Advances in Science, Technology and Engineering Systems Journal Vol. 2, No. 4, 78-91 (2017)

www.astesj.com

ASTESJ

ISSN: 2415-6698

# The main characteristics of five distributed file systems required for big data: A comparative study

Akram Elomari*, Larbi Hassouni, Abderrahim Maizate

*RITM-ESTC / CED-ENSEM, University Hassan II, ZIP Code 8012, Morocco*

ARTICLE INFO

*Article history:*
*Received: 31 March, 2017*
*Accepted: 13 June, 2017*
*Online: 26 June, 2017*

*Keywords:*
*Big Data*
*Data Storage*
*DFS*
*HDFS*
*GFS*
*AFS*
*GPFS*
*Blobseer*
*BLOB*
*Data Stripping*
*Tiered storage*

ABSTRACT

*These last years, the amount of data generated by information systems has exploded. It is not only the quantities of information that are now estimated in Exabyte, but also the variety of these data which is more and more structurally heterogeneous and the velocity of generation of these data which can be compared in many cases to endless flows. Now days, Big Data science offers many opportunities to analyze and explore these quantities of data. Therefore, we can collect and parse data, make many distributed operations, aggregate results, make reports and synthesis. To allow all these operations, Big Data Science relies on the use of "Distributed File Systems (DFS)" technologies to store data more efficiently. Distributed File Systems were designed to address a set of technological challenges like consistency and availability of data, scalability of environments, competitive access to data or even more the cost of their maintenance and extension. In this paper, we attempt to highlight some of these systems. Some are proprietary such as Google GFS and IBM GPFS, and others are open source such as HDFS, Blobseer and AFS. Our goal is to make a comparative analysis of the main technological bricks that often form the backbone of any DFS system.*

## 1. Introduction

Todays, the amount of data generated during a single day may exceed the amount of information contained in all printed materials all over the world. This quantity far exceeds what scientists have imagined there are just a few decades. Internet Data Center (IDC) estimated that between 2005 and 2020, the digital universe will be multiplied by a factor of 300, this means that we will pass from 130 Exabyte to 40,000 Exabyte, which is the equivalent of 40 billion gigabytes (more than 5,200 gigabytes for each man, woman and child in 2020) [1].

Therefore, the variety and the complexity of this deluge of data, which is often unstructured, are revolutionizing the methods of data management and exploitation of the large quantity of information they convey [2,3].

Traditional data processing technologies have rapidly reached their limits and are being replaced by new systems which allow big data storage and analysis, taking on consideration what is currently known as the four V: Volume (to handle the huge amount of generated data), Velocity (to store, analyze and retrieve huge dataset as quickly as possible), Variety (to process mostly unstructured data, from multiple sources), and Value (to ask the right questions to generate maximum value) [4].

The typical schema of Big Data architecture (e.g. MapReduce) requires partitioning and distributing the processing across as many resources as possible. Otherwise many issues relative to the quantity of processed data can emerge like:

- Big data are slow to move over any network,
- Scaling up vertically (more memory, more powerful hardware) has limitations.
- A single hard drive cannot handle the size of big data.
- Failures in computing devices are inevitable

Move the “processing” into data instead of the opposite can become an obligation rather than a choice. Cloud platforms for

*Corresponding Author: Akram Elomari, RITM-ESTC / CED-ENSEM, University Hassan II, ZIP Code 8012, Morocco | Email: akramelomari@yahoo.fr

example, seem to offer countless benefits to such architecture, among the most important between those advantages is the scalability of the infrastructure that is managed by a fully outsourced service [5].

Distributed storage systems take also the same orientation.

Although the traditional systems such as centralized network-based storage systems (client-server) or the traditional distributed systems such as NFS, managed to meet the requirements of performance, reliability and safety of the data until a certain limit, they are no longer able to respond to the new requirements in terms of volume of data, high performance, and evolution capacities. And besides their constraints of cost, a variety of technical constraints are also added, such as data replication, continuity of services etc… [6,7].

In this paper, we try to discuss a set of the main characteristics of technologies used in the market and we think they are the most relevant and representative of the state of the art in the field of distributed storage. In section II, we start by explaining what Distributed File System (DFS) is. In section III, we discuss some architecture of some DFS systems while presenting the strengths and weaknesses of each of them. In section IV, we present the logic of storage as Blob. In section V, we discuss the technique of data stripping. In section VI, we discuss the issues of concurrency and some technologies used in this field. In section VII we present the tiered storage. We conclude this paper by a benchmark table of five major systems on the market: Andrew File System (AFS), Google File System (GFS), Blobseer, Hadoop Distributed File System (HDFS) and General Parallel File System (GPFS). The comparison focuses on a set of characteristics discussed and explained throughout this paper.

More specifically, our main objective in this paper is to contribute to determine the main characteristics that a Distributed File System must integrate to respond to the multiple requirements of a BIG DATA ecosystem. This study will allow us to well target the part on which we are going to conduct our research to improve the performance of a DFS.

## 2. What is “Distributed File system (DFS)”

A distributed file system (DFS) is a system that allows multiple users to access, through the network, a file structure residing on one or more remote machines (File Servers) using a similar semantics to that used to access the local file system. It is a client / server architecture where data is distributed across multiple storage spaces, often called nodes. These nodes consist of a single or a small number of physical storage disks.

The nodes generally consist of basic equipment, configured to just provide storage services. As such, the material can be relatively inexpensive.

The disk of each machine may be divided into several segments, and each segment is stored repeatedly (often three times) on different storage spaces, each copy of each segment is a replica.

As the material used is generally inexpensive and by large quantities, failures become inevitable. However, these systems are designed to be tolerant to failure by using the replication technique which makes the loss of one node an event "of low emergency and impact" as the data is always recoverable, often automatically, without any performance degradation.

The architecture of a distributed storage system varies depending on the technological choices driven by the use case. Nevertheless, it must generally observe some basic rules, which are required for the survival of such ecosystem and which can be summarized in the following points [8]:

- Access transparency: The remote file systems are exposed on the client machine like any local file system.
- Localization transparency: The client has no indication -by the file name- about the location of the file space neither if it is a local or remote space file.
- Concurrent access transparency: The file system state is the same for all the clients. This means that if a process is modifying a file, all other processes on the same system or remote systems that access the files see the changes in a consistent way.
- Failure Transparency: Client programs should not be affected by any loss of any node or a server.
- Heterogeneity: The File service needs to be supported by different hardware platforms and operating systems.
- Scalability: The file system should work in small environments (one to a dozen machines) as well as in large environments (hundreds or even tens of thousands of systems).
- Replication transparency: To support scalability, files must be replicated on multiple servers; transparently to clients (the system is on charge to create and maintain a designed number of replicas automatically).
- Migration transparency: any file movement in the system for management purposes should be transparent to the clients.
- Support fine-grained distribution of data: To optimize performance, the individual objects need to be located near the processes that use them.
- Tolerance for network partitioning: The file system should be tolerant to the fact that the entire network or certain segments of it may be unavailable during certain periods.

