Analysis of Six Distributed File Systems

Benjamin Depardon, Gaël Le Mahec, Cyril Séguin

Analysis of Six Distributed File Systems

Benjamin Depardon
benjamin.depardon@sysfera.com
SysFera

Cyril Séguin
cyril.seguin@u-picardie.fr
Laboratoire MIS, Université de Picardie Jules Verne

Gaël Le Mahec
gael.le.mahec@u-picardie.fr
Laboratoire MIS, Université de Picardie Jules Verne

February 15, 2013

Contents

1 Definitions and related work **2**
- 1.1 Distributed file system . . . 2
- 1.2 Scalable architecture . . . 2
- 1.3 Transparency . . . 3
  - 1.3.1 Naming . . . 3
  - 1.3.2 Data access Interfaces . . . 3
  - 1.3.3 Caching . . . 4
  - 1.3.4 Fault detection . . . 4
- 1.4 Fault tolerance . . . 4
  - 1.4.1 Replication and placement policy . . . 4
  - 1.4.2 Synchronisation . . . 5
  - 1.4.3 Cache consistency . . . 5
  - 1.4.4 Load balancing . . . 5
- 1.5 Related work . . . 6

2 Introduction of distributed file systems surveyed **7**
- 2.1 HDFS . . . 7
  - 2.1.1 Architecture . . . 7
  - 2.1.2 Naming . . . 7
  - 2.1.3 API and client access . . . 7
  - 2.1.4 Cache consistency . . . 8
  - 2.1.5 Replication and synchronisation . . . 8
  - 2.1.6 Load balancing . . . 8
  - 2.1.7 Fault detection . . . 8
- 2.2 MooseFS . . . 9
  - 2.2.1 Architecture . . . 9
  - 2.2.2 Naming . . . 9
  - 2.2.3 API and client access . . . 9
  - 2.2.4 Replication and synchronisation . . . 9
  - 2.2.5 Load balancing . . . 9
  - 2.2.6 Fault detection . . . 9
- 2.3 iRODS . . . 10
  - 2.3.1 Architecture . . . 10
  - 2.3.2 Naming . . . 10
  - 2.3.3 API and client access . . . 10
  - 2.3.4 Cache consistency . . . 10
  - 2.3.5 Replication and synchronisation . . . 10
  - 2.3.6 Load balancing . . . 10
  - 2.3.7 Fault detection . . . 11
- 2.4 Ceph . . . 11

2.4.1 Architecture . . . 11
2.4.2 Naming, API and client access . . . 11
2.4.3 Cache consistency . . . 11
2.4.4 Replication and synchronisation . . . 11
2.4.5 Load balancing . . . 12
2.4.6 Fault detection . . . 12
2.5 GlusterFS . . . 12
2.5.1 Architecture . . . 12
2.5.2 Naming . . . 12
2.5.3 API and client access . . . 13
2.5.4 Cache consistency . . . 13
2.5.5 Replication and synchronisation . . . 13
2.5.6 Load balancing . . . 13
2.5.7 Fault detection . . . 13
2.6 Lustre . . . 13
2.6.1 Architecture . . . 13
2.6.2 Naming . . . 14
2.6.3 API and client access . . . 14
2.6.4 Cache consistency . . . 14
2.6.5 Replication and synchronisation . . . 14
2.6.6 Load balancing . . . 14
2.6.7 Fault detection . . . 14

3 Analysis and Comparison **15**
3.1 Scalability . . . 15
3.1.1 Discussion about architecture . . . 15
3.1.2 Small or big files? . . . 16
3.2 Transparency . . . 16
3.2.1 File access & operations transparency . . . 16
3.2.2 Fault detection . . . 17
3.2.3 System access . . . 17
3.3 Fault tolerance . . . 17
3.3.1 System availability . . . 17
3.3.2 Data availability & data synchronisation . . . 18
3.3.3 Load balancing & cache consistency . . . 18
3.3.4 Test on data servers . . . 19

4 Tests achieved on the DFSs **20**
4.1 Setting up the DFSs studied on grid5000 platform . . . 20
4.1.1 HDFS . . . 20
4.1.2 MooseFS . . . 21
4.1.3 iRODS . . . 24
4.1.4 Ceph . . . 24
4.1.5 GlusterFS . . . 26
4.1.6 Lustre . . . 26
4.2 System accessibility . . . 28
4.2.1 HDFS . . . 28
4.2.2 MooseFS . . . 28
4.2.3 iRODS . . . 28
4.2.4 Ceph . . . 29
4.2.5 GlusterFS . . . 29
4.3 System availability . . . 29

4.3.1 HDFS 29
4.3.2 MooseFS 30
4.3.3 iRODS 31
4.3.4 Ceph 32
4.3.5 GlusterFS 34
4.3.6 Lustre 35
4.4 System performance 36

5 Conclusion **37**

Abstract

A wide variety of applications, particularly in High Performance Computing, relies on distributed environments to process and analyse large amounts of data. New infrastructures have emerged to support the execution of such computations, most of them involve distributed computing, parallelizing the computing process among the nodes of a large distributed computing platform. As the amount of data increases, the need to provide efficient, easy to use and reliable storage solutions has become one of the main issue for scientific computing. A well-tried solution to this issue is the use of Distributed File Systems (DFSs). In this paper we present and compare six modern DFSs that are today widely used to deal with the problem of storage capacities and data access, federating the resource of distributed platform.

Introduction

A wide variety of applications, such as probabilistic analysis, weather forecasting and aerodynamic research, relies on distributed environments to process and analyse large amounts of data. New infrastructures have emerged to support the execution of such computations, such as computing grid and petascale architectures. As the amount of data increases, the need to provide efficient, easy to use and reliable storage solutions has become one of the main concern of datacenters administrators.

Nowadays, the principle storage solution used by supercomputers, clusters and datacenters is, of course, Distributed File Systems (DFSs). DFSs provide *permanent storage* for sharing multiple files, and build a *hierarchical and unified view* of these files by federating storage resources dispersed in a network. High performance computing applications heavily rely on these DFSs.

We consider that a thorough study and a comparison including older DFSs is needed to guide users in their choice. In this paper, we give a presentation and a comparison of four new popular and two former DFSs based on three fundamental issues: *scalability*, *transparency* and *fault tolerance*. We make the choice to study popular, used in production and frequently updated DFSs: HDFS [1], MooseFS[1], iRODS [2, 3, 4], Ceph [5, 6, 7], GlusterFS [8, 9] and Lustre [10, 11, 12, 13, 14].

This paper is organized as follows. Section 1 provides definitions and related work. Section 2 is dedicated to the introduction of the distributed file systems studied. Finally, we analyse and compare the DFSs in Section 3 and introduce some tests in Section 4.4.

[1] http://www.moosefs.org/

Chapter 1

Definitions and related work

In this section, we remind basic issues, designs and features of DFSs. These definitions are based on those of Levy and SilberSchatz [15].

1.1 Distributed file system

A DFS provides a *permanent storage* in which a set of objects that exist from their explicit creation until their explicit destruction are immune to systems failures. This permanent storage consists of several federated storage resources in which clients can create, delete, read or write files. Unlike local file systems, storage resources and clients are dispersed in a network. Files are shared between users in a *hierarchical and unified view*: files are stored on different storage resources, but appear to users as they are put on a single location. A distributed file system should be *transparent, fault-tolerant and scalable*. We now define these three notions:

- *Transparency*: users should access the system regardless where they log in from, be able to perform the same operations on DFSs and local filesystems, and should not care about faults due to distributed nature of the filesystem thanks to fault tolerance mechanisms. Transparency can also be seen in term of performance: data manipulations should be at least as efficient as on conventional file systems. In short, the complexity of the underlying system must be hidden to users.
- *Fault tolerance*: a fault tolerant system should not be stopped in case of transient or partial failures. Faults considered are: network and server failures that make data and services unavailable, data integrity and consistency when several users concurrently access data.
- *Scalability*: this is the ability to efficiently leverage large amounts of servers which are dynamically and continuously added in the system. Usually, it is about ten of thousands of nodes.

DFSs are designed to answer these issues. This is discussed in the next three Sections.

1.2 Scalable architecture

There are no step by step designs for DFSs, but some designs should be avoided to build a scalable system. For example *centralised system* depending on a single server can be a bottleneck for the system. Limits of a single server processing users' requests can be

reached more quickly than limits of several. Another example is about network congestion in which scalability depends on machines interactions. Performing a lot of data transfers or sharing a lot of messages can lead to congestion. Therefore, some key concepts must be taken into account before building one.

Currently, some systems still adopt a centralised architecture, but provide tools to push limits. *Multi-Threading* are one of such tools. Requests will use little resources and will not block the entire server in the contrary of single threading systems. Thus, several requests can be processed in parallel, but the system will always be limited to the computation power of the machine it runs on. Though local parallelism using multi-threading improves the capacities of a system to scale, it is far to be enough for the today's volumes of data. Another solution consists in *caching data*, which can reduce the number of data transfers. This is discussed in Section 1.3.3.

Following these observations and those of Thanh and al. [16], we now introduce some DFSs' architectures:

- In **Client-Server Architectures**, several servers manage, store and share metadata (information about data) and data between multiple clients by providing a *global namespace* of the system (see Section 1.3.1). Adding new servers increases both storage and query processing capacity.
- In **Cluster-Based Distributed File System** metadata and data are decoupled. One or more servers are dedicated to manage metadata and several ones store data. A system with only one metadata server is called **centralised**, whereas a system with distributed metadata servers is called **totally distributed**.

All of these DFSs can also be **parallel**: data are divided into several blocks which are simultaneously distributed across several servers, thus maximising throughput. This is called *striping* [16, 17].

1.3 Transparency

In a DFS, the end-user does not need to know how the system is designed, how data is located and accessed, and how faults are detected. In this section, we present some features that ensure transparency.

