrangeFS and HDFS as an example, OrangeFS distributes its metadata across many nodes, with the idea that this is more efficient when the metadata can be handled on-site next to the actual data it represents (Moore et al., 2011). The number of nodes containing metadata increases as the number of data storage nodes increase. HDFS on the other hand centralizes its metadata on a single NameNode and a possible backup node within a cluster instance. This might seem like a potential bottleneck as more than 50% of all file system accesses are on metadata (Brandt, Miller, Long, & Xue, 2003; Li, Xue, Shu, & Zheng, 2006). Both file systems seem to scale well despite the possible initial thought that HDFS's performance might be suffering from its centralized metadata storage. This could be attributed to HDFS applications not being very metadata intensive and perhaps the fact that HDFS clusters can be chained together (HDFS Architecture Guide, 2017). It is important to bear in mind that no actual data passes through the metadata servers in the example file systems. So even in a multi-process highly distributed platform a single metadata node might not get very high loads until the cluster grows very large.

The optimal choice may depend on the use case, and the only way to find the best file system for any use case is testing with correct data sets. Some tests have been done, for example in a 2014 study Donvito et al. tested HDFS, GlusterFS and a younger file system CephFS for their I/O performance. GlusterFS performed best on all tests with its no-metadata architecture. However, its reliability has caused issues, making it potentially unrealistic solution at least at the time of the study. The DFSs are constantly evolving and this includes the metadata solutions involved.

5.1 Common Issues and Solutions

There are some drawbacks included in any DFS architecture. Some issues are common in specific metadata architectures, and will be reviewed next along with possible solutions.

5.1.1 Centralized Metadata

Highly scalable applications with thousands of processes opening a file simultaneously have the potential to be a major bottleneck (Alam, El-Harake, Howard, Stringfellow, & Verzelloni, 2011). This is referred to as the Metadata Wall. This would be an issue only on larger systems. For example, Cloudera, a company that provides Hadoop and HDFS based solutions, has stated that one metadata master node with 64Gb ram can easily handle 100 DataNodes worth of metadata (Managing HDFS, 2017). Alam et al. (2011) also point out that due to the software limitations of the cluster DFSs and the separated metadata servers, the applications do not benefit much from improved hardware and lots of potential is lost. They mention that the overhead from accessing a file increases linearly the more nodes are involved in the process.

5.1.2 Directory Distribution

Another storage related issue is that directories may not scale across nodes in some distributed file systems (Yang et al., 2011). In some cases, this is due to the

implementation of the common directory subtree partitioning for metadata allocation (Brandt et al., 2003). In directory subtree partitioning, metadata is partitioned based on the directory tree structure so that a single directory tree is hosted on single metadata server. Bottlenecking occurs when a single directory, file or subdirectory comes under heavy load and the needed metadata is found only on one metadata server (2003).

Data mining and real-time monitoring solutions tend to write large numbers of small files during their processes in a short amount of time, resulting in very large directories with small files. One file system that used to suffer from this was the OrangeFS file system, which stored its directories as objects and they would not be striped across the cluster. A large enough cluster can create thousands of files in a single burst of less than a second (Patil, Gibson, Lang, & Polte, 2007). The ability to distribute the files in a directory would benefit the performance of the directory significantly in use cases where numerous small files (<1 Mb) are stored in single directories (Yang et al., 2011). New solutions have emerged for OrangeFS and other similar file systems, providing a small file splitting process (Patil et al., 2007; Yang et al., 2011). In the case of OrangeFS the proposed solution is to implement an array of directory metadata handles, which may point to directories in different servers (Yang et al., 2011). File names are hashed and a single directory's files spread according to it to achieve load balancing. The data throughput benefits are obvious in clusters larger than 8 nodes, single directory file creation in a 16-node cluster is three times as fast with the OrangeFS changes proposed by Yang et al., (2011). In HDFS the client does not enter files into the DFS but stores it locally until it has accumulated at least a data block size's worth of data (by default 64Mb). Otherwise small file read and writes could affect the network speed and congestion significantly (HDFS Architecture Guide, 2017). These blocks then are stored in parallel, removing the problem from HDFS.

5.1.3 Metadata Allocation in Metadata Server Clusters

A cluster of a hundred data nodes with separate metadata storage may still be managed with only one or two metadata servers (Shvachko et al., 2010; Sun Microsystems Inc., 2007). However, a cluster may grow so large that the metadata servers alone form a separate metadata server cluster (Li et al., 2006). In these cases, the way metadata is allocated between the metadata servers begin to have significance in the file system performance.

