Using Object-Based Storage to Build Distributed File Systems: a Survey

Bibliographical study

Viet-Trung TRAN
viet-trung.tran@etudiant.univ-rennel.fr

Supervisors: **Gabriel Antoniu, Luc Bougé**
{Gabriel.Antoniu,Luc.Bouge}@irisa.fr

ENS de Cachan, IFSIC, IRISA, Paris Project
February 2009

Keywords: Distributed File Systems, Object-based Storage, Massive Data, Fine-grain Access, Versionning.

Contents

1 Introduction

Nowadays, research in data management has made a lot of progress in all of its aspects. From local data management with low concurrency control, sequential access in the primitive times, we arrive now to distributed data management, parallel access and high concurrency. Obviously, this progress comes from the requirements of data processing, where performance depends on the data management techniques that are used.

In spite of the progress made in any aspects, almost all existing data management techniques have not change in the ways of physically storing data on disks. In fact, current block-based interfaces of storage are holding system designers back. This means data management has to take care of each data block and this increases the complexity of data management system. Recent industry and academic research suggest a shift in storage technology, from relatively unintelligent and externally-managed devices to intelligent, self-managed devices aware of the storage applications they serve.

This report studies recent solutions based on the concept of object-based storage. It is a survey of current state of the art in the design of distributed file systems using object-based storage devices (OSDs) and high-level object-based approaches.

Section 2 takes a look at distributed file systems that do not rely on objects. Section 3 introduces object-based storage architectures. It also discusses some efforts which try to integrate OSDs into file systems. Section 4 presents another approach in distributed file systems which store data as objects but at a higher level.

2 Distributed file systems

The purpose of a distributed file system (DFS) is to allow users to share distributed data and storage resources by using a common file system interface [9]. The system consists of a set of distributed components interconnected to each other. In addition, it aims to provide a global view to enable data sharing in spite of the physical data distribution. Most of existing DFS today usually try to achieve the following goals:

Uniform access to data from multiple sites

Distributed processes access shared data in the same way, without having to explicitly move data. Beside, if some of the processes migrate to different places, they keep accessing the data the same way.

Provide access transparency

Because of the importance of "sharing", much effort aims to improve this function in DFS, to simplify to share and how to retrieve shared data. This leads to achievement of a fundamental property, called *network transparency*, which implies that clients should be able to access remote file using the same set of file operations applicable to local files. In a distributed context, DFS might export file systems interface at two levels of transparency : *location transparency* and *location independence*. *Location transparency* means that the name of the file does not reveal any hint as to its physical storage location. Almost all file systems today (NFS, PVFS, ...) have this kind of transparency. A highest level of file name abstraction, *location independence* requires that the file name doesn't need to be changed when the file's physical storage location changes.

Improve performance

The best performance metric of a DFS is the amount of time needed so satisfy service requests. The distributed structure has some impact on the total data access time. But thanks to high-speed networks (Gb) and to massive parallel data processing techniques, DFS can be more efficient.

Improve reliability

In a distributed environment, the reliability was one of the primitive challenges for DFS because the large number of components may be volatile or subject to failures. DFS, however, can achieve reliability by using some fault tolerance techniques such as *replication* and *erasure coded*[1].

With these goals, DFS today provide various features which can be considered as challenges for DFS design: security, availability, consistency management, recovery, support for versionning and snapshoting, scalability, and performance [13]. These features are covered in the following subsection.

2.1 Features and trade-offs

Security

Security was not be the first priority for DFS design in the past. But, due to the increasing scale of file systems, which can spread over a open networks (Internet), security issues become important. In Grid environment, DFS must provide *authentication*. Furthermore, once users are authenticated, the systems must ensure that the performed operations are permitted on the resources accessed. This process is called *authorization*. The secure connection for accessing remote file must also be taken into account.