In this paper, we compare five distributed file systems: AFS, GFS, Blobseer, HDFS and GPFS. The choice to compare only those specific systems, despite of the fact that the market includes dozens of technologies, is particularly led by two reasons:

1. Our main objective is to study by focusing on the main features of the most Data File Systems required for a Big Data context. It is technically difficult to study all systems in the market in order to know their technical specifications, especially as lots of them are proprietary and closed systems. Even more, the techniques are similar in several cases and are comparable to those of the five we compare in this paper. The best known and not included in our paper because of that are: Amazon S3 File System,

OCFS (Oracle Cluster File System), GFS2 (Red Hat), VMFS (Virtual Machine File System by VMware).

2. These five systems allowed us to make a clear idea about the state of the art of this domain, thanks to the following particularities:

- AFS (Andrew File System) is a system that can be considered as a bridge between conventional systems such as NFS and advanced distributed storage systems. His big advantage is that it is available on a wide range of platforms: AIX, Mac OS X, Darwin, HP-UX, Irix, Solaris, Linux, Microsoft Windows, FreeBSD, NetBSD and OpenBSD.
- GFS (Google File System) is a proprietary system used internally by Google, which is one of the leading innovating companies. Google aims to manage huge quantities of data because of its activities.
- Blobseer is an open source initiative, particularly driven by research as it is maintained by INRIA Rennes. Blobseer choices, especially in the area of concurrency, are very interesting as discussed hereafter.
- HDFS (Hadoop Distributed File System), which is a subproject of HADOOP, a very popular Big Data system, is considered as a reference in this domain. It is therefore interesting to review its mechanisms and compare them to the other DFS systems.
- GPFS (General Parallel File System) is a system developed by IBM, a global leader in the field of Big Data. IBM commercializes this system as a product.

By choosing those five systems, we tried to make sure to have an illustration of these specific initiatives:

- Open source initiatives (BlobSeer, AFS, HDFS),
- Academic initiatives (BlobSeer)
- Big Data leader's initiatives (IBM GPFS, Google GFS)
- Business market initiatives (IBM GPS)

We think that considering these four initiatives can help to make a clear idea about the main orientations in the market of distributed storage today.

## 3. DFS architectures

In the following, we study the architecture of each of the five systems in order to explore the mechanisms and architectural choices of each of them and thus understand the reasons which justify these choices.

### *3.1. Andrew File System (AFS) architecture*

A standard system that supports some characteristics of this kind of architecture is AFS.

AFS (or Open AFS currently) is a distributed file system originally developed by Carnegie Mellon University (as part of the Andrew Project. Originally named "Vice", AFS is named after Andrew Carnegie and Andrew Mellon). It is supported and developed as a product by Transarc Corporation (now IBM Pittsburgh Labs). It offers client-server architecture for federated file sharing and distribution of replicated read-only content [9].

AFS offers many improvements over traditional systems. In particular, it provides the independence of the storage from location, guarantees system scalability and transparent migration capabilities. AFS can be deployed on a wide range of heterogeneous systems, including UNIX, Linux, MacOS X and Microsoft Windows.

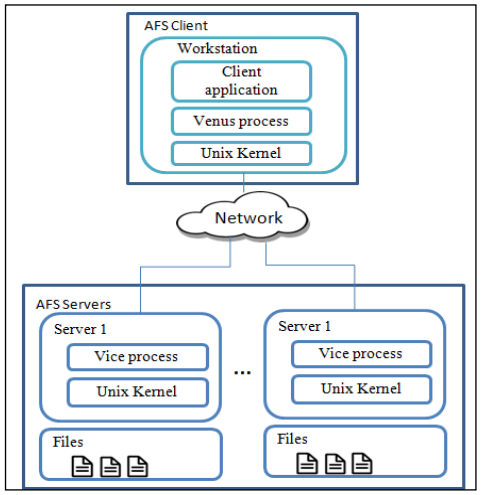


Figure 1 : AFS Design

As shown in Figure 1, the distribution of processes in AFS can be summarized as follows:

• A process called "Vice" is the backbone of the system; it is composed by a set of dedicated file servers and a complex LAN.

• A process called "Venus" runs on each client workstation; it mediates access to shared files. Venus gets the requested files from the vice process and keep them in the local cache of the client. Venus also emulates a "UNIX like" file system access semantic on the client station. "Vice" and "Venus" processes work in the back ground of the client workstation process, so the client sees a normal UNIX file system [10].

To better manage the transfer of files between servers and clients, AFS assumes the following hypothesis [11]:

- Concerned files remain unchanged for long periods;
- Those files will be updated only by their owners;
- A large local cache is enough to contain all the client files;
- Generally concerned files are of small size, less than 10 Kbytes;
- Read operations are more common than write operation;
- The sequential access is usually more common than random access;
- Most of the files are used by a single user, their owner;

- Once the file has been used, it will likely be used again in the near future.

These assumptions led AFS to adopt a fairly simple caching mechanism based on these two main elements:

- The whole content of directories and files are transferred from the server to the client (in AFS-3 by pieces of 64 kilobytes)
- Caching whole file: when the file is transferred to the client, it will be stored on the local client disk (client cache)

Using the client cache may actually be a good compromise to improve system performances, but it will only be effective if the assumptions that the AFS designers have adopted are respected. Otherwise, this massive use of the cache may compromise the data integrity.

### *3.2. Google File System (GFS) architecture*

Another interesting approach is that adopted by GFS, which does not use cache at all.

GFS is a distributed file system developed by Google for its own applications. Google GFS system (GFS cluster) consists of a single master and multiple Chunkservers (nodes) and can be accessed by multiple clients, as shown in Figure 2 [12].

Each of these nodes is typically a Linux machine running a server process at a user level. It is possible to run both a Chunkserver and a client on the same machine if its resources allow it.

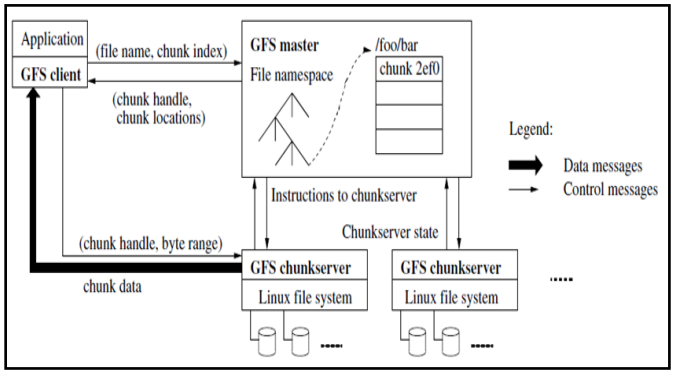


Figure 2 : GFS Design

The files to be stored are divided into pieces of fixed size called "chunks". Each "chunk" is identified by an immutable and unique “Chunk Handle” of 64 bits, assigned by the Master at its creation. The Chunkservers store chunks on local disks as Linux files, and manage to read or write a chunk using her Chunk Handle associated with a byte range.