1.3.1 Naming

This is a mapping between a logical name and a physical location of a data. For example, in a classic file system, clients use logical name (textual name) to access a file which is mapped to physical disk blocks. In a DFS, servers name holding the disk on which data is stored must be added. DFSs must respect *location transparency*: details of how and where files are stored are hidden to clients. Furthermore, multiple copies of a file (see Section 1.4.1) may exist, so mapping must return a set of locations of all the available copies. DFSs should also be *location independent*: the logical name should not change even if the file is moved to another physical location. To do so, allocation tables [1] or sophisticated algorithms [7] are used to provide a global name space structure, that is the same name space for all clients. For more technical details about naming see [15].

1.3.2 Data access Interfaces

Clients can create, read, write, delete files without thinking about the complex mechanisms of the underlying system which performs operations and must be provided with an access to the system with the help of simple tools. Here are some examples of such tools:

- *Command line interface (CLI)* is used to access files with traditional unix command (cp, rm, mv ...).
- *Java, C, C++, other programming languages and REST (web-based) API* can be used to design graphic interface like the Windows explorer.
- Users can be allowed to *mount* (attach) remote directories to their local file system, thus accessing remote files as if they were stored in a local device. The FUSE mechanism or the unix mount command are some examples.

1.3.3 Caching

This is a technique which consists in temporarily storing requested data into client's memory. DFSs use *caching* to avoid additional network traffic and CPU consumption caused by repeated queries on the same file and thus increase performance [18]. When a data is requested for the first time, a copy is made from the server that holds this data to the client's main memory. Thus for every future request of this data, the client will use the local copy, avoiding communications with the server and disk access. This feature is related to *performance transparency* since with this technique requests can be quickly performed, hiding data distribution to users. However, when a data is changed, the modification must be propagated to the server and to any other clients that have cached the data. This is the *cache consistency problem* discussed in Section 1.4.3.

1.3.4 Fault detection

This is the ability to detect overloaded servers (see Section 1.4.4), incorrect behaviour of a server or corrupted data, and make decisions to correct these faults. In DFSs, faults (see Section 1.4) must be detected by the system, using minimum resources, before corrected, so that, users do not be aware that such faults occur. All machines communicate together in a transparent manner by exchanging small messages. For example, *report* [1] allows servers managing the name space to know what data are held by which server. Since data are always in movement (see Section 1.4.4), this allows the system to identify which data is lost, needs to be moved or recopied when a server become unavailable or overloaded. Another message is *heartbeats* [1] which is used to confirm servers availability. If one does no send heartbeats for a time, it is moved to quarantine and report messages are used to apply the correct decision.

1.4 Fault tolerance

In DFSs, network and servers failures are the norm rather than the exception. Tools must be deployed to maintain data always available, to guarantee query processing in case of faults. Integrity and consistency of data must also be taken into account since mechanisms like caching or replication are provided. We present in this section some features to ensure fault tolerance.

1.4.1 Replication and placement policy

In order to make data always available, even if a server crashes, DFSs use replication of files by making several copies of a data on different servers. When a client requests a data, he transparently accesses one of the copies. To improve fault tolerance, replicas are stored on different servers according to a *placement policy*. For example, replicas can be stored on different nodes, on different racks, at different geographical locations, so that

if a fault occurs anywhere in the system, data is still available [1]. However this can lead to consistencies issues that are discussed in the next Section.

1.4.2 Synchronisation

In DFSs, *synchronisation* between *copies* (see Section 1.4.1) of data must be taken into account. When a data is rewritten all of its copies must be updated to provide users with the latest version of the data. Three main approaches exist:

- In the *synchronous* method any request on modified data is blocked until all the copies are updated. This ensures the users access the latest version of the data, but delays queries executions.
- In the second method called *asynchronous*, requests on modified data are allowed, even if copies are not updated. This way, requests could be performed in a reasonable time, but users can access to an out-of-date copy.
- The last approach is a trade-off between the first two. In the *semi-asynchronous* method, requests are blocked until some copies, but not all, are updated. For example, let assume there are five copies of a data, a request on this data will be allowed once three copies will be updated. This limits the possibility to access to an out-of-date data, while reducing delay for queries executions.

1.4.3 Cache consistency

This is the same problem as synchronisation: how to update all copies of a data in cache, when one them is modified. As seen in Section 1.3.3 data can be cached to improve the performance of the system, which can lead to inconsistencies between copies when one of them is changed by a user. These modification needs to be propagated to all copies and data in cache to provide users an up-to-date version of them [18]. To avoid this problem, different approaches are used:

- *Write Only Read Many* (WORM): is a first approach to ensure consistency. Once a file is created, it cannot be modified. Cached files are in read-only mode, therefore each read reflects the latest version of the data.
- A second method is *Transactional locking* which consists in: obtaining a read lock on the requested data, so that any other users cannot perform a write on this data; or obtaining a write lock in order to prevent any reads or writes on this data. Therefore, each read reflect the latest write, and each write is done in order.
- Another approach is *Leasing*. It is a contract for a limited period between the server holding the data and the client requesting this data for writing. The lease is provided when the data is requested, and during the lease the client is guaranteed that no other user can modify the data. The data is available again if the lease expires or if the client releases its rights [19]. For future read requests, the cache is updated if the data has been modified. For future write requests a lease is provided to the client if allowed (that is, no lease exists for this data or the rights are released).

1.4.4 Load balancing

is the ability to auto-balance the system after adding or removing servers. Tools must be provided to recover lost data, to store them on other servers or to move them from a hot device to a newly added one. As seen in Section 1.3.4 communications between machines

allow the system to detect servers failures and servers overload. To correct these faults, servers can be added or removed. When a server is removed from the system, the latter must be able to recover the lost data, and to store them on other servers. When a server is added to the system, tools for moving data from a hot server to the newly added server must be provided. Users don't have to be aware of this mechanism. Usually, DFSs use a scheduled list in which they put data to be moved or recopied. Periodically, an algorithm iterates over this list and performs the desired action. For example, Ceph uses a function called **Controlled Replication Under Scalable Hashing (CRUSH)** to randomly store new data, move a subset of existing data to new storage resources and uniformly restore data from removed storage resources [5].

1.5 Related work

Many DFS's have been developed over the years providing different architectures and features [10, 1, 5]. Little comparisons have been done on DFSs making users hesitant in their choice. In 1989, Satyanarayanan [20] provides a taxonomy about the different issues a DFS can meet and must solve. A detailed introduction on Sun NFS, Apollo Domain, AFS, IBM AIX DS, AT&T RFS and Sprite is given and a comparison according to the taxonomy is made. However, new DFSs have appeared (Ceph, GlusterFS ...) and only a few of those studied (NFS, AFS) are still in use. In [16] Thanh and al. take back Satyanarayanan's taxonomy to compare contemporary DFSs like Panasas, PVFS, GFS ... Nevertheless, no presentation on the surveyed DFSs is provided and only a summary table is given. More recently, HDFS, Lustre and MooseFS were briefly studied [21]. The comparison focus on few performance and functions which are little detailed. Since 2011, architectures and features have not dramatically changed but new DFSs have gained popularity (Ceph, GlusterFS, iRODS) and must be taken into account for a new comparison. Furthermore, the survey of this DFSs must be more detailed. In the next section, we present the scalability, transparency and fault-tolerance of six DFSs: HDFS, MooseFS, iRODS, Ceph, GlusterFS and Lustre.

Chapter 2

Introduction of distributed file systems surveyed

It is difficult to make an exhaustive study given the number of existing DFSs. In this paper, we choose to study popular, used in production, and frequently updated DFSs: HDFS [1], MooseFS[1], iRODS [2, 3, 4], Ceph [5, 6, 7], GlusterFS [8, 9] and Lustre [10, 11, 12, 13, 14].

2.1 HDFS

HDFS[2] is the Hadoop Distributed File System under Apache licence 2.0 developed by the Apache Software Foundation [1].

2.1.1 Architecture

HDFS is a centralised distributed file system. Metadata are managed by a single server called the *namenode* and data are split into blocks, distributed and replicated at several *datanodes*. A *secondary namenode* is provided and is a persistent copy of the namenode. This allows HDFS to restart with an up-to-date configuration, in case of namenode failures, by restoring the namespace from the secondary namenode.

2.1.2 Naming

HDFS handles its name space in a hierarchy of files and directories using inodes which hold metadata such as permissions, space disk quota, access time... The name space and metadata are managed by the namenode which also performs the mapping between filename and file blocks stored on the datanodes.

2.1.3 API and client access

HDFS provides a code library that allows users to read, write and delete file and create and delete directories. Users access a file using its path in the namespace and contact the namenode to know where to retrieve or put files' blocks, and then request the transfer by communicating directly with the datanodes. This is done in a transparent way using some API in C, Java or REST. HDFS also provides a module based on FUSE (file system

[1] http://www.moosefs.org/
[2] http://hadoop.apache.org/

in user space). This is an interface which exposes users with a virtual file system which corresponds to a physical remote directory. Thus, each client request is relayed to a remote file by this interface.

2.1.4 Cache consistency

HDFS implements a write only read many model: when a file is created, data written to it, and then the file is closed, it cannot be modified anymore. A client wanting to create a new file is granted a lease which guarantees exclusive write access to the file. Thus no concurrent write access is allowed. However, HDFS allows users to read a file that is currently written. In this case and in case of network or servers failures, data consistency is ensured by a checksum which is generated for each data blocks. When a client wants to read a file, he compares the checksum stored in HDFS with the checksum he computed.