Availability

There is a growing need for distributed file systems that are highly resilient to failures. Two mechanisms, replication and recovery, are used to make data highly available despite server or network failures. Replication leads to the problem of replica consistency maintaining. On the other hand, recovery techniques like implementing *erasure code* have affects that reduce *Performance*. Thus, the trade-off between *Availability*, *Performance*, and *Consistency management* needs to be considered. For the best design, data must be continuously and transparently available to all clients with minimum performance penalty, high consistency semantics.

Consistency management

Consistency management is one of the most complex components in the file system stack while using caches for better data access. File system designer needs to find the right trade-off between the *semantics of sharing* implemented by *file locking mechanism*, the sharing granularity, and the complexity of *consistency management*.

Recovery

A recovery mechanism tries to ensure, as much as possible, that the file system is in a consistent state and has not lost data after failure. There are two main approaches.

[1] An erasure code transforms a message of n blocks into a message with more than n blocks, such that the original message can be recovered from a subset of those blocks.

The first is to use checkpoints to always maintain a recent consistent snapshot of the file system, the second is to replicate information to more than one file server.

Support for versioning and snapshoting

These two techniques give the user access to multiple successive versions (instances) of the files.

Scalability

Scalability is the ability of the file system to provide acceptable performance for any requested file system operation as the number of clients served by the network increases. Scalability is the most obvious problem of centralized network file systems. At present, we target a file system design for the Grid scale.

2.2 State of the art: Some examples

PVFS

PVFS [3] stands for Parallel Virtual File System developed and maintained at Argonne National Labs and Clemson University. At present, PVFSv2 allows servers to be configured to be data and/or metadata servers. But it currently supports only one metadata server. Data is typically striped among the data servers in 64KByte chunks (RAID-0). PVFS is tightly integrated into MPI-IO. Thus, the clients can describe complicated access patterns that extract subsets of large datasets. PVFS was designed for performance of parallel access, but it has a inherent weakness on reliability because of the lack of fault tolerance techniques.

kDFS

kDFS [8] stands for kernel distributed file system and is designed for cluster environment. Exploited kernel distributed data manager service [8], kDFS gives system services the illusion that the cluster physical memory is a shared memory. Therefore, kDFS fully supports POSIX API.

In contrast with PVFS, kDFS doesn't distinguish storage nodes and compute nodes. kDFS makes a better usage of the available storage resources and throughput capacities. At present, kDFS intends to support "file stripping" and "replication" techniques.

Gfarm

Gfarm [1] is designed for facilitating reliable file sharing and high-performance distributed and parallel data computing in a Grid across administrative domains by providing a global virtual file system. In its first version, Gfarm considers a grid file system as a cluster of cluster file system in which metaserver manages metadatas at each site.

3 Toward object-based file systems using object-based storage devices

Most file systems today relies on block-based storage. In this way, the file system interact directly with a storage device via block-based interface such as SCSI, ATA/IDE or fiber channel to manipulate data. However, because of its limited operations, block-based interface is now becoming a limiting factor for many storage architectures, and thus, file system architectures.

The two DFS architectures in common use today rely on message-passing and on direct access. The direct access model used in Storage area networks (SANs) offers high performance by exploiting dedicated hardware protocols like SCSI or fiber channel. Because of permitting direct access to block-base storage, hosts in SAN need to share *metadata* which is used for mapping from data structures (files and directories) to blocks, and do so in a manner that guarantees metadata consistency. The complexity of this process is a real challenge for scalability because SANs operate at block-level, making it difficult for data sharing between applications. Furthermore, at present, most SAN solutions are proprietary and expensive.

In the message-passing model used on Network-attached storages (NASs), the Server node manages all the metadata and exposes its file system to the clients. This level of indirection enables cross-platform data sharing, however, at the cost of directing all I/O through the single file server. NAS may be implemented on top of a SAN to form a *NAS heads* (Figure 1). In either case, the file server will be a single point of failure (SPoF) which limits the performance of its file system. Neither NAS or NAS head, can easily provide the aggregate performance of storage devices.