The chunks are replicated on several Chunkservers. By default three replicas are stored, although users can designate a different number of replications if needed.

The "master" server maintains all metadata of the file system. This includes the namespace, access control information, the mapping from files to chunks and locations of existing chunks. It also controls the operations of the entire system, such as the selection and management of the master copy of a chunk (chunk lease), garbage collection (orphan chunks) and the migration of chunks between Chunkservers. The master communicates periodically with each Chunkserver to give instructions and collect its state.

The GFS client code uses the API of the file system. It communicates with the master and Chunkservers to read or write data. Clients interact with the master regarding transactions related to metadata, but all communications relating to the data themselves goes directly to Chunkservers.

Unlike AFS, neither the client nor the Chunkserver use a dedicated cache. Caches, according to Google, offer little benefit because most applications use large files or large work spaces which are too big to be cached. Not using the cache can simplify the work of the client and also the entire system by eliminating the cache coherence issues. The only exception to this rule is the metadata which can be cached on the client station. The Chunkservers does not need to use cache because the chunks are stored as local files and thus benefit from the "cache" of the Linux buffer that "cache" frequently accessed data in memory.

GFS was able to manage the failure possibility related to the cache coherence that can be noticed on AFS. But using a single master in the architecture of GFS is a real challenge; its involvement in read and write operations should absolutely be controlled so that it does not become a bottleneck. Google has tried to reduce the impact of this weak point by replicating the master on multiple copies called "shadows". These replicas are a backup of the master and better yet they can be accessed in read-only and so allowing access even when the master is down.

Google measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. All the machines are configured with dual 1.4 GHz processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch.

The test conditions was for 15 concurrent client accessing simultaneously N distinct files to read or write 1 GB of data

Read Average throughput: 90 MB/s

Write Average throughput: 34 MB/s

### *3.3. Blobseer architecture*

Blobseer is a project of KerData team, INRIA Rennes, Brittany, France. The main features of Blobseer are:

- Storage of data in BLOBs,

- Data segmentation,
- Management of distributed metadata
- Control of concurrency based on a versioning mechanism.

The data stored by Blobseer is wrapped in a level of abstraction that is a long sequence of bytes called BLOB (Binary Large Object) [13].

Blobseer has focused on the problems posed by the master in GFS and HDFS, but also on competitive access to data.

The Blobseer system consists of distributed processes (Figure 3), which communicate through remote procedure calls (RPC). A physical node can run one or more processes and can play several roles at the same time.

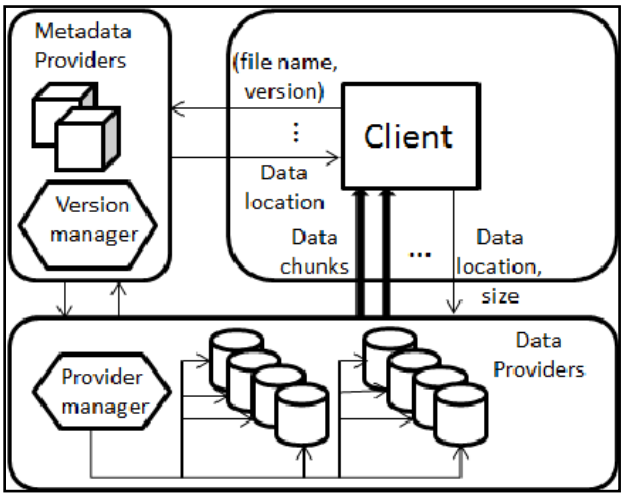


Figure 1 : Blobseer Design

The bricks of Blobseer are:

- Data providers: The data providers physically store the chunks. Each data provider is simply a local key-value store, which supports accesses to a particular chunk given a chunk ID. New data providers may dynamically join and leave the system.
- Provider manager: The provider manager keeps information about the available storage space and schedules the placement of newly generated chunks. It employs a configurable chunk distribution strategy to maximize the data distribution benefits with respect to the needs of the application. The default strategy implemented in Blobseer simply assigns new chunks to available data providers in a round-robin fashion.
- Metadata providers: The metadata providers physically store the metadata that allow identifying the chunks that make up a snapshot version of a particular BLOB. Blobseer employs a distributed metadata management organized as a Distributed Hash Table (DHT) to enhance concurrent access to metadata.
- Version manager: The version manager is in charge of assigning new snapshot version numbers to writers and to unveil these new snapshots to readers.
- The version manager is the key component of Blobseer, the only serialization point, but is designed to not involve in actual metadata and data Input/output. This approach keeps the version manager lightweight and minimizes synchronization.
- Clients: Blobseer exposes a client interface to make available its data-management service to high-level applications. When linked to Blobseer's client library, application can perform the following operations: CREATE a BLOB, READ, WRITE, and APPEND contiguous ranges of bytes on a specific BLOB.

Unlike Google GFS, Blobseer does not centralize access to metadata on a single machine, so that the risk of bottleneck situation of this type of node is eliminated. Also, this feature allows load balancing the workload across multiple nodes in parallel.

Since each BLOB can be stored as fragments over a large number of storage space providers, some additional metadata are needed to map sequences of BLOB. Although these additional metadata seem to be insignificant compared to the size of the data itself, on a large scale it represents a significant overhead. In those conditions, traditional approaches which centralize metadata management reach their limits.

Therefore, Blobseer argues for a distributed metadata management system, which brings several advantages:

- Scalability: A distributed metadata management system is potentially more scalable and open to concurrent accesses, This scalability can also cover the increase of the size of metadata.
- Data availability: Since metadata can be reproduced and distributed to multiple metadata providers, this avoids having a single centralized metadata server which then provides a single point of failure.

In addition, the implementation of the versioning mechanism via the «version manager» improves significantly the processing of concurrent access (as seen in Concurrent access paragraph).

A set of experiments was carried out on the Rennes cluster of the Grid'5000 platform [14,15]. The used nodes are interconnected through a 1 Gbps Ethernet network, each node being equipped with at least 4 GB of memory. The BlobSeer deployment consists of one version manager, one provider

manager, one node for the namespace manager. A BlobSeer chunk size of 32 MB was fixed, as previous evaluations of BlobSeer have shown this value enables the system to sustain a high-throughput for multiple concurrent data transfers. The test concerns the writing and reading of 2 GB and the Average throughput was measured:

Read Average throughput: 52 MB/s

Write Average throughput: 62 MB/s

The installation of a platform under Blobseer is of moderate difficulty. The preparation of the packages and their deployment is not very complicated but optimizations and tuning (snapshots, versioning, and concurrent accesses) require several tests.

### *3.4. Hadoop Distributed File System (HDFS)*

A standard system that supports some characteristics of this kind of architecture is AFS. Hadoop Distributed File System (HDFS) is a distributed file system component of the Hadoop ecosystem. The Apache Hadoop software library is a framework that allows distributing the processing of large data sets across clusters of computers using simple programming models[16].