2.1.5 Replication and synchronisation

HDFS splits data into blocks which are replicated and distributed across several datanodes according to a default placement policy. No datanode holds more than one copy of a block, and no rack holds more than two copies of the same block. The namenode verifies that each block has the intended number of replicas. If a block is over-replicated, the namenode chooses a replica to delete, trying not to reduce the number of racks in which replicas are stored, and preferring to remove a replica on the datanode with the least amount of available disk space. If a block is under-replicated, creation of a new replica is scheduled and placed in a priority queue which is verified periodically. The new block will be stored in accordance with the placement policy. If all replicas of a block are on the same rack, the namenode will add a new replica on a different rack. The block will become over-replicated, thus triggering the deletion of some replicas. Finally, data are replicated in asynchronous mode, but tools are provided to tolerate synchronous mode.

2.1.6 Load balancing

HDFS defines the utilisation of a server (or cluster) as the ratio of the space used to the total capacity of the server (or cluster), and fixes a threshold value in the range of (0,1). It decides that a cluster is balanced if for each datanode the usage of the server is different from the usage of the cluster by no more than the threshold value. In HDFS, if a node is unbalanced, replicas are moved from it to another one respecting the placement policy.

2.1.7 Fault detection

Servers in HDFS are fully connected and communicate with each other to detect some faults such as network or server failures and to keep the system secure and available. At startup each datanode compares the software version and the *namespace ID* (which is assigned when the datanode is formatted) with those of the namenode. If they don't match, the datanode shuts down, thus preserving the integrity of the system. Datanodes also perform a *registration* with the namenode which consists in making the datanodes recognizable even if they restart with a different IP address or port. After this registration and every hour, datanodes send *block reports* (information about blocks held) to the namenode in order to provide the latter with an up-to-date view of the location of each block. Every three seconds, datanodes send *heartbeats* to the namenode to confirm their availability. The namenode considers datanodes as out-of-service if it does not receive any heartbeats during ten minutes. Heartbeats also provide statistic information (storage capacity, number of data transfers in progress...) to the namenode so that it can make

decisions for load balancing. Namenode's instructions (to correct faults) like removing or replicating a block are also sent to datanodes thanks to heartbeats.

2.2 MooseFS

MooseFs[3] is an open source (GPL) distributed file system developed by Gemius SA.

2.2.1 Architecture

MooseFS acts as HDFS. It has a *master server* managing metadata, several *chunk servers* storing and replicating data blocks. MooseFS has a little difference since it provides failover between the master server and the *metalogger servers*. Those are machines which periodically download metadata from the master in order to be promoted as the new one in case of failures.

2.2.2 Naming

MooseFS manages the namespace as HDFS does. It stores metadata (permission, last access...) and the hierarchy of files and directories in the master main memory, while performing a persistent copy on the metalogger. It provides users with a global name space of the system.

2.2.3 API and client access

MooseFS clients access the file system by mounting the name space (using the `mfsmount` command) in their local file system. They can perform the usual operations by communicating with the master server which redirects them to the chunk servers in a seamless way. `mfsmount` is based on the FUSE mechanism.

2.2.4 Replication and synchronisation

In MooseFS, each file has a *goal*, that is, a specific number of copies to be maintained. When a client writes data, the master server will send it a list of chunks servers in which data will be stored. Then, the client sends the data to the first chunk server which orders other ones to replicate the file in a synchronous way.

2.2.5 Load balancing

MooseFS provides load balancing. When a server is unavailable due to a failure, some data do not reach their goal. In this case, the system will store replicas on other chunks servers. Furthermore, files can be over their goal. If such a case appears, the system will remove a copy from a chunk server. MooseFS also maintains a data version number so that if a server is again available, mechanisms allow the server to update the data it stores. Finally, it is possible to dynamically add new data server which will be used to store new files and replicas.

2.2.6 Fault detection

In MooseFS, when a server becomes unavailable, it is put in quarantine and no I/O operations to it can be perform.

[3] http://www.moosefs.org/

2.3 iRODS

iRODS[4] [2, 3, 4] is a highly customisable system developed by the Data Intensive Cyber Environments (DICE) research group.

2.3.1 Architecture

iRODS, a centralised system, has two major components: the *iCat server* which stores metadata in a database and handles queries to these metadata; and several *iRODS servers* which store data to storage resources. An iCat server and several iRODS servers form a *zone*. Compared to the other distributed file systems, iRODS relies on storage resources' local file system (Unix file system, NTFS...) and does not format or deploy its own file system.

2.3.2 Naming

iRODS stores the name space and metadata in a database, and provides tools similar to SQL, to query the metadata . Users can see the same hierarchy of files and directories like in Unix file system (e.g., /home/myname/myfile.txt). iRODS also provides tools to federate different zones, making files of one zone reachable to clients of another zone.

2.3.3 API and client access

Clients in iRODS need to connect to only one server (iCat or iRODS server), since iRODS ensures routing between the different components. iRODS provides a client interface in command line, a FUSE module and some API (PHP, Java ...) to ensure I/O operations and to process queries. Similarly to DFSs seen above, clients communicate with the iCat server for metadata requests and directly with iRODS servers for the I/O.

2.3.4 Cache consistency

iRODS uses a WORM mechanism. However, in some cases, data could be rewritten with a `--force` option. In this case other options are provided to ensure consistency: `--wlock` and `--rlock`. These are write lock and read lock which allow to block a file during a write or read operation.

2.3.5 Replication and synchronisation

By default, iRODS does not automatically replicate data. Instead it implements a replication command (`irepl`) which can be manually run like other usual operations (`iput`, `iget`...). However, iRODS being a customisable system, it is possible to create *rules*, that is, designing little commands to be run after a specific operation. For example, building a rule that order the system to replicate data after its creation. In this case, the copies are made in a synchronous fashion. The placement policy is however the responsibility of the users. iRODS also allows users to create groups of storage resources and to choose in which group they want to store data and replicas.

2.3.6 Load balancing

In iRODS, storage resources belong to a default group or a group created by users. These resources are monitored to measure servers activity. A rule is periodically run to determine

[4] https://www.irods.org/index.php

whenever a server is overloaded, according to configurable parameters (CPU load, used disk space...) and allows iRODS to choose the appropriate storage in a group to place new data. However, it is possible to tell iRODS to avoid or force a data to be placed on a specific resource using other rules. Users can also move data from a storage to another one. Therefore, just like replication, users can choose how to balance the system.

2.3.7 Fault detection

iRODS is implemented as a peer-to-peer fashion. Thus, servers can communicate together to detect whenever a server become unavailable. In this case, the inaccurate server is removed from the group of storage resources and client cannot perform any I/O operations from or to this server.

2.4 Ceph

Ceph[5] [5, 6, 7] is an open source (LGPL) distributed file system developed by Sage Weil.

2.4.1 Architecture

Ceph is a totally distributed system. Unlike HDFS, to ensure scalability Ceph provides a dynamic distributed metadata management using a metadata cluster (*MDS*) and stores data and metadata in *Object Storage Devices (OSD)*. MDSs manage the namespace, the security and the consistency of the system and perform queries of metadata while OSDs perform I/O operations.

2.4.2 Naming, API and client access

In contrast to the other DFSs in which metadata server looks up for the localisation of each data blocks, Ceph allows clients to calculate which objects comprise the requested data. Indeed, each data are striped into blocks which are assigned to objects. These objects are identified by the same inode number plus an object number, and placed on storage devices using a function called *CRUSH* [7]. When a client requests a data, it contacts a node in the MDSs cluster which sends it the inode number and the file size. Then, the client can calculate how many objects comprise the file, and contacts the OSDs cluster to retrieve the data with the help of CRUSH. Transparency is ensured thanks to a REST API or by mounting the name space in user space (FUSE or mount command).

2.4.3 Cache consistency

Ceph is a near POSIX system in which reads must reflect an up-to-date version of data and writes must reflect a particular order of occurrences. It allows concurrent read and write accesses using a simple locking mechanism called *capabilities*. These latter, granted by the MDSs, give users the ability to read, read and cache, write or write and buffer data. Ceph also provides some tools to relax consistency. For example the `O_LAZY` options allows users to read a file even if it is currently rewritten.

2.4.4 Replication and synchronisation

When a client writes a data, corresponding objects are put on different *placement group (PG)* and then stored on OSDs. The choice of PGs and OSDs is ensured by the CRUSH

[5] http://ceph.com/

function according to free disk space and weighted devices and using the same placement policy as HDFS. Ceph implements three synchronous replication strategies: *primary-copy*, *chain* and *splay replication*. In primary-copy replication, the first OSD in the PG forwards the writes to the other OSDs and once the latter have sent a acknowledgement, it applies its writes, then reads are allowed. In chain replication, writes are applied sequentially and reads are allowed once the last replication on the last OSD have been made. Finally, in splay replication, half of the number of replicas are written sequentially and then in parallel. Reads are permitted once all OSDs have applied the write.

2.4.5 Load balancing

Ceph ensures load balancing at two levels: data and metadata. Firstly, it implements a counter for each metadata which allows it to know their access frequency. Thus, Ceph is able to replicate or move the most frequently accessed metadata from an overloaded MDS to another one. Secondly, Ceph uses weighted devices. For example, let assume having two storage resources with respectively a weight of one and two. The second one will store twice as much data as the first one. Ceph monitors the space disk used by each OSD, and moves data to balance the system according to the weight associated to each OSD. For example, let assume a new device is added. Ceph will be able to move data from OSDs which have become unbalanced to this new device.

2.4.6 Fault detection

Ceph uses the same fault detection model as HDFS. OSDs periodically exchange heartbeats to detect failures. Ceph provides a small cluster of *monitors* which holds a copy of the cluster map. Each OSD can send a failure report to any monitor. The inaccurate OSD will be marked as down. Each OSD can also query an up-to-date version of the cluster map from the cluster of monitor. Thus, a formerly defaulting OSD can join the system again.

2.5 GlusterFS

GlusterFS[6] [8, 9] is an open source (GPL) distributed file system developed by the gluster core team.