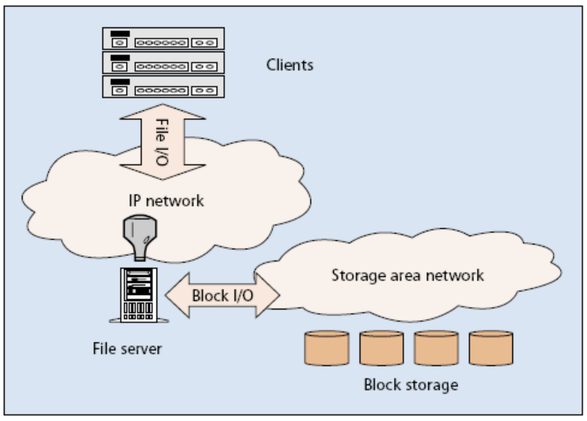


Figure 1: Network-attached storage architecture (Extracted from [10]).

SAN file system, a hybrid architecture, has recently been introduced in an attempt to capture the features of both NAS and SAN. In a SAN file system (Figure 2), the file server and clients are all connected to a SAN on which the file system is stored. The main point is that file servers hold only metadata and its clients access the storage devices directly. Given this approach, workload on metadata servers is reduced but the security was ignored. The SAN mechanisms for device security only protect the physical device, not the data contained in it.

In order to benefit from high performance (blocks), security and cross data sharing (files), a new storage design is required which provides both the performance of direct access to disk and the ease of administration provided by shared files and metadata. The new storage system design is Object-based Storage Architecture.

3.1 Object-based Storage Devices (OSDs)

The Object-based storage device (OSD) [10, 5] is an intelligent evolution of today's disk drive which is capable of storing and serving objects rather than simply putting data on tracks and sectors. The difference between an OSD and a block-based device is the interface, not

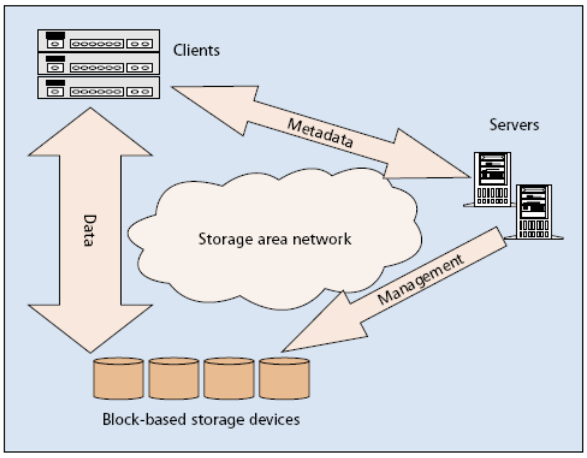


Figure 2: SAN file system (Extracted from [10]).

the physical media. Unlike block I/O, creating objects on a storage device is accomplished through a rich interface similar to a file system. The OSD interface standard [17] permits space management (i.e. allocation and tracking of used and free blocks) to be moved into storage devices.

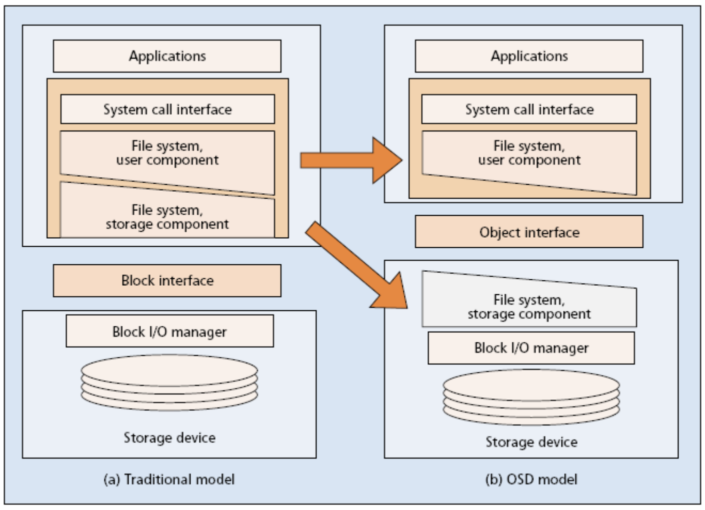


Figure 3: Offloading of storage management from the file system (Extracted from [10]).