HDFS is designed to run on commodity hardware, it is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS also provides high throughput access to application data and is suitable for applications that have large data sets. It relaxes a few POSIX requirements to enable streaming access to file system data[17].

As shown in figure 4, HDFS stores file system metadata and application data separately. Like other distributed file systems, HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols. The DataNodes in HDFS do not use data protection mechanisms such as RAID to make the data durable. Instead of that, the file content is replicated on multiple DataNodes for reliability. While ensuring data durability, this strategy has the added advantage that data transfer bandwidth is multiplied, and there are more opportunities for locating computation near the needed data [18].

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; which are the same size except the last one. The blocks of a file are replicated for fault tolerance. Files in HDFS are write-once and have strictly one writer at any time [19].

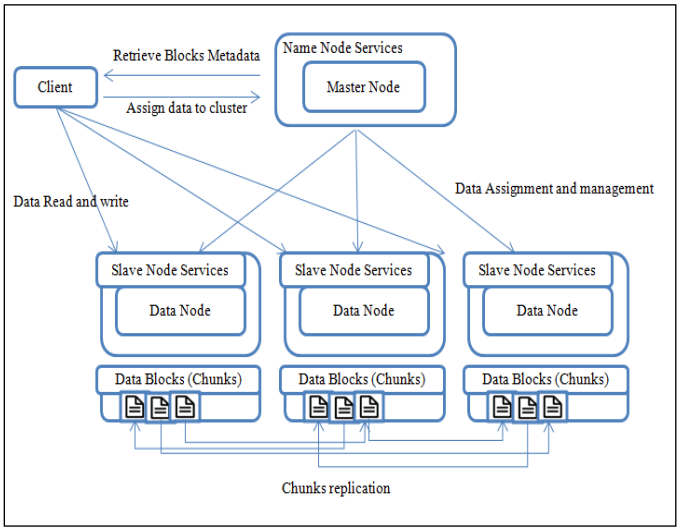


Figure 4: HDFS Design

An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client. When writing data, the client requests the NameNode to nominate a suite of three DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipeline fashion. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

Since the NameNode is unique in the cluster, saving a transaction to disk becomes a bottleneck for all other threads which have to wait until the synchronous operations initiated by one of them are complete [21]. In order to optimize this process the NameNode batches multiple transactions initiated by different clients. When one of the NameNodes threads initiates a flush-and-sync operation, all transactions batched at that time are committed together. Remaining threads only need to check that their transactions have been saved and do not need to initiate a flush-and-sync operation.

Regarding the performance, a basic test was performed on a test cluster composed by 8-nodes. The first 5 nodes of this Hadoop cluster provided both computation and storage resources (as Data Node servers). One node served as Job Tracker (Resource-Manager) and one node served as NameNode storage manager. Each node is running at 3.10 GHz CPU, 4GB RAM and a gigabit Ethernet. All nodes used Hadoop framework 2.4.0.

The test concerns the writing and reading of 10 GB of data and the average i/o rate was measured by TestDfsIO tool

“Write” Average i/o rate = 65 mb/s

“Read” Average i/o rate = 75 mb/s

The HDFS system remains simple enough to set up and manage, to add or to delete a node it needs the preparation of the post concerned and the change of some configuration files. Web interfaces make it possible to easily monitor the general condition of the nodes and even the distribution of the storage or the size of the chunks used.

Recompile the code on a particular machine can be more complicated but remains relatively simple for a system administrator.

*3.5. General Parallel File System (GPFS)*

A standard system that supports some characteristics of this kind of architecture is AFS. The General Parallel File System (GPFS) is a cluster developed by IBM which provides concurrent access to a single or set of file systems from multiple Storage Area Network (SAN) or network attached nodes [22].

GPFS is highly scalable and enables very high performances and availability thanks to a variety of features like data replication, policy based storage management, and multi-site operations. GPFS cluster can be deployed under AIX (Advanced IBM Unix), Linux or Windows server nodes. It can also be deployed on a mix of some or all those operating systems. In addition, multiple GPFS clusters can share data locally or across wide area network (WAN) connections [23].

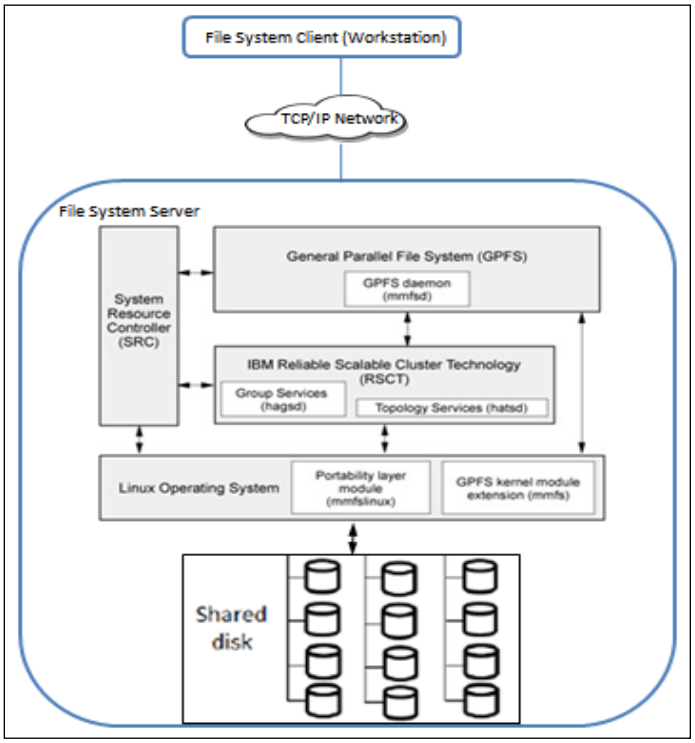


Figure 5: GPFS Design

GPFS uses the Network Shared Disk (NSD) protocol over any TCP/IP capable network fabric to transfer data to the client file system.

On the other side, GPFS server architecture is based on four modules as illustrated in Figure 5, which manage the shared disks

**System resource controller (src):** The main purpose of the System Resource Controller is to give to the system manager or a developer a set of commands and subroutines by which he can control and interact with the subsystems of the GPFS cluster.

**GPFS daemon (mmfsd)**: The GPFS daemon is charged of all I/O and buffers for GPFS, this include all read/write synchronous /asynchronous operations. To grant data consistency of the system, the daemon uses a token management system. On the other hand, the Daemon manages multi threads to ensure the priority to some critical processes and protect the whole system from lagging because of some intensive routines.

The daemons running on all the nodes of one cluster keep communicating with each other to insure that any configuration changes, recovery or parallel updates of the same data structures is shared between all of them.

**RSCT daemons:** GPFS uses Two RSCT daemons:

- The Group Service RSCT daemon (**hagsd**) ensures a distributed coordination and synchronization with the other subsystems.