2.5.1 Architecture

GlusterFS is different from the other DFSs. It has a client-server design in which there is no metadata server. Instead, GlusterFS stores data and metadata on several devices attached to different servers. The set of devices is called a *volume* which can be configured to stripe data into blocks and/or replicate them. Thus, blocks will be distributed and/or replicated across several devices inside the volume.

2.5.2 Naming

GlusterFS does not manage metadata in a dedicated and centralised server, instead, it locates files algorithmically using the *Elastic Hashing Algorithm (EHA)* [8] to provide a global name space. EHA uses a hash function to convert a file's pathname into a fixed length, uniform and unique value. A storage is assigned to a set of values allowing the system to store a file based on its value. For example, let assume there are two storage

[6] http://www.gluster.org/

devices: disk1 and disk2 which respectively store files with value from 1 to 20 and from 21 to 40. The file myfile.txt is converted to the value 30. Therefore, it will be stored onto disk2.

2.5.3 API and client access

GlusterFS provides a REST API and a mount module (FUSE or mount command) to give clients with an access to the system. Clients interact, send and retrieve files with, to and from a volume.

2.5.4 Cache consistency

GlusterFS does not use client side caching, thus, avoiding cache consistency issues.

2.5.5 Replication and synchronisation

Compared to the other DFSs, GlusterFS does not replicate data one by one, but relies on RAID 1. It makes several copies of a whole storage device into other ones inside a same volume using synchronous writes. That is, a volume is composed by several subsets in which there are an initial storage disk and its mirrors.

Therefore, the number of storage devices in a volume must be a multiple of the desired replication. For example, let assume we have got four storage medias, and replication is fixed to two. The first storage will be replicated into the second and thus form a subset. In the same way, the third and fourth storage form another subset.

2.5.6 Load balancing

GlusterFS uses a hash function to convert files' pathnames into values and assigns several logical storage to different set of value. This is uniformly done and allows GlusterFS to be sure that each logical storage almost holds the same number of files. Since files have not the same size and storage devices can be added or removed, GlusterFS affects each logical storage to a physical one according to the used disk space. Thus, when a disk is added or removed, virtual storage can be moved to a different physical one in order to balance the system.

2.5.7 Fault detection

In GlusterFS, when a server becomes unavailable, it is removed from the system and no I/O operations to it can be perform.

2.6 Lustre

Lustre[7] [10, 11, 12, 13, 14] is a DFS available for Linux and is under GPL licence.

2.6.1 Architecture

Lustre is a centralised distributed file system which differs from the current DFSs in that it does not provide any copy of data and metadata. Instead, Lustre chooses to It stores metadata on a shared storage called *Metadata Target (MDT)* attached to two *Metadata Servers (MDS)*, thus offering an active/passive failover. MDS are the servers that handle the requests to metadata. Data themselves are managed in the same way. They are split

[7] http://wiki.lustre.org/index.php/

into objects and distributed at several shared *Object Storage Target (OST)* which can be attached to several *Object Storage Servers (OSS)* to provide an active/active failover. OSS are the servers that handle I/O requests.

2.6.2 Naming

The Lustre's single global name space is provided to user by the MDS. Lustre uses inodes, like HDFS or MooseFS, and extended attributes to map file object name to its corresponding OSTs. Therefore, clients will be informed of which OSTs it should query for each requested data.

2.6.3 API and client access

Lustre provides tools to mount the entire file system in user space (FUSE). Transfers of files and processing queries are done in the same way like the other centralised DFSs, that is, they first ask the MDS to locate data and then directly perform I/O with the appropriate OSTs.

2.6.4 Cache consistency

Lustre implements a *Distributed Lock Manager (DLM)* to manage cache consistency. This is a sophisticated lock mechanism which allows Lustre to support concurrent read and write access [14]. For example, Lustre can choose granting write-back lock in a directory with little contention. Write-back lock allows clients to cache a large number of updates in memory which will be committed to disk later, thus reducing data transfers. While in a highly concurrently accessed directory, Lustre can choose performing each request one by one to provide strong data consistency. More details on DLM can be found in [14].

2.6.5 Replication and synchronisation

Lustre does not ensure replication. Instead, it relies on independent software.

2.6.6 Load balancing

Lustre provides simple tools for load balancing. When an OST is unbalanced, that is, it uses more space disk than other OSTs, the MDS will choose OSTs with more free space to store new data.

2.6.7 Fault detection

Lustre is different from other DFSs because faults are detected during clients' operations (metadata queries or data transfers) instead of being detected during servers' communications. When a client requests a file, it first contacts the MDS. If the latter is not available, the client will ask a LDAP server to know what other MDS it can question. Then, the system perform a failover, that is, the first MDS is marked as dead, and the second become the active one. A failure in OST is done in the same way. If a client sends data to an OST which does not respond during a certain time it will be redirected to another OST.

Chapter 3

Analysis and Comparison

3.1 Scalability

DFSs must face with an increasing number of clients performing requests and I/O operations and a growing number of files of different sizes to be stored. Scalability is the system's ability to grow to answer the above issues without disturbing system's performance. Here, we discuss about the benefits and disadvantages of the different architectures used.

3.1.1 Discussion about architecture

In Lustre, metadata are managed by a single server and stores on its storage devices. The number of client's requests per second that can be performed relies on the single server's computing power and on disk latency. Therefore, this system is limited. HDFS and MooseFS meets the same problems but choose to store metadata on the metadata server's memory. This improves the time to perform a client's request, but the number of files that can be created is smaller than in Lustre. iRODS also uses a centralised architecture but relies on a database to manage and store metadata. This allows iRODS to increase client's request using a SQL query while storing more files than in HDFS or MooseFS. However, this number of files depends on the available disk space and remains limited.

GlusterFS stores metadata on data servers, setting up to unlimited the number of files

Table 3.1: Summary table

	HDFS	iRODS	Ceph	GlusterFS	Lustre
Architecture	Centralized	Centralized	Distributed	Decentralized	Centralized
Naming	Index	Database	CRUSH	EHA	Index
API	CLI, FUSE REST, API	CLI, FUSE API	FUSE, mount REST	FUSE, mount	FUSE
Fault detection	Fully connect.	P2P	Fully connect.	Detected	Manually
System availability	No failover	No failover	High	High	Failover
Data availability	Replication	Replication	Replication	RAID-like	No
Placement strategy	Auto	Manual	Auto	Manual	No
Replication	Async.	Sync.	Sync.	Sync.	RAID-like
Cache consistency	WORM, lease	Lock	Lock	No	Lock
Load balancing	Auto	Manual	Manual	Manual	No

Table 3.2: Input and Output performances

	HDFS		iRODS		Ceph		GlusterFS		Lustre		MooseFS	
Input/Output	I	O	I	O	I	O	I	O	I	O	I	O
1 × 20GB	407s	401s	**520s**	**500s**	419s	**382s**	**341s**	403s	374s	415s	448s	385s
1000 × 1MB	72s	17s	**86s**	**23s**	76s	21s	**59s**	18s	66s	5s	68s	**4s**

that can be created. Furthermore, it does not separate data and metadata management which allows it to quickly scale by just adding one server. Ceph acts as GlusterFS but distributes the metadata management across several metadata servers. It allows it to face with a large number of client's requests. However, to increase both amount of data and client's queries, Ceph needs to add two kind of servers: metadata or data, which makes the system more complex to scale.

3.1.2 Small or big files?

Based on the information above, HDFS, MooseFS, iRODS and Lustre are more suitable to store small quantity of big files whereas Ceph and GlusterFS can hold both small and big data. However, note that, except iRODS, all the DFSs surveyed use *striping* to speed up data transfers, but it is only beneficial for big files, since small files can not be split into blocks. We have perform a simple test on grid5000 platform [1] on *the pastel cluster* [2]. We have measured the time to put (write) and get (read) a 20GB data and one thousand of 1MB data. We have used two metadata servers for Ceph to benefit from metadata distribution, four data servers, one client and no replication. The table 3.2 shows the results for read and write operations. Red color indicates the best result while the blue one indicates the worst results. Though these tests depend on different factors like network traffic and more tests are needed to conclude, we can see that the GlusterFS' architecture perform better than the others for writing small files. This may be due to the distributed request management. The second observation is that striping speed up the performance on big files. Indeed, the only system (iRODS) that does not use this method obtains the worst result. Note that the two Ceph's MDS does not improve performance. Finally, no DFSs are better for reading files, may be except Lustre for small files. Other tests are available in Section 4.4.

3.2 Transparency

In a DFS, the complexity of the underlying system must be hidden to users. They should access a file and perform operations in a same way as in a local file system and should not care about faults due to distributed nature of the filesystem. We now compare the different features used to ensure transparency.

3.2.1 File access & operations transparency

Although DFSs use their own methods to locate files, all of them provide users with a global namespace of the system and APIs are also the same regardless of the DFS. Information about API can be found in table 3.1.

HDFS, MooseFS, iRODS and Lustre maintain an index in which a physical location is associated to a filename This is easy to maintain since when a file is moved from a storage to another one, created or deleted, the index is simply updated. The main disadvantage

[1] https://www.grid5000.fr
[2] https://www.grid5000.fr/mediawiki/index.php/Toulouse:Hardware

is that it is the responsibility of the metadata server to find where data are stored when a client request a file, adding more computing pressure on this server. Moreover, HDFS and MooseFS store metadata in the memory, restricting the number of files to be created. This is not the case for iRODS and Lustre since they put metadata on large space disk.

Ceph and GlusterFS use an algorithm to calculate data's location. It reduces the metadata servers workload because this is clients that search for data's location. Metadata servers only have to provide the information needed to correctly run the algorithm. Nevertheless, contrary to maintaining an index, with this method when clients request a file, they do not immediately know where the data is stored but they need to calculate the data's location before accessing them.