Offloading the space management from file system to storage device can be illustrated in Figure 3. In the OSD model, the file system storage component which maps the data structures to the physical storage is offloaded to the storage device, and the device interface transform blocks into objects. OSD model removes the dependency between block metadata and storage application, e.g. file system, thus leading to the feasibility and scalability of data sharing. For example, the consistency of block metadata shared by hosts in SAN is a real matter because it restricts the scalability of file system.

Furthermore, OSD organizes data structures in its metadata as single objects. Each object is composed of data, user-accessible attributes, and device-managed metadata. This way, OSD is capable of setting security policies on a per-object basic, similar to the manner in which files can be protected by a file sever. It is also equipped with the capability of self-management because OSD is aware of its contents via objects attributes. Self-management includes functions such as reorganizing data to improve performance, scheduling regular backups, and recovering from failures.

In brief, OSD has some following advantages :

- Improved storage management: self-managed, policy-driven storage (e.g. backup, recovery)
- Improved device and data sharing: security, shared devices and data across OS platforms
- Improved storage performance: hints, quality of service (QoS), differentiated services (by using user-accessible attributes).
- Improved scalability: devices directly handle client requests (by offloading block metadata).

3.2 OSD-based file systems

Using OSDs requires that OSD-based file system is restructured to fit the new storage architecture. At present, object-based file systems for distributed environments rely mostly on the common architecture which consists of three following components (Figure 4):

Metadata servers
Metadata servers maintain all file system metadata. This includes file system structures, the mapping from file to objects, and the current locations of objects.

Object-based storage devices
OSDs provide object-based IO service. OSDs are fully responsible for managing data blocks.

Clients
The file system client accesses file content directly via OSDs. Metadata servers are only inquired once when the file is opened, thereby eliminating the metadata bottleneck of block-based distributed file systems. Remember that, this architecture is not the same with SAN file system in which clients access IO nodes via block-based interface.

Based on this architecture, each file system may be supplied some additional tasks for metadata servers according the proprietary design. In the proposed OSD-based file system, OSDs are replaced by object storage servers because no OSD implementation has been proposed by industry so far.

3.3 Case studies

In this section, we present some OSD-based file system implementations.

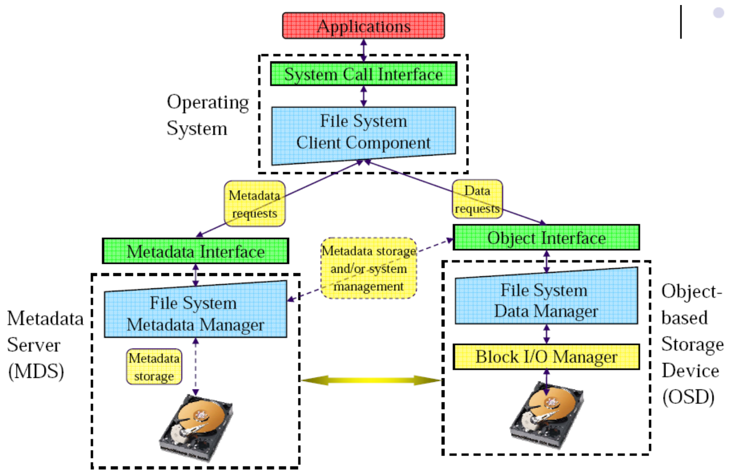


Figure 4: Distributed OSD-based file system architecture

3.3.1 Integrating parallel virtual file system with OSDs

Observe that OSDs can potentially bring back certain powerful advantages. Integrating parallel virtual file system (PVFS) with OSDs [4] was made in an effort to examine the feasibility and the benefits of OSDs for use in parallel file system. To construct such implementation, the I/O storage servers are replaced with software-emulated OSDs in the layout of file system architecture. Software-emulated OSD is standard compliant with OSD T10 specification [17] for a long vision that PVFS aims to exploit true OSDs in the future. Figure 5 shows the software architecture of PVFS using OSDs.