- The Topology Service RSCT daemon (**hatsd**) insures providing other subsystems with network adapter status, node connectivity information, and a reliable messaging service.

**Linux Operating system :** Under Linux, GPFS need to run two modules:

- Portability layer module **(mmfslinux):** This module enables communication between Linux Kernel and GPFS kernel, based on hardware platform particularity and Linux distribution specifications.

- Kernel extension **(mmfs)**: which provides mechanisms to access a file system where data is physically stored from the client operating system transparently. In fact, GPFS appear to the client like any other local file system. When any application makes a call to any file system, this call is transmitted by the client Operating system into GPFS kernel extension. The kernel extension can respond to any file system call, by using the local resources if exists, or make a request to GPFS daemon if not.

GPFS have many specific features that make it very scalable and efficient:

- A GPFS cluster can integrate and optimize the use of different disk drives with different performances;

- GPFS use data striping across disks therefore the spreading of any processing over the cluster is possible;

- Metadata management is optimized to avoid the unnecessarily access to the server;

- GPFS uses caches on the client side to increase throughput of random reads;

- GPFS allows access to files from multiple programs on read and write mode;

- GPFS improves query languages such as Pig and Jaql by providing sequential access that enables fast sorts.

On the other hand, GPFS eliminates the risk of a single point of failure because the architecture is based on the following attributes:

- Distributed metadata;

- Replication of both metadata and data;

- Minimum number of nodes (quorum);

- The recovery and reassignment of failed node is automatic;

- GPFS provides a fully POSIX semantic;

- Workload isolation;

- Enhanced Security thanks to a native encryption, stronger cryptographic keys and more robust algorithms (NIST SP800-131a);

- Provides cluster-to-cluster replication over a wide area network.

All those features make GPFS a very scalable and high available system, but it does not seem to be designed for low cost hardware platforms unlike the GFS or Blobseer for example. Nevertheless, it remains proposing interesting mechanisms for data caching or parallel access to files.

## 4. Data Storage as Binary Large Object (blob)

The architecture of a distributed storage system can predict and improve the accessibility of files on storage spaces. It also enables the system design scalability and resilience to the risk of failures that amplify with the quality of equipment in use. However, among the main criteria that a distributed storage system must take into consideration is how files are stored on the disks.

In fact, we are talking about applications that process large quantities of data, distributed on a very large scale. To facilitate the management of data in such conditions, one approach is to organize these data as objects of considerable size. Such objects, called Binary Large Objects (BLOBs), consist of long sequences of bytes representing unstructured data and can provide the basis for a transparent data sharing of large-scale. A BLOB can usually reach sizes of up to 1 Tera Byte.

Using BLOBs offers two main advantages:

- The Scalability: Applications which deal with data sets that grow rapidly to easily reach around terabytes or more, can evolve more easily. In fact, maintaining a small set of huge BLOBs including billions of small items in the order of a few Kbytes is much easier than directly managing billions of small files of a few kilobytes. In this case, the simple mapping between the application data and file names can be a big problem compared to the case where the data are stored in the same BLOB and that only their offsets must be maintained.
- The Transparency: A data management system based on shared BLOBs, uniquely identifiable through ids, relieves application developers of the burden of codifying explicitly management and transfer of their locations. The system thus offers an intermediate layer that masks the complicity of access to data wherever it is stored physically [24].

## 5. Data striping

Data striping is a well-known technique for increasing the data access performance. Each stored object is divided into small pieces that are distributed across multiple machines over the storage system. Thus, requests for access to data may be distributed over multiple machines in parallel, allowing achieving high performances. Two factors must be considered in order to maximize the benefits of access to the distributed data:

- A configurable Strategy of distribution of chunks: Distribution strategy specifies where to store the chunks to achieve a predefined goal. For example, load balancing is one of the goals that such strategy can allow. By storing the chunks on different machines, we can parallelize the concurrent access to the same object and therefore improve performances. More complex scenarios are conceivable, for example optimizing access by geographical location or by the characteristics of storage machines (place the most requested chunks on the most powerful machines ...)[25,26]
- Dynamic configuration of the size of the chunks: The performance of distributed data processing is highly dependent on how the calculation is distributed and planned on the system. Indeed, if the chunks size is too small, applications must then retrieve the data to be processed from several chunks because of increasing probability of that the size of these data requires a high number of chunks. On the other hand, the use of too large chunks will complicate simultaneous access to data because of the increasing probability that two applications require access to two different data but both stored on the same chunk. A compromise will have to be made regarding the size of chunks to enable a balance between performance and efficiency of such system.

The majority of systems that use this type of architecture, such as Google GFS, HDFS or Blobseer use a chunk size of 64 MB that seems to be the most optimized for those two criteria.

## 6. Concurrency

Processing concurrency is very dependent on the nature of the desired data processing and the nature of data changes. It's clear that the Haystack system which manages Facebook pictures that do not changes during their lives [27], will be different from Google GFS or IBM GPFS which are intended to manage more dynamic data.

The "lock" is a known method to solve this type of problems, which is used by many DFS including GPFS.

The General Parallel File System (GPFS) propose a parallel access mechanism using block level locking based on a very sophisticated scalable token management system. This mechanism provides data consistency while allowing concurrent access to the files by multiple application nodes. A token server manages the lock acquisition and the lock revocation, and between these too operations only the system that has the lock can modify the file.

It is clear that in case of very large file, the lock operation can cause a considerable loss of time. Fortunately, IBM has developed a sophisticated mechanism that allows locking byte ranges instead of whole files/blocks (Byte Range Locking) [28]

GFS meanwhile, offers a relaxed consistency model that supports Google highly distributed applications, but is still relatively simple to implement. Practically all Google applications mutate files by appending rather than overwriting. The mutation operations on GFS are atomic. They are treated exclusively by the "master". The namespace locks guarantee its atomicity and accuracy. The status of a file region (a region of storage space which contains a part or the entire file) after a data transfer depends on the type of mutation, the success or failure of the mutation, and the existence or not of simultaneous mutations.

Table 1 summarizes the states of a file region after a transfer. A file region is "consistent" if all clients see the same data regardless of the replicas they are reading. A region is called "defined" after a change if it is consistent and clients will see all of what this mutation wrote.

When a mutation succeeds without simultaneous write interference, the affected region is defined (and coherent by consequence): All customers will see all what the mutation wrote.

Successful simultaneous mutations leave the region undefined but consistent: all clients see the same data, but the data may not reflect what any one mutation wrote, it will be composed of mixed fragments from multiple mutations. Failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times. GFS makes the difference subsequently between the defined regions and undefined regions.

Table 1 : File region state after mutation

| | Write | Record Append |
|---|---|---|
| Serial success | Defined | Defined interspersed with inconsistent |
| Concurrent successes | Consistent but undefined | |
| Failure | Inconsistent | |

On GFS, Data mutations may be a record write or a record append. A "record append" in GFS is different from a standard "append" in which the customer writes at the end of file. Indeed, a "record append" in GFS consists of writing a record in a block at least once even in the case of competitive changes, but at an offset that GFS itself chooses. The offset is returned to the client and marks the beginning of a defined region that contains the record.