3.2.2 Fault detection

Failures must be detected by the system before users. HDFS, MooseFS, iRODS, Ceph and GlusterFS provide strong fault detection. Servers are fully connected and can detect when one of them becomes unavailable. In this case, it is put in quarantine or removed from the system limiting the risk that users are aware of such faults and thus ensuring transparency. However, this implies exchanging a high number of messages between servers which could have an impact on performance. In Lustre no mechanism is provided to detect and correct an unavailable server. Users which try to access an inaccurate server will have to ask another one by himself. therefore, it does not fit with the transparency issue introduced in Section 1.

3.2.3 System access

In this test, we try to access to a cluster in a private network from a client in another one with only a ssh connection using port forwarding. We easily succeed for MooseFS and iRODS and clients can transparently read and write files. For HDFS, we can communicate with the namenode and so perform all the operations related to metadata (`ls`, `stat` ...). However, to write files, we need to forward datanodes ports too which is more complex. Currently, for Ceph and GlusterFS, we have not succeeded yet to interact with the system from a outside client. Therefore, in this configuration, the system is less transparent than the others. More details in Section 4.2

3.3 Fault tolerance

The distributed nature of DFS implies failures are the norm rather than the exception. As seen before, faults considered can be: network and server failures that make data and services unavailable, data integrity and consistency when several users concurrently access data and finally, servers overloaded. In this section we compare the different features used by the DFSs studied to face with these faults.

3.3.1 System availability

Ceph and GlusterFS are highly available. Metadata are replicated and their management is distributed across several servers. Contrary, in a centralised system, the metadata server is a single point of failure (SPOF) which can cause metadata loss and system unavailability. To solve this problem, HDFS chooses to periodically save the metadata and the namespace in a secondary namenode, allowing it to restart in a healthy state. However, during this reboot, the system remains unavailable. In its side, iRODS provides tools to replicate the database (`pgpool`) but, like HDFS, the system is unavailable until the iCAT server is restarted. Finally, MooseFS and Lustre succeeds in avoiding the SPOF

using failover: several metadata servers, in standby, periodically save the metadata to be ready to take control of the system.

3.3.2 Data availability & data synchronisation

Regardless the DFS, data can be inaccessible if a server crashes. Except Lustre, the DFSs surveyed use replication: several copies of data are made so that, in case of failures, there is at least one available replica. However, some consistency issues are raised: all replicas must be synchronised. HDFS uses asynchronous replication: when a data is written, it is possible to request it even if replicas are not committed to disk. Usually, it does not solve consistency issues since some out-to-date replicas can be accessed, but HDFS solves this problem relying on WORM mechanism: once created, data can not be modified. Therefore, consistency issues will only appear when file will be created. HDFS handles this problem by granting lease on files during creation, making them inaccessible until they are committed to disk. iRODS, Ceph and GlusterFS use synchronous replication: queries on a file are blocked until all replicas are saved to disk. It avoids consistency problems, but data are unavailable during the synchronisation. Furthermore, some DFSs are better protected from failures by providing a placement strategy. HDFS and Ceph automatically store replicas on different geographical rack whereas iRODS and GlusterFS let administrators to determine the placement strategy to use. They all provide strong data availability. In its side, MooseFS does not provide any placement strategy which makes it more vulnerable to outage or network failures. Moreover, HDFS, MooseFS and Ceph automatically verify if the desired replication is satisfied and replicate or remove copies when it is not the case whereas in iRODS this is done manually. Finally, by default, Lustre does not make data always available. Instead, it relies on independent software which must be set up in addition. Without, these independent tools, this system is not well protected from failures.

3.3.3 Load balancing & cache consistency

Overloaded servers can delay or abort request execution. While Ceph and GlusterFS provide algorithms to distribute the metadata workload and allows to dynamically add new servers, the single metadata server of the centralised system is a bottleneck. To counter this problem, these DFSs use thread which allows them to perform several requests in parallel. They also cache data on client side to avoid useless query. However, these solutions merely push the limits of the system but do not delete them. Moreover, caching data can lead to consistency issues: a data being updated must not be accessible by other users. As seen in the above Section, HDFS does not allow files modification, removing consistency issues. iRODS and Lustre employ locks on file to ensure concurrent access. Files are blocked until operations are performed, thus, consistency issues are solved. However locks have an impact on data availability: if operations can not be well executed, due to failures, the file might be indefinitely blocked. These DFSs must periodically remove old locks to unblock files. Note that, Ceph also use caching with locks. Another problem is overloaded data server which causes network congestion: if a data server stores more files than the others, it will perform more I/O operations. All the DFSs detect overloaded servers in different ways. MooseFS and Lustre stores new data on server which have the most free disk space, but they do not relieve overloaded servers. iRODS must be configured to avoid the overload whereas HDFS, Ceph and GlusterFS succeed in it by placing data according to free disk space and moving data from an overloaded server to another one. Furthermore, DFSs allow to dynamically add new data servers. As seen above, in MooseFS and Lustre, adding new servers will only be used for new data but will not relieve overloaded servers. In iRODS, Ceph and GlusterFS, commands must be manually

run to perform a load balancing whereas HDFS is better since it is automatically done.

3.3.4 Test on data servers

Here, we have just simulated a crash on a data server. For all DFSs, the inaccurate server is detected and put in quarantine. Data are still available, except for Lustre, thanks to replication and the desired number of replicas is maintained except for GlusterFS and iRODS. These tests are detailed in Section 4.3.

Chapter 4

Tests achieved on the DFSs

We have perform some tests on the different DFSs on grid5000 platform. In this chapter, we explain how we have set up the DFSs surveyed on this platform, detail how we have accessed to the DFSs from an outside network, show the DFSs' behaviour in case of faults and finally introduce the results of some performance tests.

4.1 Setting up the DFSs studied on grid5000 platform

The grid5000 platform allows users to reserve some nodes on different clusters[1] and to deploy an environment on these nodes. Several image are available according to the cluster[2]. Here, we reserve nodes on the *pastel cluster*[3] and deploy a debian image:

```
local~: ssh toulouse.grid5000.fr
toulouse~: oarsub -I -t deploy -l nodes=number,walltime=reservation_time
toulouse~: kadeploy3 -f list_of_nodes -e squeeze-x64-base -k
```

Now, we introduce how to set up the different DFSs surveyed.

4.1.1 HDFS

Installation

HDFS[4] needs installing Java before setting it up. Secondly, we have downloaded the hadoop package[5], and put it on all nodes (including clients), then we install it with root permissions:

```
apt-get install sun-java6-jre
dpkg -i hadoop_1.0.3-1_x86_64.deb
```

Configuration

First we choose a node to be the namenode. Then, for all servers, we edit four files: *hdfs-site.xml*, *core-site.xml*, *slaves* and *hadoop-env.sh*. The first includes settings for the namespace checkpoint's location and for where the datanodes store filesystem blocks. The

[1] https://www.grid5000.fr/gridstatus/oargridmonika.cgi
[2] https://www.grid5000.fr/mediawiki/index.php/Category:Portal:Environment
[3] https://www.grid5000.fr/mediawiki/index.php/Toulouse:Hardware
[4] http://developer.yahoo.com/hadoop/tutorial/
[5] http://wwwftp.ciril.fr/pub/apache/hadoop/core/stable/

Table 4.1: HDFS config files

hdfs-site.xml	core-site.xml	slaves
<configuration> <property> <name>dfs.name.dir</name> <value>/tmp/dfs/name</value> </property> <property> <name>dfs.data.dir</name> <value>/tmp/dfs/data</value> </property> </configuration>	<configuration> <property> <name>fs.default.name</name> <value>hdfs://namenode_host:port</value> </property> </configuration>	datanodes1 datanodes2 datanodes3 ... datanodesN

second specifies which node is the namenode, the third must contain all the datanodes' hostname and the last holds the JAVA_HOME variable which specifies the path to Java directory. Note that for hdfs's clients, only the *JAVA_HOME* variable and the *core-site.xml* must be modified. Table 4.1 shows the config files used in our tests.

Running HDFS

Once connected to the namenode, we can start HDFS and then, from the clients, perform some operations:

```
namenode~: hadoop namenode -format
namenode~: start-dfs.sh

user~: hadoop dfs -put local_file hadoop_destination
user~: hadoop dfs -get hadoop_file local_destination
```

4.1.2 MooseFS

MooseFS[6] needs installing *pkg-config* and *zlib1g-dev* before setting it up. Secondly, we have downloaded the MooseFS archive[7] and the fuse package[8]. The latter is needed for MooseFS's clients. On all nodes (including clients) we extract the archive and create a MooseFS group and user:

```
node~: apt-get install pkg-config zlib1g-dev
node~: tar xzf mfs-1.6.25-1.tar.gz
node~: groupadd mfs; useradd -g mfs mfs

user~: tar xzf fuse-2.9.2.tar.gz
```

According to the kind of servers (master, backup, chunk or user), the installation is different.