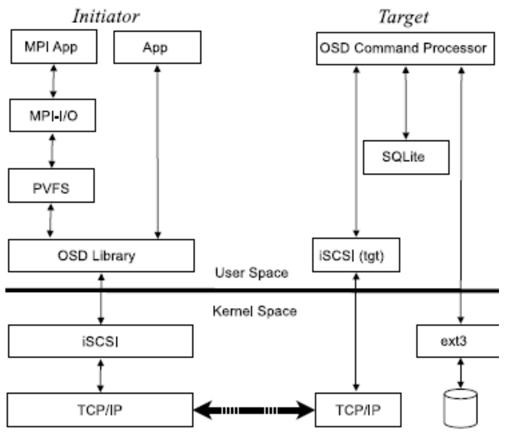


Figure 5: Software architecture of PVFS using OSDs.

In this system design, client communicates with OSDs using iSCSI [17]. The iSCSI protocol enables the transport of SCSI commands (here, OSD command sets) over a networks, usually TCP/IP. The existing Linux in-kernel iSCSI initiator is used on client's side, but on emulated OSD's side, *tgt* [15], a user-space iSCSI target implementation is used because of the lack of iSCSI target support in Linux. On top of iSCSI components, the OSD initiator library exports the interface used by client applications (e.g. PVFS) to communicate with the

OSD target (OSD command processor). The OSD command processor is responsible for manipulation, creation and destruction of objects and their attributes. Currently, emulated OSD rely on an underlying local file system to store objects for the simplicity. Thus, it implements one-to-one mapping between user objects and files. Moreover, a *SQlite database* is used for fast look-ups and flexible manipulation of attributes.

In this architecture, instead of sending requests to the PVFS IO servers, clients must generate SCSI commands and send them to an OSD server. The communication therefore changes from remote procedure calls to the iSCSI protocol. These are the main modifications made to PVFS to adapt it for OSDs. Other techniques like file stripping are preserved. The novel architecture also preserves two active metadata servers to share the load for operations such as file look-ups, creates, and directory traversals.

3.3.2 Lustre file system

Lustre [14] was started in 1999 with the goal of removing the bottlenecks traditionally found in NAS architectures, providing the scalability, performance, and fault tolerance for file system in cluster environment. Lustre is one of the first file system based on OSDs approach.

In a Lustre file system, Object-based storage target (OST) is only attached to the server nodes that are emulated-OSDs. If failover capability is desired, storage must be attached to multiple servers. A SAN can also act as OST in Lustre to enable failover, and high-performance feature.

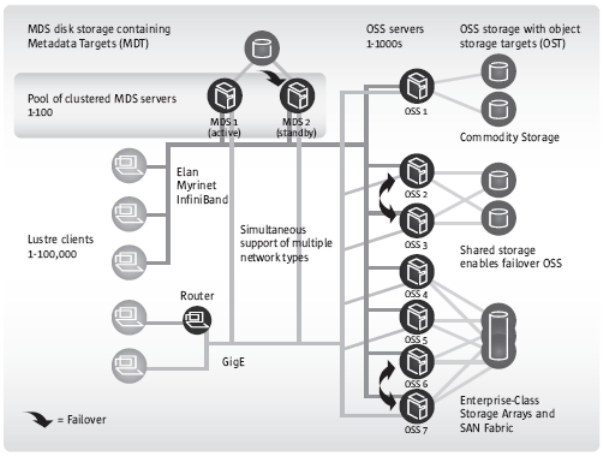


Figure 6: Lustre file system

Lustre file system designed typically uses two metadata servers sharing the same metadata target (MDT). One is in active mode, and the second one is in standby mode. This is a bit different in comparison with PVFS using OSDs. By this way, Lustre doesn't need to take care of metadata consistency.