After a sequence of successful mutations, the mutated region of the file is guaranteed to be "defined" and contains data written by the last mutation. GFS achieves this by applying chunk mutations in the same order on all replicas, but also using chunks version numbers to detect any replica that has become obsolete because it missed mutations. Obsolete replicas will never be used and will be destroyed by a garbage collector at the first opportunity.

Blobseer developed a more sophisticated technique, which theoretically gives much better results. The basic needs can be defined as following: the BLOB access interface must allow users to create a BLOB, read / write a sequence of bytes (of a known size starting from an offset) from or to the BLOB, and add a byte sequence of a certain size at the end of the BLOB.

However, given the requirements regarding competitive access to data, Blobseer developers claim that BLOB access interface should be able to:
- Manage Asynchronous operations;
- Have access to previous versions of the BLOB;
- Ensure the atomic generation of snapshots whenever the BLOB is updated.

Each of these points is covered by the following capabilities:

1. The explicit versioning: Applications that process large quantities of data must often manage the acquisition and processing of data in parallel. Versioning can be an effective solution to this situation. While the acquisition of data can lead to the generation of new snapshot of the BLOB, the data processing can continue quietly on its own snapshot that is immutable and therefore never leads to potential inconsistencies. This can be achieved by exposing data access interface based on versioning, which allows the user to directly express these workflow templates, without the need to explicitly manage synchronization.

2. Atomic snapshots generation: Snapshots can be used to protect the file system contents against any error by preserving at a point in time a version of the file system or a sub-tree of a file system called a fileset. In Blobseer, a snapshot of the blob is generated atomically each time the Blob is updated. Readers should not be able to access transiently inconsistent snapshots that are being generated. This greatly simplifies development of applications because it reduces the need for complex synchronization schemes at the application level.

The "snapshot" approach using versioning that Blobseer brings is an effective way to meet the main objectives of maximizing competitive access. Data and metadata are always created, but never overwritten. This will parallelize concurrency as much as possible, in terms of data and also metadata, in all possible combinations: simultaneous reads, simultaneous writes and concurrent reads and writes [29].

The disadvantage of such a mechanism based on snapshots, is that it can easily explode the storage space required to maintain the system. However, although each write or append generates a new version of the blob snapshot, only the differential updates from previous versions are physically stored. This eliminates unnecessary duplication of data and metadata and greatly optimizes storage space.

## 7. Tiered storage

Despite the high scalability of DFSs existing on the market and their ability to manage a very large number of nodes, they still dealing with managed nodes in a similar way.

Indeed, a node network in a DFS can technically be composed of several types of machines with heterogeneous storage units, managing these nodes similarly would often prevent DFS from taking advantage of the most powerful storage spaces or otherwise imposing many constraints on rudimentary storage spaces. A simple way to avoid this situation is to equip the DFS with a single type of node, therefore the management will be linear and the performance will not be impacted by the identity of the storage node. In this case the DFS is indifferent to the I/O characteristics of each node and will have to keep the same category of devices even if the technology is outdated (the case of the HDD disks), otherwise pro-actively opt for advanced technologies (the SSD for example) and undergo costs of maintenance and evolution.

Another way to address this problem is to allow DFSs to manage different device categories while equipping them with technology that enables them to intelligently manage storage policies on heterogeneous storage resources.

The "tiered storage" allows to create groups of "devices" (tiers) that have the same I / O characteristics and to manage the distribution of the storage on these groups according to the degree of solicitation of data.

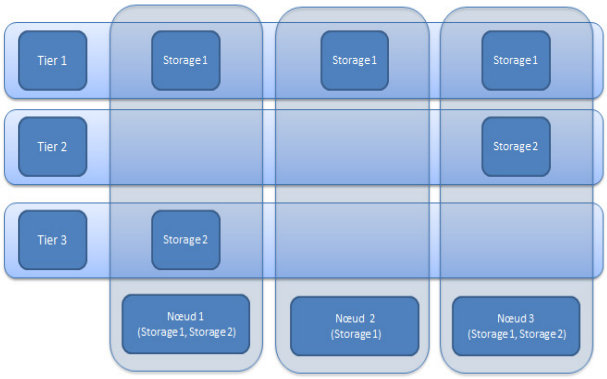


Figure 6: Tiered storage concept

Hadoop, since version 2.3.0, had introduced a major evolution that allowed the management of heterogeneous storage spaces; by using this option combined with a storage policy management API, the user can specify on which storage type this data should be stored.

Other works on Hadoop has made it possible to automate the choice of the storage space for specific data, for example based on the temperature of the data (hot data for the very demanded and cold data for those less solicited for example)[30] or even improve the architecture of HDFS as has been proposed by hatS [31] which logically groups all storage devices of the same type across the nodes into an associated “tier.” Or yet by TS-Hadoop [32] which utilizes tiered storage infrastructure, besides HDFS, to improve map reduce operations. TS-Hadoop automatically distinguish hot and cold data based on current workload, and move hot data into a specific shared disk (hcache) and cold data into HDFS respectively, so that the hot data in HCache could be processed efficiently.

The same concept is assured by other DFS like GPFS by "Spectrum Scale ILM toolkit" which allows the management of groups of storage spaces but also to automate the management of the files within these spaces. It allows to create hierarchized and optimized storage sets by grouping, in separate storage pools, discs that have close performances, similar budget characteristics or even hosted in the same physical location. Thereafter, a storage strategy tells the system what rules should be followed when storing each file.

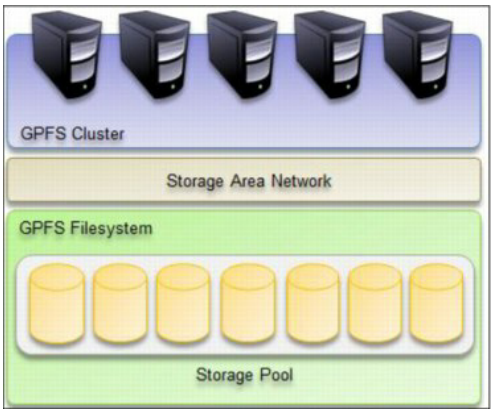


Figure 7: GPFS Storage spools as Tiered storage

The performance of a tired-storage compared to a traditional DFS can be very remarkable, allowing improvements up to 36% on the reading times on hatS for example. However, an automatic analysis must be associated to the architecture to allow automatic determination of the best storage location. This analysis can be done at the time of data storing via a specific algorithm based on the information of storage areas, or by analyzing the situation of the response time and redistribute data according to the results of the analysis (log analysis for example)

## 8. DFS Benchmark

As we have detailed in this article, often there is no better or worse methods for technical or technological choices to be adopted to make the best of a DFS, but rather compromises that have to be managed to meet very specific objectives.