Master server

- Installation:

[6] http://www.moosefs.org/tl_files/manpageszip/moosefs-step-by-step-tutorial-v.1.1.pdf
[7] http://www.moosefs.org/download.html
[8] http://sourceforge.net/projects/fuse/

```
master~: cd mfs-1.6.25-1
master~: ./configure --prefix=/usr --sysconfdir=/etc
\--localstatedir=/var/lib --with-default-user=mfs
\--with-default-group=mfs --disable-mfschunkserver
\--disable-mfsmount
master~: make; make install
```

- Configuration:
  - First add master's server IP to hosts files:

    ```
    master~: echo "ip_master_server mfsmaster" >> /etc/hosts
    ```

  - Then run the following commands to avoid some errors:

    ```
    master~: cd /etc
    master~: cp mfsmaster.cfg.dist mfsmaster.cfg
    master~: cp mfsmetalogger.cfg.dist mfsmetalogger.cfg
    master~: cp mfsexports.cfg.dist mfsexports.cfg
    master~: cd /var/lib/mfs
    master~: cp metadata.mfs.empty metadata.mfs
    ```

- Running master server:

  ```
  master~: /usr/sbin/mfsmaster start
  ```

Backup server

- Installation:

  ```
  backup~: cd mfs-1.6.25-1
  backup~: ./configure --prefix=/usr --sysconfdir=/etc
  \--localstatedir=/var/lib --with-default-user=mfs
  \--with-default-group=mfs --disable-mfschunkserver \
  \--disable-mfsmount
  backup~: make; make install
  ```

- Configuration:
  - First add master's server IP to hosts files:

    ```
    backup~: echo "ip_master_server mfsmaster" >> /etc/hosts
    ```

  - Then run the following commands to avoid some errors:

    ```
    backup~: cd /etc
    backup~: cp mfsmetalogger.cfg.dist mfsmetalogger.cfg
    ```

- Running backup server:

  ```
  backup~: /usr/sbin/mfsmetalogger start
  ```

Chunk server

- Installation:

```
chunk~: cd mfs-1.6.25-1
chunk~: ./configure --prefix=/usr --sysconfdir=/etc
\--localstatedir=/var/lib --with-default-user=mfs
\--with-default-group=mfs --disable-mfsmaster
chunk~: make; make install
```

- Configuration:
  - First add master's server IP to hosts files:

    ```
    chunk~: echo "ip_master_server mfsmaster" >> /etc/hosts
    ```

  - Then configure the storage which will store data's blocks:

    ```
    chunk~: echo "/tmp" >> mfshdd.cfg
    chunk~: chown -R mfs:mfs /tmp
    ```

  - Finally run the following commands to avoid some errors:

    ```
    chunk~: cd /etc
    chunk~: mfschunkserver.cfg.dist mfschunkserver.cfg
    chunk~: cp mfshdd.cfg.dist mfshdd.cfg
    ```

- Running chunk server:

```
chunk~: /usr/sbin/mfschunkserver start
```

Client

- Installation:
  - FUSE:

    ```
    user~: cd fuse-2.9.2
    user~: ./configure; make; make install
    ```

  - MooseFS:

    ```
    user~: ./configure --prefix=/usr --sysconfdir=/etc
    \--localstatedir=/var/lib --with-default-user=mfs
    \--with-default-group=mfs --disable-mfsmaster
    \--disable-mfschunkserver
    user~: make; make install
    ```

- Configuration:
  - First add master's server IP to hosts files:

    ```
    user~: echo "ip_master_server mfsmaster" >> /etc/hosts
    ```

  - Then create the moun directory:

    ```
    user~: mkdir -p /tmp/mfs
    ```

  - Finally mount the filesystem:

    ```
    user~: /usr/bin/mfsmount /tmp/mfs -H mfsmaster
    ```

- Perform operations:

```
user~: cp local_file /tmp/mfs
user~: cp /tmp/mfs/file local_destination
```

4.1.3 iRODS

iRODS[9][10] setting up is made with non root user and is interactive. When the script is run, some questions are asked to configure iRODS. We can choose, for example, which server will be the iCat server or where to store data on iRODS servers. iCat server needs *postgresql* and *odcb* to store metadata. Their installation is automatically run during iRODS setting up. However, on grid5000 platform, some downloads are blocked, and we had to retrieve these software manually and put them on the iCat, so that iRODS can detect that a transfer is not needed:

```
icat~: tar xzf postgresql-9.0.3.tar.gz
icat~: mkdir postgresql-9.0.3/src/interfaces/odbc
icat~: cd postgresql-9.0.3/src/interfaces/odbc
icat~: tar xzf /home/cseguin/unixODBC-2.2.12.tar.gz
```

Then, extract the iRODS archive, install and perform some operations:

```
node~: tar xzf irods3.1.tgz
node~: cd iRODS/
node~: ./irodssetup

user~: cd iRODS/clients/icommands/bin
user~: ./iput local_file irods_destination
user~: ./iget irods_file local_destination
```

4.1.4 Ceph

Installation

To set up Ceph, download and install the following package[11] on all nodes (including clients) with root permissions:

```
node~: dpkg -i ceph_0.47.3-1~bpo60+1_amd64.deb
node~: dpkg -i ceph-common_0.47.3-1~bpo60+1_amd64.deb
node~: dpkg -i ceph-fuse_0.47.3-1~bpo60+1_amd64.deb
node~: dpkg -i libcephfs1_0.47.3-1~bpo60+1_amd64.deb
node~: dpkg -i librados2_0.47.3-1~bpo60+1_amd64.deb
node~: dpkg -i librbd1_0.47.3-1~bpo60+1_amd64.deb
node~: apt-get -f install
```

Configuration

Ceph[12] uses a unique config file for all nodes (including clients). Here is the config file used for our tests:

[9] https://www.irods.org/index.php/Downloads
[10] https://www.irods.org/index.php/Installation
[11] http://ceph.com/debian/pool/main/c/ceph/
[12] http://ceph.com/docs/master/start/

```
[global]
 auth supported = none
 keyring = /etc/ceph/keyring

[mon]
 mon data = /tmp/mon.$id
 keyring = /etc/ceph/keyring.$name

[mds]
 keyring = /etc/ceph/keyring.$name

[osd]
 osd data = /tmp/osd.$id
 osd journal = /root/osd.$id.journal
 osd journal size = 1000
 filestore xattr use omap = true
 keyring = /etc/ceph/keyring.$name

[mon."num_mon"]
 host = "mon_hostname"
 mon addr = "ip_mon":6789

[mds."num_mds"]
 host = "mds_hostname"

[osd."num_osd"]
 host = "osd_hostname"
```

For each monitor, metadata server and data server, change *"mon_num"*, *"mds_num"* and *"osd_num"* by the server's number (1, 2, 3 ...). Finally, do not forget to create all directories needed on monitors and data servers:

```
mon~: mkdir /tmp/mon."num_mon"
osd~: mkdir /tmp/osd."num_osd"
```

Mounting ceph on server side

For all the servers, run the following command:

```
server~: mount -o remount,user_xattr /tmp
```

Running Ceph

Choose a main monitor and run the following command from it:

```
mkcephfs -a -c /etc/ceph/ceph.conf -k /etc/ceph/keyring
service ceph -a start
```

Mounting Ceph on client side

From the client mount the ceph file system and perform some operations:

```
user~: mkdir /ceph
user~: ceph-fuse -m $ip_main_mon:6789 /ceph
```

```
user~: cp local_file /ceph
user~: cp /ceph/file local_destination
```

4.1.5 GlusterFS

Installation

First download the glusterFS package[13] for all nodes (including clients) edit the sources.list file and install glusterFS[14] with root permissions:

```
node~: glusterfs_3.3.0-1_amd64.deb
node~: echo "deb http://ftp.de.debian.org/debian sid main" >> /etc/apt/sources.list
node~: apt-get update
node~: dpkg -i glusterfs_3.3.0-1_amd64.deb
node~: apt-get -f -y install
```

Finally, on all servers create a directory in which data will be stored and run GlusterFS:

```
node~: mkdir /tmp/data
node~: /etc/init.d/glusterd start
```

Configuration

First, choose a main server in which create a pool of trusted server (the main server is automatically include in the pool). Then, from the main server, we can create a replicated and/or striped volume. Note that, for n stripe and p replicas, the number of server needed is n $\times$ p:

```
main~: gluster peer probe "server1_hostname"
main~: gluster peer probe "server2_hostname"
...
main~: gluster peer probe "serverN_hostname"

main~: gluster volume create test-volume stripe 4 replica 2 transport tcp
\"main_server_hostname:/tmp/data" ... "server8_hostname:/tmp/data"
main~: gluster volume start test-volume
```

Mounting GlusterFS on client side

To mount the Gluster file system run the following commands from the client and perform some operations:

```
user:~ mkdir /tmp/gluster
user~: mount -t glusterfs "main_server_hostname":/test-volume /tmp/gluster

user~: cp local_file /tmp/gluster
user~: cp /tmp/gluster/file local_destination
```

4.1.6 Lustre

Lustre needs to install[15] a new linux kernel and reboot on it. We had to create a new environment on grid5000 platform which is detailed here:

[13] http://www.gluster.org/download/
[14] http://www.gluster.org/community/documentation/index.php/Main_Page
[15] http://wiki.debian.org/Lustre#Installation_of_Lustre_2.2_in_Debian_Squeeze

Lustre environment on grid5000 platform

On one node, download the following packages[16] [17] and install them with root permissions:

```
node~: dpkg -i ldiskfsprogs_1.42.3-1_amd64.deb
node~: dpkg -i linux-headers-2.6.32+lustre1.8.7-wc+0.
\credativ.squeeze.1_2.6.32+lustre1.8.7-wc1-0.credativ.squeeze.1_amd64.deb
node~: dpkg -i linux-image-2.6.32+lustre1.8.7-wc+0.credativ.squeeze.1_2.6.32
\+lustre1.8.7-wc1-0.credativ.squeeze.1_amd64.deb
node~: dpkg -i lustre-modules-2.6.32+lustre1.8.7-
\wc+0.credativ.squeeze.1_1.8.7-wc1x+dfsg-0.credativ.squeeze.1_amd64.deb
node~: dpkg -i lustre-utils_1.8.7-wc1+dfsg-0.credativ.squeeze.1_amd64.deb
```

Then create an archive of the new environment[18] and modify the config file[19]:

```
node~: mount -o bind / /mnt
toulouse~: ssh root@node.site.grid5000.fr "cd /mnt;
\tar --posix --numeric-owner --one-file-system -zcf - *" > archive.tgz
node~: umount /mnt
toulouse~: kaenv3 -p squeeze-x64-base -u deploy > mysqueeze-x64-base.env
toulouse~: vim mysqueeze-x64-base.env

tarball : archive.tgz|tgz
kernel : /boot/"new_kernel"
initrd : /boot/"new_initrd"
```