Lustre supports strong file and metadata locking semantics to maintain coherency of the file systems even under a high volume of concurrent access. File locking is distributed across the Object Storage Targets (OSTs) that constitute the file system, with each OST managing locks for the objects that it stores.

3.3.3 Ceph file system

Ceph [18] was recently developed at University of California, Santa Cruz. It is yet another distributed file system using OSDs. The primary goals of Ceph architecture are scalability, performance, and reliability. The Ceph file system has three main type of components: clients, a cluster of OSDs which collectively stores all data and metadata, and a metadata cluster which manages the namespace, security, consistency and coherence (Figure 7).

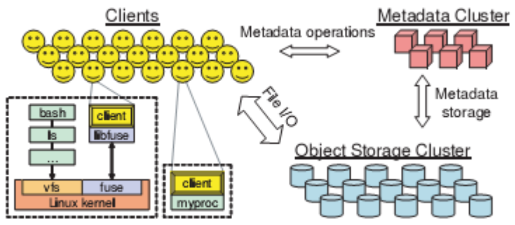


Figure 7: Ceph file system architecture.

One novel approach in Ceph is that the metadata cluster architecture was based on dynamic subtree partitioning [19] which manages dynamically distributed metadata. Using metadata cluser instead of single metadata server, Ceph permits to potentially achieve workload balancing, and avoid the single point of failure of metadata management. Furthermore, in contrast to existing object-based file systems [14, 6], Ceph clients don't need to request metadata server to look up object location lists. They calculate directly (rather than look up) the name and location of objects comprising a file's contents instead. This design eliminates the need to maintain and distribute object lists, and so then reduces the metadata cluster workload.

OSD design is another point important of Ceph. Ceph file system tries to exploit self-managed, intelligence property of OSD architecture. OSDs were delegated the responsibility for data migration, replication, failure detection, and failure recovery. Ceph groups OSDs by failure domain. Files are striped across many objects, grouped into *placement groups* (PGs), and distributed to list of OSDs which are in different failure domains. So then, the OSDs that share PGs made a group in which file replication, failure detection are self-managed. To facilitate fast failure recovery, OSDs maintain a version number for each object and a log for recent changes for each PG. After failure or after booting time, an OSD could determine the latest version of any its object for recovery by communicating with other OSDs in group.

In addition, Ceph uses an Extent and B-tree based Object-File System (EBOFS) instead of ext3 to manage low-level storage in order to improve performance suited for object workloads.

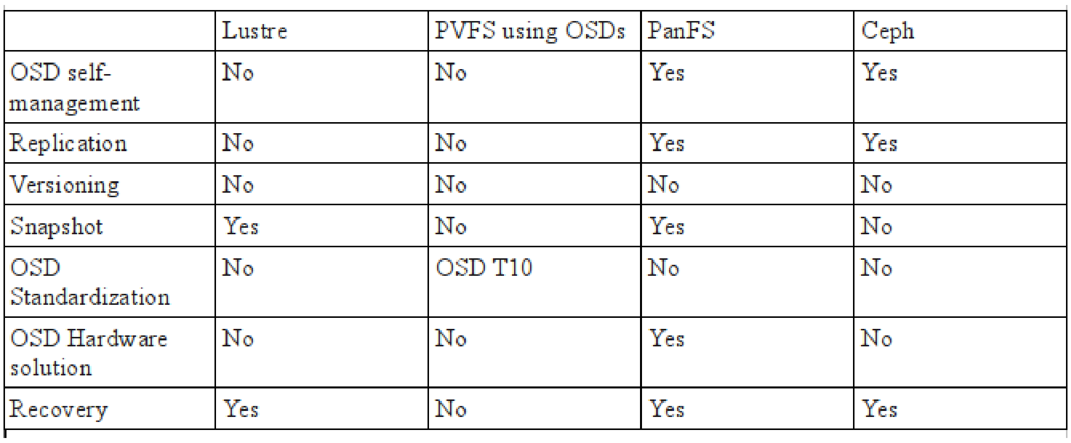
3.3.4 PanFS parallel file system

The PanFS [16] architecture's similar to the common architecture of OSD-based file system. PanFS consists three main components such as clients, metadata server (DirectorBlade Cluster) and OSDs (StorageBlade Cluster).