In Table 3, we compare the implementation of some key technologies that meet the requirements listed in the paragraph "What is a Distributed File system", and that can be summarized as follows:

- Data Storage Scalability: the system can be scalable natively on the data storage capability.
- Meta Data Storage Scalability: the system can be scalable natively on the Meta data storage capability.
- Fault tolerance: the system is fault tolerant transparently to the user.
- Data Access Concurrency: how the system manages competitive access to data.
- Meta Data Access Concurrency: how the system manages competitive access to Meta data.
- Snapshots: does the system keep snapshots of files to recover from errors or crashes.
- Versioning: does the system records versions of changed files and data.
- Data Striping: does the system uses data striping over his nodes.
- Storage as Blobs: does the system store data as blobs.
- Data replication: does the system automatically replicate data.
- Supported OS: which operating systems can be used by the DFS.
- Dedicated cash: does the system support the using of dedicated cash.

Analysis of the results of Table 3 leads to the following conclusions:

- The five systems are expandable in data storage. Thus they cover one of the principal issues that lead to the emergence of Distribute File System: the capacity to extend the system to absorb more volumes, transparently to the user.

- Only Blobseer and GPFS offers the extensibility of metadata management to overcome the bottleneck problem of the master machine which manage the access to metadata; while AFS architecture does not provide metadata supporting to access to the file, GFS and HDFS has not considered necessary to extend the metadata management feature. Google considers that having a single master vastly simplifies the design of GFS and enables the master to make sophisticated chunk placement and replication decisions using global knowledge.

- Except AFS, all studied systems are natively tolerant to crash, relying essentially on multiple replications of data.
- The competitive access to the data and metadata is an important point in all big data systems. All systems use locks to enable exclusive data mutation. To minimize the slowing effect caused by locks on the whole file, GPFS manage locks on specific areas of the file (Byte range locks). Nevertheless, the most innovative method is the use of versioning and snapshots by Blobseer to allow simultaneous changes without exclusivity.

- Except AFS, all systems are using the striping of data. As discussed earlier, this technique provides a higher input/output performance by "striping" blocks of data from individual files over multiple disks, and reading and writing these blocks in parallel way.

- Blobseer seems to be the only one among the systems studied that implements the storage on blobs technique, despite the apparent advantages of such technique.

- To allow a better scalability, a DFS system must support as much operating systems as possible. However, despite that, the studied technologies remain discorded on this point. While AFS, HDFS and GPFS supports multiple platforms, GFS and Blobseer run exclusively on Linux. This can be partly explained by the popularity of AFS, HDFS and GPFS which are used in many professional contexts.

- Use of dedicated cache is also a point of discord between studied systems, GFS and Blobseer are categorical and consider that the cache has no real benefits, but rather causes many consistency problems. AFS and GPFS use dedicated cache on both client computers and servers. HDFS seems to use dedicated cache only at client level.

Table 3: Comparative table of most important characteristics of distributed file storage

| | Data Scalability | Meta Data Scalability | Fault tolerance | Data access Concurrency | Meta Data access Concurrency | Snapshots | Versioning | Data Striping | Storage as Blobs | Supported OS | Dedicated cache |
|---|---|---|---|---|---|---|---|---|---|---|---|
| HDFS | YES | NO | Block Replication. Secondary Namenode. | Files have strictly one writer at any time | NO | YES | NO | YES (Data blocks of 64 MB) | NO | Linux and Windows are the supported , but BSD, Mac OS/X, and Open Solaris are known to work | YES (Client) |
| Blobseer | YES | YES | Chunk Replication Meta data replication | YES | YES | YES | YES | 64 MB Chunks | YES | LINUX | NO |
| GFS by Google | YES | NO | Fast Recovery. Chunk Replication. Master Replication. | Optimized for concurrent "appends" | Master shadows on read only | YES | YES | 64 MB Chunks | NO | LINUX | NO |
| AFS (OPEN FS) | YES | NO | NO | Byte-range file locking | NO | NO | NO | NO | NO | AIX, Mac OS X, Darwin, HP-UX, Irix, Solaris, Linux, Microsoft Windows, FreeBSD, NetBSD and OpenBSD | YES |
| GPFS IBM | YES | YES | Clustering features. Synchronous and asynchronous data replication. | Distributed byte range locking | Centralized management | YES | unknown | YES | NO | AIX, Red Hat, SUSE , Debian Linux distributions, Windows Server 2008 | YES by AFM technology |

## 9. Conclusion

In this paper, we made a comparative study of the main characteristics of five distributed file storage systems. Firstly, we introduced the general objective of this kind of systems and reviewed related technologies, such as architectures, Blob use, data striping and concurrent access. At the end, we provide a table (Table 3) whose each column's header is a main characteristic of a DFS system and each line's header corresponds to one of the five DFS systems compared. At the intersection of each row and column, we specify whether the characteristic is implemented by the system as well as the particularities of the implementation.

It is clear from this analysis that the major common concern of such systems is scalability. Those systems are designed to manage the amount of data that extends day after day. Centralized storage systems have many limitations and their maintenance is complicated and raises major concerns about cost. A DFS should therefore be extended with a minimum cost and effort.

Also data availability and fault tolerance remain among the major concerns of DFS. Many systems tend to use non expensive hardware for storage. Such condition will expose those systems to frequent or usual breakdowns. This issue is remedied by replication mechanisms, versioning, snapshots… that aim restoring the system state, often automatically, after a fault or total loss of any nodes.

To these mechanisms, data striping and lock mechanisms are added to manage and optimize concurrent access to the data. Systems that manage large files in large quantities need to have a developed parallel access. Locking an entire file to change a part of it can halt the access to this file for an indeterminate duration. It was therefore important to adopt solutions that will just lock the byte range concerned by the change, or even like what Blobseer implements, continue editing in a new version without blocking other clients who continue to use the current version transparently.

Working on multiples operating systems can bring big advantages to DFS. AFS is the one offering the largest variety of operating systems that can support its implementation, but as seen above AFS have some serious limitations. In perspective, we can think to improve AFS with some mechanisms of data striping and concurrency management that we think the most important features to add to this DFS.

Furthermore, saving data as BLOB combined with a mechanism of data striping and cache, which is already proposed by AFS, can ameliorate considerably the efficiency of such system and allow it to manage larger files.

## Conflict of Interest

The authors declare no conflict of interest.

## References


[1] John Gantz and David Reinsel, “THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East.” Tech. rep. Internet Data Center(IDC), 2012.