Now we can run the new environment on all nodes (including clients) and boot on the new kernel:

```
toulouse~: kadeploy3 -f "list_of_nodes" -a mysqueeze-x64-base.env
```

Finally, on all nodes run:

```
node~: modprobe lnet
node~: modprobe lustre
node~: modprobe ldiskfs
```

Running Lustre

On metadata server side (mds), choose a partition to format and mount it:

```
mds~: mkfs.lustre --fsname=lustrefs --mgs --mdt /dev/sda4
mds~: mkdir /lustre
mds~: mount -t lustre /dev/sda4 /lustre
```

On data server side, choose a partition to format and mount it:

```
osd~: mkfs.lustre --ost --fsname=lustrefs --mgsnode"ip_mds"@tcp0 /dev/sda4
osd~: mkdir /lustre
osd~: mount -t lustre /dev/sda4 /lustre
```

On client side, mount the Lustre file system and perform some operations:

[16] http://pkg-lustre.alioth.debian.org/backports/lustre-2.2.0-squeeze/
[17] http://pkg-lustre.alioth.debian.org/backports/ldiskfsprogs-1.42.3/
[18] https://www.grid5000.fr/mediawiki/index.php/Deploy_environment-OAR2
[19] https://www.grid5000.fr/mediawiki/index.php/Kadeploy-v3

```
user~: mkdir /lustre
user~: mount -t lustre "ip_mds"@tcp0:/lustrefs /lustre

user~: cp local_file /lustre
user~: cp /lustre/file local_destination
```

4.2 System accessibility

In this test, we try to access to a cluster in a private network from a client in another one with only a ssh connection using port forwarding. The following tests are made on client side.

4.2.1 HDFS

We run the following command and then modify the core-site.xml file:

```
user~: ssh -NfL 9000:namenode_hostname:9000 gateway_grid5000

<configuration>
 <property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:9000</value>
 </property>
</configuration>
```

We are able to communicate with the namenode and so perform all the operations related to metadata (`ls`, `stat` ...). However, to write files, we need to forward datanodes ports too which is more complex and we do not try it.

4.2.2 MooseFS

We modify the hosts file and run the following commands:

```
user~: echo "127.0.0.1 mfsmaster" >> /etc/hosts

user~: ssh -NfL 9421:master_hostname:9421 gateway_grid5000
user~: mfsmount /tmp/mfs -H mfsmaster
```

We easily succeed in writing and reading files.

4.2.3 iRODS

We run the following command and then modify the .irodsEnv file:

```
user~: ssh -N -f -L 12477:icat_hostname:1247 gateway_grid5000

user~: vim ~/.irods/.irodsEnv

# iRODS server host name:
irodsHost 'localhost'
# iRODS server port number:
irodsPort 12477
```

We easily succeed in writing and reading files.

4.2.4 Ceph

We run the following command and then modify the ceph.conf file:

```
user~: ssh -N -f -L 6789:main_mon_hostname:6789 gateway_grid5000

user~: vim /etc/ceph.conf

[mon.1]
 host = localhost
 mon addr = 127.0.0.1:6789
```

Currently, we have not succeeded yet to interact with the system from a outside client.

4.2.5 GlusterFS

For GlusterFS, it is harder because several tcp and udp ports are open. We try to redirect all of them without any success.

4.3 System availability

In this test, we examine how the system behaves when one of its nodes crashes. To do that, we first put a data, look at the space disk used and crash a node. Is this node detected as unavailable? Is desired replication satisfied? Are data always available? Is the system balanced when this node is available again? We, now, answer these questions.

4.3.1 HDFS

We use 1 namenode, 5 datanodes, 3 replicas.

- Put a data (34MB):

  ```
  hadoop dfs -put toto toto
  ```

- Space disk used before and after the put:

		Server1	Server2	Server3	Server4	Server5
SD before		189M	189M	189M	189M	189M
Put toto × 3	34M					
SD after		211M	214M	204M	209M	208M
Modification	101M	22M	25M	15M	20M	19M

  The file is replication three times across all the nodes.

- Crash a node:

  ```
  kapower3 -m server1 --off
    The system is going down for system halt NOW!
  ```

- Detection:

  ```
  hadoop dfsadmin -report
    Datanodes available: 4 (5 total, 1 dead)
  ```

The inaccurate node is detected and ten minutes after it is removed from the system.

- Satisfying the replication:

		Server1	Server2	Server3	Server4	Server5
SD	Before crash	211M	214M	204M	209M	208M
Status		KO	Ok	Ok	Ok	Ok
SD	After crash	189M	218M	211M	217M	211M
Modification		-22M	4M	7M	8M	3M

The 22MB lost are recover on other nodes.

- Get a data:

```
hadoop dfs -get toto toto
```

The data is still available.

- Rebooting node:

```
server1:~# hadoop-daemon.sh start datanode
```

- Detection:

```
hadoop dfsadmin -report
  Datanodes available: 5 (5 total, 0 dead)
```

The node is quickly detected.

- Load balancing

		Server1	Server2	Server3	Server4	Server5
SD	Before reboot	189M	218M	211M	217M	211M
Statut		Ok	Ok	Ok	Ok	Ok
SD	After reboot	209M	211M	208M	210M	208M
Modification		20M	-7M	-3M	-7M	-3M

Finally, the system is automatically balanced.

4.3.2 MooseFS

We use 1 master server, 1 backup node, 5 chunk servers, 3 replicas.

- Put a data (34MB):

```
cp  toto /tmp/mfs/toto
```

- Space disk used before and after the put:

		Server1	Server2	Server3	Server4	Server5
SD before		198M	198M	198M	198M	1983M
Put toto × 3	34M					
SD after		211M	234M	198M	221M	233M
Modification	107M	13M	36M	0M	23M	35M

The file is replication three times across four the nodes. We do not know why server3 does not hold any data.

- Crash a node:

```
kapower3 -m server1 --off
  The system is going down for system halt NOW!
```

- Detection: We have used a REST API which provides user with a global system's monitoring. The inaccurate node is detected and removed from the system.

- Satisfying the replication:

		Server1	Server2	Server3	Server4	Server5
SD	Before crash	211M	234M	198M	221M	233M
Status		KO	Ok	Ok	Ok	Ok
SD	After crash	198M	234M	198M	221M	233M
Modification		-13M	0M	0M	0M	0M

The 13MB lost are not recover on other nodes. Replication is not satisfied.

- Get a data:

```
cp /tmp/mfstoto toto
```

The data is still available.

- Rebooting node:

```
server1:~# mfschunkserver start
```

- Detection: Using the REST API, we can see that the node is quickly available again.

- Load balancing

		Server1	Server2	Server3	Server4	Server5
SD	Before reboot	198M	234M	198M	221M	233M
Statut		Ok	Ok	Ok	Ok	Ok
SD	After reboot	211M	234M	198M	221M	233M
Modification		13M	0M	0M	0M	0M

Finally, we do not remove data from the inaccurate server, that is why 13MB are recover. The system is not automatically balanced.

4.3.3 iRODS

We use 1 iCat, 4 iRODS servers.

- Put a data (34MB):

```
iput toto toto
```

- Space disk used before and after the put:

		Server1	Server2	Server3	Server4	Server5
SD before		197M	197M	197M	197M	197M
Put toto	34M					
SD after		231M	197M	197M	197M	197M
Modification	34M	34M	0M	0M	0M	0M

The file is put on one node since iRODS does not split data into blocks.

- Crash a node:

```
kapower3 -m server1 --off
  The system is going down for system halt NOW!
```

- Detection:

```
ips -a
  ERROR: for at least one of the server failed.
```

The inaccurate node is detected and data is lost since there is no replication.

- Rebooting node:

```
server1:~#irodsctl istart
  Starting iRODS server...
```

- Detection:

```
ips -a
  Server: server1
  28237 rods#tempZone  0:00:00  ips  192.168.159.117
```

The node is quickly detected and if data are not removed from disk, they are available again.

4.3.4 Ceph

We use 2 mds, 2 mon, 3 osd, 2 replicas.

- Put a data (34MB):

```
cp toto /ceph/toto
```

- Space disk used before and after the put:

		Server1	Server2	Server3
SD before		203M	204M	204M
Put toto × 2	34M			
SD after		218M	238M	228M
Modification	73M	15M	34M	24M

The file is replication twice across all the nodes.

- Crash a node:

```
kapower3 -m server1 --off
  The system is going down for system halt NOW!
```

- Detection:

```
ceph -s
  osd : 3 osds: 2 up, 2 in
```

The inaccurate node is detected and removed from the system.

- Satisfying the replication:

		Server1	Server2	Server3
SD	Before crash	218M	238M	228M
Status		KO	Ok	Ok
SD	After crash	203M	242M	239M
Modification		-15M	4M	11M

The 15MB lost are recover on other nodes.

- Get a data:

```
cp /ceph/toto toto
```

The data is still available.

- Rebooting node:

```
server1:~#service ceph -a start
```

- Detection:

```
ceph -s
  osd : 3 osds: 3 up, 3 in
```

The node is quickly detected.

- Load balancing

		Server1	Server2	Server3
SD	Before reboot	203M	242M	239M
Statut		Ok	Ok	Ok
SD	After reboot	223M	230M	231M
Modification		20M	-12M	-8M

Finally, the system is automatically balanced.

4.3.5 GlusterFS

We use 4 servers, 2 stripes and 2 replicas.

- Put a data (34MB):

```
cp toto /tmp/gluster/toto
```

- Space disk used before and after the put:

		Server1	Server2	Server3	Server4
SD before		201M	201M	201M	201M
Put toto × 2	34M				
SD after		218M	218M	218M	218M
Modification	68M	17M	17M	17M	17M

  The file is replicated twice across all the nodes.