The directory subtree partitioning and pure hashing are two common, so called static solutions in metadata allocation (Li et al., 2006). Both have bottlenecks at high parallel access rates (Brandt et al., 2003). In pure hashing, the entire allocation of file metadata onto metadata servers is done by creating a hash code based on some attribute of the file, such as the filename which used to be the method in use in Lustre, for example. This provides a very balanced allocation of file metadata across metadata servers. A drawback according to Brandt et al. (2003) is that a directory hierarchy still need to be maintained and traversed to reach the files. They also add that if the filename is used as the hash, files with the same name will be put in the same location possibly resulting in an unbalanced distribution and a hotspot of traffic. Another problem mentioned to be included in both allocation methods is that adding new metadata servers require the readjusting of big amounts of metadata, which can easily cause significant downtime or temporary performance issues until the cluster has rebuilt its metadata structure (Brandt et al., 2003).

Brandt et al. (2003) have proposed an interesting hybrid method instead, which they call "Lazy Hybrid (LH) metadata management". The aim is to combine the best of both

hashing and subtree partitioning in an object-based storage system. It addresses all the mentioned problems with the use of hierarchical directories, hashing and lazy management of metadata relocation and access control lists (2003). File metadata is hashed by the full path, so that each hash is unique. The lazy update policies help cope with scaling the metadata cluster and name changes. Metadata may only be moved to the correct place when it is needed and is detected to not be in the expected location, for example, after a new metadata server was added in the cluster.

Another, newer metadata management method is called Dynamic Hashing (DH). Its principles resemble the Lazy Hybrid, but it introduces a method of adjusting the metadata distribution under changing loads by first identifying hot spots in the system and dynamically redistributing the metadata as needed (Li et al., 2006). Hashing is random, and the actual load the servers experience cannot be predicted in advance. In Dynamic Hashing the busy metadata servers redistribute their subtrees to less busy metadata servers (Li et al., 2006).

5.2 Summary of Reviewed Distributed File Systems

Table 1 summarizes the metadata design architectures of the reviewed DFSs and the limitations their design may bring. All of the reviewed file systems are parallel and asymmetric, GlusterFS is the only aggregation-based DFS among the reviewed DFSs, the rest are object-based storing their data as objects (GlusterFS Documentation, 2017; Thanh et al., 2008). The dedicated metadata storage of HDFS and Lustre bring a potential bottleneck in the system, but as the research showed, the other designs are not flawless either. All of the DFSs are still being developed, and many of the current limitations are promised to be solved in future releases.

Table 1. Overall summary of reviewed architectures and their limitations.

DFS	**Metadata Architecture**	**Metadata Storage**	**Design Limitations**
HDFS	Asymmetric, parallel, object-based	Dedicated server	Potential centralized metadata bottleneck in large clusters
Lustre	Asymmetric, parallel, object-based	Dedicated server	Potential centralized metadata bottleneck in large clusters
OrangeFS	Asymmetric, Parallel, object-based	Parallel on-site with related object data	No parallel file access Poor recovery from failures
GlusterFS	Asymmetric, Parallel, Aggregation-based	No distribution metadata needed	Poor file modification efficiency “Split-brain” file inconsistencies

6. Conclusions

After the basic concepts, such as file system, metadata and distributed file system were briefly introduced, the methods with which distributed file systems can handle metadata, was investigated by comparing four commonly used distributed file systems. Finally, conclusions were made based on the results of the comparison.

It is found that the existing distributed file systems can have significantly differing metadata management architectures. Metadata may be separated from the data itself, so that it is managed by a dedicated server or sub-cluster of servers within the cluster. Metadata may also be stored on-site with the represented data, so that reading and writing of metadata and actual data happens on the same storage node and the network use is reduced. Also “metadata-free” designs exist, where the entire allocation and look-up processes happen based on a pure hash algorithm and the need for maintaining actual metadata is minimal.

There is no clear winner between these metadata management methods. While a pure hash algorithm would theoretically be most efficient according to some research, in practice other designs may be more desirable due to better scalability, better management, more suitable interfacing or better overall support. The most suitable method of metadata management depends on the cluster size among other factors. All the reviewed distributed file systems are still being developed, and the metadata management is also subject to change.

The concept of distributed file systems is still evolving. Plenty of research on the metadata management was found, often the more recent ones improving on the concepts proposed in the previous ones. The documentation of the reviewed file systems pointed out that even the metadata handling has changed during their development in favour of more efficient methods. The fact that such radical differences in the architectures can be seen and that no architecture has fully surpassed the others yet, hints that there is no established standard on how a DFS should be constructed. One of the main principles of DFSs is that they must be dynamically scalable, so that it would solve the storage matter regardless of cluster size. This would hint that eventually some file system will surpass the others if they find a balance in scalability and performance. It could also be that a handful of DFSs will remain, best suited to their specific use cases.