[2] Weili Kou, Xuejing Yang, Changxian Liang, Changbo Xie ,Shu Gan,”HDFS enabled storage and management of remote sensing data” 2nd IEEE International Conference on Computer and Communications (ICCC), Chengdu, China, 2016. https://doi.org/10.1109/CompComm.2016.7924669

[3] D. Chen , Y.Chen, B.N. Brownlow, “Real-Time or Near Real-Time Persisting Daily Healthcare Data Into HDFS and ElasticSearch Index Inside a Big Data Platform” IEEE Transactions on Industrial Informatics, 2017. https://doi.org/10.1109/TII.2016.2645606

[4] Yanish Pradhananga, Shridevi Karande, Chandraprakash Karande, “High Performance Analytics of Bigdata with Dynamic and Optimized Hadoop Cluster” International Conference on Advanced Communication Control and Computing Technologies (ICACCCT),2016. https://doi.org/10.1109/ICACCCT.2016.7831733

[5] Richard J Self, “Governance Strategies for the Cloud, Big Data and other Technologies in Education” IEEE/ACM 7th International Conference on Utility and Cloud Computing, 2014. https://doi.org/10.1109/UCC.2014.101

[6] Purva Grover, Rahul Johari, “BCD: BigData,Cloud Computing and Distributed Computing” Global Conference on Communication Technologies(GCCT), 2015. https://doi.org/10.1109/GCCT.2015.7342768

[7] T. L. S. R. Krishna, T. Ragunathan and S. K. Battula, “Improving performance of a distributed file system using a speculative semantics-based algorithm” Tsinghua Science and Technology, vol. 20, no. 6, pp. 583-593, 2015. https://doi.org/10.1109/TST.2015.7349930

[8] Paul Krzyzanowski, “Distributed File Systems Design” Rutgers University, 2012.

[9] R. Tobbicke, "Distributed file systems: focus on Andrew File System / Distributed File Service (AFS/DFS)," Proceedings Thirteenth IEEE Symposium on Mass Storage Systems. Toward Distributed Storage and Data Management Systems, Annecy, 1994. https://doi.org/10.1109/MASS.1994.373021

[10] Monali Mavani, “Comparative Analysis of Andrew Files System and Hadoop Distributed File System” LNSE (Vol.1(2): 122-125 , 2013.

[11] Stefan Leue, “Distributed Systems” tele Research Group for Computer Networks and Telecommunications Albert-Ludwigs-University of Freiburg, 2001.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung Google*, “The Google File System” SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles,2003.

[13] T. L. S. R. Krishna and T. Ragunathan, "A novel technique for improving the performance of read operations in BlobSeer Distributed File System," 2014 Conference on IT in Business, Industry and Government (CSIBIG), Indore, 2014. https://doi.org/10.1109/CSIBIG.2014.7056982

[14] D.Santhoshi, V.Teja, T.Tejaswini Singh, K.Shyam Prasad, “Supplanting HDFS with BSFS” International Journal of Advanced Research in Computer Science Volume 5, No. 4, 2014.

[15] Alexandra Carpen-Amarie, “BlobSeer as a data-storage facility for clouds :self-Adaptation, integration, evaluation.” Ph.D Thesis ENS CACHAN – BRETAGNE,2011.

[16] M. Sogodekar, S. Pandey, I. Tupkari and A. Manekar, "Big data analytics: hadoop and tools," 2016 IEEE Bombay Section Symposium (IBSS), Baramati, India, 2016. https://doi.org/10.1109/IBSS.2016.7940204

[17] K. Qu, L. Meng and Y. Yang, ”A dynamic replica strategy based on Markov model for hadoop distributed file system (HDFS)” 4th International Conference on Cloud Computing and Intelligence Systems (CCIS), Beijing, 2016. https://doi.org/10.1109/CCIS.2016.7790280

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler Yahoo!, “The Hadoop Distributed File System” MSST '10 IEEE 26th Symposium on Mass Storage Systems and Technologies, 2010. https://doi.org/10.1109/MSST.2010.5496972

[19] C. B. VishnuVardhan and P. K. Baruah, "Improving the performance of heterogeneous Hadoop cluster," 2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC), Waknaghat, 2016. https://doi.org/10.1109/PDGC.2016.7913150

[20] Dhruba Borthakur, “HDFS Architecture Guide” The Apache Software Foundation, 2008.

[21] Passent M ElKafrawy, Amr M Sauber, Mohamed M Hafez, "HDFSX: Big data Distributed File System with small files support." 12th International Computer Engineering Conference (ICENCO), 2016. https://doi.org/10.1109/ICENCO.2016.7856457

[22] A. C. Azagury , R. Haas, D. Hildebrand, "GPFS-based implementation of a hyperconverged system for software defined infrastructure" IBM Journal of Research and Development, vol. 58, no. 2/3, pp. 6:1-6:12, 2014. https://doi.org/10.1147/JRD.2014.2303321

[23] Kuo-Yang Cheng, Hui-Shan Chen and Chia-Yen Liu, "Performance evaluation of Gfarm and GPFS-WAN in Data Grid environment," IET International Conference on Frontier Computing. Theory, Technologies and Applications, Taichung, 2010. https://doi.org/10.1049/cp.2010.0530

[24] Bogdan Nicolae, Gabriel Antoniu, Luc Boug_e, Diana Moise, Alexandra, Carpen-Amarie, "BlobSeer: Next Generation Data Management for Large Scale Infrastructures" Journal of Parallel and Distributed Computing, Elsevier, 2011. http://doi.org/10.1016/j.jpdc.2010.08.004

[25] Mariam Malak Fahmy,Iman Elghandour, Magdy Nagi, "CoS-HDFS: Co-Locating Geo-Distributed Spatial Data in Hadoop Distributed File System" IEEE/ACM 3rd International Conference on Big Data Computing Applications and Technologies (BDCAT), 2016 . http://doi.org/10.1145/3006299.3006314

[26] Cong Liao, Anna Squicciarini, Dan Lin. LAST-HDFS, "Location-Aware Storage Technique for Hadoop Distributed File System." IEEE 9th International Conference on Cloud Computing. 2016. https://doi.org/10.1109/CLOUD.2016.0093

[27] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel, Facebook Inc,, "Finding a needle in Haystack: Facebook's photo storage" OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation, 2010.

[28] Scott Fadden, "An Introduction to GPFS Version 3.5 Technologies that enable the management of big data" IBM Corporation, 2012.

[29] Bogdan Nicolae,Diana Moise, Gabriel Antoniu,Luc Boug´e, Matthieu Dorier, "BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications" Research Report RR-7140, INRIA , 2010. https://doi.org/10.1109/IPDPS.2010.5470433

[30] Rohith Subramanyam, "HDFS Heterogeneous Storage Resource Management based on Data Temperature" International Conference on Cloud and Autonomic Computing, 2015. https://doi.org/10.1109/ICCAC.2015.33

[31] Krish K.R., Ali Anwar, Ali R. Butt. "hatS, A Heterogeneity-Aware Tiered Storage for Hadoop" 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014. https://doi.org/10.1109/CCGrid.2014.51

[32] Zhanye Wang, Jing Li, Tao Xu, Yu Gu, Dongsheng Wang. TS-Hadoop, "Handling Access Skew in MapReduce by Using Tiered Storage Infrastructure" International Conference on Information and Communication Technology Convergence, 2014. https://doi.org/10.1109/ICTC.2014.6983331