3.4 Discussion

The concept of object storage was introduced in the early 1990's by the research community. Since then it has greatly matured, but yet OSD is still an active area of research in academia. As for the industry, while there is significant industry involvement in the standardization effort for object storage such as OSD T10 standard [17], pNFS extension for NFSv4 [2], the actual adoption is low and most current object stores use proprietary interfaces. At present, only PVFS among all mentioned file system, tries to design OSD architecture which conform with OSD standardization. Likewise, Panasas is actively involved in the standardization effort and has expressed the intent of moving toward the standard object store.

A summary of the features proposed by OSD-based file system is presented in Figure 8. While PanFS offers a server hardware solution by using smart object iSCSI storage devices (Panasas Storageblade Cluster), other file system such as Lustre [14], Ceph [18], and integrating PVFS with OSDs [4] offer storage servers as emulated-OSDs. PanFS may be get better performance of OSDs, but it comes however with a expensive cost. Ceph file system presented the most powerful feature of OSD architecture as self-management. But, consequently, Ceph's design is the most complex. Using a metadata cluster requires to take care seriously of consistency. While almost all workloads is offloaded from metadata server to storage nodes, one question is: are the benefits worth the complexity of design? In addition, Ceph is work in progress. A lot of challenges and evaluations still need to be performed.



	Lustre	PVFS using OSDs	PanFS	Ceph
OSD self-management	No	No	Yes	Yes
Replication	No	No	Yes	Yes
Versioning	No	No	No	No
Snapshot	Yes	No	Yes	No
OSD Standardization	No	OSD T10	No	No
OSD Hardware solution	No	No	Yes	No
Recovery	Yes	No	Yes	Yes

Figure 8: Object-based file systems compared.

4 Using objects in file systems at a higher level

All of DFS that I mentioned above in Section 3 proposed DFS architecture with a simulation version of OSD storage (in many ways). Such trends are relatively complex and require a lot of efforts. Moreover, beside such trends, there exist already certain DFS (GoogleFS, HadoopFS, ...) which were proposed and implemented before the birthdate of some current OSD standardization (OSD T10 [17], pNFS extension for NFSv4 [2]). In their approach, they don't try to emulate OSD storage, to exploit all new features of OSD architecture, but they really store data as objects. According some proprietary aims, these DFSs already meet its functional requirements, with a simpler design and lower cost.

4.1 Case studies

4.1.1 GoogleFS

Google file system (GFS) was designed to meet the rapidly growing requirements of Google's data processing. Some key observations of Google application workloads and technological environment offered both challenges and opportunities to implement GFS. GFS has to cope with the volatile hardware components, the huge data processing, the concurrent access from multiple clients, and the fact that most files are mutated by appending new data rather than overwriting existing data. Taking into account these demands, GFS simplifies its design in holding the same goals as other DFS such as performance, scalability, reliability, and availability.