As the reviewed DFSs are evolving, comparing the roadmap of each file system to see where they are moving in the future would be worthwhile. Only somewhat mature DFSs were used in this thesis, each of them having been in public use for years. It would also be worthwhile to compare the architectures of newer DFSs that have not reached public distribution yet. Another way to expand the research would be to compare other types of DFSs, such as ones using the peer-to-peer architecture which would not have fitted in the scope of this thesis, but would be a good target for future research due to its unconventionality. There is research that have compared the actual metadata handling efficiency of example file systems in terms of I/O performance, and were used as references in this thesis. However, as the research targeted older and mature DFSs, similar research could be done on the newer ones.

7. References

Alam, S. R., El-Harake, H. N., Howard, K., Stringfellow, N., & Verzelloni, F. (2011). *Parallel I/O and the metadata wall* ACM.

Brandt, S. A., Miller, E. L., Long, D. D., & Xue, L. (2003) Efficient metadata management in large distributed storage systems. *Mass Storage Systems and Technologies (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference On,* 290-298.

Buyya, R., Calheiros, R. N., & Dastjerdi, A. V. (2016). *Big data: Principles and paradigms*

Chen, W., & Liu, C. (2016). *Performance comparison on the heterogeneous file system in cloud storage systems* doi:10.1109/CIT.2016.90

Davies, A., & Orsaria, A. (2013). Scale out with GlusterFS. *Linux Journal, 2013*(235), 1.

GlusterFS documentation. (2017). Retrieved 14.8.2017, from http://gluster.readthedocs.io/en/latest/

HDFS architecture guide. (2017). Retrieved 8.8.2017, from https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

Li, W., Xue, W., Shu, J., & Zheng, W. (2006). *Dynamic hashing: Adaptive metadata management for petabyte-scale file systems*

Lustre software release 2.x operations manual. (2017). Retrieved 10.8.2017, from http://doc.lustre.org/lustre_manual.pdf

Managing HDFS. (2017). Retrieved 22.8.2017, from https://www.cloudera.com/documentation/enterprise/5-8-x/topics/admin_hdfs_config.html

Moore, M., Bonnie, D., Ligon, B., Marshall, M., Ligon, W., Mills, N., . . . Wilson, B. (2011). OrangeFS: Advancing PVFS. *FAST Poster Session,*

OrangeFS documentation. (2017). Retrieved 10.8.2017, from http://docs.orangefs.com/v_2_9/index.htm

Patil, S. V., Gibson, G. A., Lang, S., & Polte, M. (2007). *Giga : Scalable directories for shared file systems* ACM.

Riley, J. (2017). Understanding metadata - what is metadata, and what is it for? Retrieved 28.8.2017, from http://www.niso.org/apps/group_public/download.php/17446/Understanding%20Metadata.pdf

Sandberg, R. (1986). *The sun network file system: Design, implementation and experience*

Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010) The hadoop distributed file system. *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010,* doi:10.1109/MSST.2010.5496972

Singh, K., & Kaur, R. (2014) Hadoop: Addressing challenges of big data. *Souvenir of the 2014 IEEE International Advance Computing Conference, IACC 2014,* 686-689. doi:10.1109/IAdCC.2014.6779407

Sun Microsystems Inc. (2007). *LUSTRE™ FILE SYSTEM high-performance storage architecture and scalable cluster file system*

Thanh, T. D., Mohan, S., Choi, E., Kim, S., & Kim, P. (2008). In Kim, J Delen, D Jinsoo, P Ko, FK Na,YJ (Ed.), *A taxonomy and survey on distributed file systems* doi:10.1109/NCM.2008.162

Tmpfs. (2017). Retrieved 1.9.2017, from https://wiki.archlinux.org/index.php/tmpfs

Vadiya, M., & Deshpande, S. (2016). *Comparative analysis of various distributed file systems and performance evaluation using map reduce implementation* doi:10.1109/ICRAIE.2016.7939473

Verma, A., Mansuri, A. H., & Jain, N. (2016). *Big data management processing with hadoop MapReduce and spark technology: A comparison* doi:10.1109/CDAN.2016.7570891

Volume and file structure of CDROM for information interchange. (1987). from https://global.ihs.com/doc_detail.cfm?gid=EMOSCBAAAAAAAAAA&input_doc_number=ECMA ECMA 119

Weets, J-F., Kakhani, M. K., & Kumar, A. (2015) Limitations and challenges of HDFS and MapReduce. *Proceedings of the 2015 International Conference on Green Computing and Internet of Things, ICGCIoT 2015,* 545-549. doi:10.1109/ICGCIoT.2015.7380524

Wirzenius, L., Oja, J., Stafford, S., & Weeks, A. (2003). *Linux system administrators guide* (0.9th ed.)

Yang, S., Ligon III, W. B., & Quarles, E. C. (2011). Scalable distributed directory implementation on orange file system. *Proc.IEEE Intl.Wrkshp.Storage Network Architecture and Parallel I/Os (SNAPI),*