- Crash a node:

```
kapower3 -m server1 --off
  The system is going down for system halt NOW!
```

- Detection:

```
gluster volume status test-volume
  Status of volume: test-volume
  Gluster process Port Online Pid
  ------------------------------------------------------------
  Brick server2:/tmp/data 24009 Y 4861
  Brick server3:/tmp/data 24009 Y 5055
  Brick server4:/tmp/data 24009 Y 5251
```

  The inaccurate node is detected and removed from the system.

- Satisfying the replication:

		Server1	Server2	Server3	Server4
SD	Before crash	218M	218M	218M	218M
Status		KO	Ok	Ok	Ok
SD	After crash	201M	218M	218M	218M
Modification		-17M	0M	0M	0M

  The 17MB lost are not recover on other nodes.

- Get a data:

```
cp /tmp/gluster/toto toto
```

  The data is still available.

- Rebooting node:

```
server1:~#/etc/init.d/glusterd start
  Starting glusterd service: glusterd.
```

- Detection:

```
gluster volume status test-volume
  Status of volume: test-volume
  Gluster process Port Online Pid
  ------------------------------------------------------------
  Brick server1:/tmp/data 24009 Y 4891
  Brick server2:/tmp/data 24009 Y 4861
  Brick server3:/tmp/data 24009 Y 5055
  Brick server4:/tmp/data 24009 Y 5251
```

  The node is quickly detected.

- Load balancing

		Server1	Server2	Server3	Server4
SD	Before reboot	201M	218M	218M	218M
Statut		Ok	Ok	Ok	Ok
SD	After reboot	218M	218M	218M	218M
Modification		17M	0M	0M	0M

  Finally, if data are not removed from the inaccurate disk, they are available again, but the system is not automatically balanced.

4.3.6 Lustre

We use 1 mds, 4 data servers, no replica.

- Put a data (34MB):

```
cp toto /lustre/toto
```

- Space disk used before and after the put:

		Server1	Server2	Server3	Server4
SD before		482M	482M	482M	482
Put toto	34M				
SD after		491M	491M	491M	491M
Modification	36M	9M	9M	9M	9M

  The file is striped across all the nodes.

- Crash a node:

```
kapower3 -m server1 --off
  The system is going down for system halt NOW!
```

- Detection:

```
lfs check servers
  lustrefs-OST0000: check error
```

  The inaccurate node is detected.

- Get a data:

```
cp /lustre/toto toto
  Input/Output error
```

  The data is not available.

- Rebooting node:

```
server1:~#mount -t lustre /dev/sda4 /lustre
```

- Detection:

```
lfs check servers
  lustrefs-OST0000: active
```

  The node is quickly detected.

- Load balancing

		Server1	Server2	Server3	Server4
SD	Before reboot	482M	491M	491M	491M
Statut		Ok	Ok	Ok	Ok
SD	After reboot	491M	491M	491M	491M
Modification		9M	0M	0M	0M

  Finally, if data are not removed from disk, they are available again, but the system is not automatically balanced.

4.4 System performance

In addition to the tests introduced in Section 3.1.2, we have perform another simple test on grid5000 platform [20] on *the pastel cluster* [21]. We have measured the time to put (write) and get (read) a 20GB data and one thousand of 1MB data with two replicas and compare with the results obtained with one replica. We have used two metadata servers for Ceph to benefit from metadata distribution, four data servers, one client and 2 replicas. The tables 4.2 shows the results for read and write operations with 1 and 2 replicas. Red color indicates the best result while the blue one indicates the worst results.

iRODS' performance decrease for writing small and big files with 2 replicas, time is doubled. This is due to the replication operation which is identical to perform another put. On its side, Ceph decrease for writing big files may be due to the synchronous replication Performance of the other DFSs do not dramatically decrease. Note that no consequent modification is noticed about reads performance but it could be interesting to test the system with concurrent access and so with more users.

Table 4.2: Input and Output performances with 1 and 2 replicas

	HDFS		iRODS		Ceph		GlusterFS		MooseFS	
Input/Output	I	O	I	O	I	O	I	O	I	O
1 × 20GB	407s	401s	**520s**	**500s**	419s	**382s**	**341s**	403s	448s	385s
2 × 20GB	626s	422s	**1070s**	468s	873s	**495s**	**426s**	**385s**	504s	478s
1000 × 1MB	72s	17s	**86s**	**23s**	76s	21s	**59s**	18s	68s	**4s**
2 × 1000 × 1MB	96s	17s	**179s**	20s	**85s**	**23s**	86s	17s	89s	**4s**

[20] https://www.grid5000.fr
[21] https://www.grid5000.fr/mediawiki/index.php/Toulouse:Hardware

Chapter 5

Conclusion

DFSs are the principle storage solution used by supercomputers, clusters and datacenters. In this paper, we have given a presentation and a comparison of five DFSs based on scalability, transparency and fault tolerance. DFSs surveyed are : Lustre, HDFS, Ceph, MooseFS, GlusterFS and iRODS. We have seen that the DFSs ensure transparency and fault tolerance using different methods that provide the same results. The main difference lies on the design. In theory, decentralised architectures seem to scale better than a centralised one thanks to the distributed workload management. Furthermore, the choice of a DFS should be done according to their use. For performance, an asynchronous replication and the use of an index to maintain the namespace are preferable whereas a decentralised architecture is better for managing large amounts of data and requests. The comparison given in this paper is theoretical. However we have performed some simple tests to measure the system accessibility and fault tolerance. We try to access to a cluster in a private network from another one with only a ssh connection. Using port forwarding, we have conclude that only iRODS and MooseFS are is easily accessible. About fault tolerance, we just have simulated a crash on a data server. For all DFSs, except Lustre, the inaccurate server is detected and put in quarantine in a transparent way. The desired number of replicas is maintained except for GlusterFS and iRODS. We hope to perform stronger tests in future to provide a practical analysis. In particular, measuring scalability and limits of metadata server(s) by stressing them, that is, sending several requests. Asynchronous and synchronous I/O operations can also be compared. Finally, testing fault tolerance in a more thorough way is needed.

Acknowledgment

This work was developed with financial support from the ANR (Agence Nationale de la Recherche) through the SOP project referenced 11-INFR-001-04.

Bibliography

[1] Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symp. on Mass Storage Systems and Technologies (MSST), Washington, DC, USA, IEEE Computer Society

[2] Rajasekar, A., Moore, R., Hou, C.y., Lee, C.A., Marciano, R., de Torcy, A., Wan, M., Schroeder, W., Chen, S.Y., Gilbert, L., Tooby, P., Zhu, B.: iRODS Primer: integrated Rule-Oriented Data System. Morgan and Claypool Publishers (2010)

[3] Wan, M., Moore, R., Rajasekar, A.: Integration of cloud storage with data grids. In: Proc. Third Int. Conf. on the Virtual Computing Initiative. (2009)

[4] Hünich, D., Müller-Pfefferkorn, R.: Managing large datasets with irods - a performance analyses. In: Int. Multiconf. on Computer Science and Information Technology - IMCSIT 2010, Wisla, Poland, 18-20 October 2010, Proceedings. (2010) 647–654

[5] Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: A scalable, high-performance distributed file system. In: In Proceedings of the 7th Symp. on Operating Systems Design and Implementation (OSDI). (2006) 307–320

[6] Weil, S.A.: Ceph: reliable, scalable, and high-performance distributed storage. PhD thesis, Santa Cruz, CA, USA (2007)

[7] Weil, S., Brandt, S.A., Miller, E.L., Maltzahn, C.: Crush: Controlled, scalable, decentralized placement of replicated data. In: Proceedings of SC '06. (nov 2006)

[8] Gluster: An Introduction to Gluster Architecture (2011)

[9] Gluster: Performance in a Gluster System (2011)

[10] Schwan, P.: Lustre: Building a file system for 1000-node clusters. In: Proceedings of the 2003 Linux Symp. Volume 2003. (2003)

[11] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0 (Nov 2002)

[12] Braam, P.J., Others: The Lustre storage architecture. White Paper, Cluster File Systems, Inc. **23** (2003)

[13] Sun Microsystems, Inc., Santa Clara, CA, USA: Lustre file system - High-performance storage architecture and scalable cluster file system (2007)

[14] Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T., Huang, I.: Understanding lustre filesystem internals. Technical Report ORNL/TM-2009/117, Oak Ridge National Lab., National Center for Computational Sciences (2009)

[15] Levy, E., Silberschatz, A.: Distributed file systems: concepts and examples. ACM Comput. Surv. **22**(4) (dec 1990) 321–374

[16] Thanh, T.D., Mohan, S., Choi, E., Kim, S., Kim, P.: A taxonomy and survey on distributed file systems. In: Proceedings of the 2008 Fourth Int. Conf. on Networked Computing and Advanced Information Management. NCM '08, Washington, DC, USA, IEEE Computer Society (2008) 144–149

[17] Nicolae, B., Antoniu, G., Bougé, L., Moise, D., Carpen-amarie, R.: Blobseer: Next-generation data management for large scale infrastructures. J. Parallel Distrib. Comput (2011) 169–184

[18] Nelson, M.N., Welch, B.B., Ousterhout, J.K.: Caching in the sprite network file system. ACM Trans. Comput. Syst. **6**(1) (feb 1988) 134–154

[19] Gray, C., Cheriton, D.: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. SIGOPS Oper. Syst. Rev. **23**(5) (nov 1989) 202–210

[20] Satyanarayanan, M.: A survey of distributed file systems. In: Annual Review of Computer Science. (1989)

[21] Songlin Bai, H.W.: The performance study on several distributed file systems. In: Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 Int. Conf. on. (2011)