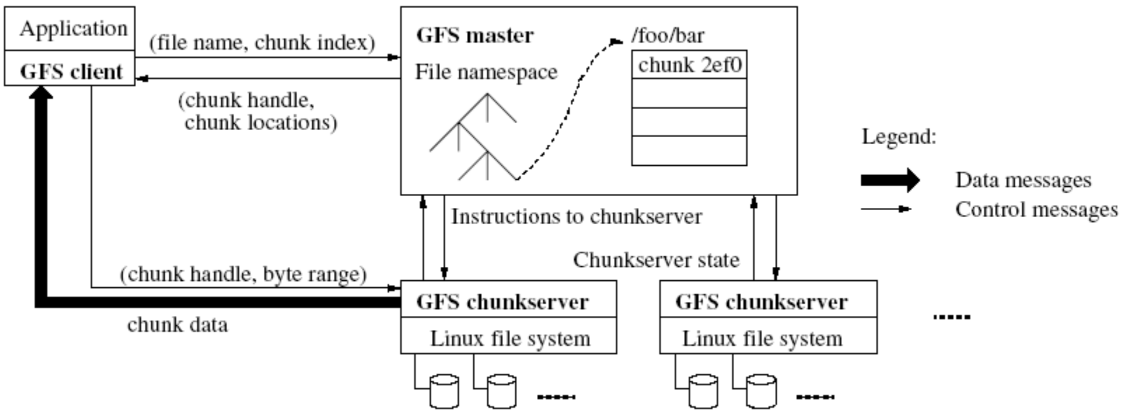


Figure 9: Google file system.

GFS consists of a single master and multiple chunkservers and is accessed by multiple clients, as show in Figure 9. Files are broken into fixed-size chunks 64 MB which are distributed over chunkservers. Each chunk is replicated on multiple servers, typically on 3 servers for reliability. The chunkservers use the underlying file system ext3 to store chunks.

The master maintains all file system metadata such as the namespace, the mapping from files to chunks, access control information, and the current locations of chunks in main memory in order to speed up master operations. It also doesn't intend to keep a persistent record of chunk location for simplifying the design. To know about chunk locations, the master simply polls chunkservers for that information at startup. The master thereafter stays up to date by controlling placement of new chunks and through Heartbeat messages (when monitoring chunkservers).

The clients use directly the file system API instead of the standard POSIX API. To minimize the number of requests sent to the master, the clients cache only metatdata (e.g. chunk location). Remember that neither the client nor the chunkserver caches file data because this is not effective in GFS design.

According some key assumptions of environment, GFS designers propose a consistency model which is a bit different to other models.

Consistency model
GFS has a relaxed consistency model. Since clients cache chunk locations, they may read

from a stale replica before that information is refreshed. GFS uses cache entry's timeout to limited this inconsistency. File namespace is handled exclusively by the master, so then it's guaranteed in consistent state.

On other hand, the mutated file is consistent and contains data written by the last mutation after a series of data mutations. This is done by a strategy in which (1) the mutations are applied to replicas in same order, (2) the chunk version numbers are used to detect stale replicas, (3) the mutations are never applied to stale replicas, (4) the master never gives client location of a stale replica, and (5) the master controls garbage collection of stale replica.

Some features
Recovery and replication are presented in GFS to keep it highly available. Both of chunks and metadatas are replicated. Each chunk is replicated on 3 chunkserver in default, but users can specify different replication levels for different files. GFS uses a complementary "shadow" master server which only provide read-only access to the file system when the primary master is down. Fast recovery is achieved by using operation logs and snapshot.

4.1.2 HadoopFS

Hadoop's Distributed File System [7] is designed to reliably store very large files across machines in a large cluster. It is inspired by the GFS [6]. Hadoop DFS stores each file as a sequence of blocks, all blocks in a file except the last block are the same size. Blocks belonging to a file are replicated for fault tolerance. The block size and replication factor are configurable per file. Files in HDFS are "write once" and have strictly one writer at any time. This assumption simplifies data coherency issues and enables high throughput data access.

Like GFS, HadoopFS offers some features such as replication and recovery. But it is open source in contrast to the other. Given the assumption: "moving computation is cheaper than moving data in massive data processing", HadoopFS in addition, provides interfaces for applications to move themselves closer to where the data is located.

4.2 Discussion

Both of GoogleFS and HadoopFS have some relaxed constrains in their designs. The object size when dealing with chunks and blocks is fixed, whereas object-based file system can take the advantage of variable object size in their designs. The storage servers are responsible solely for storing objects. Some features such as replication and recovery techniques are implemented by metadata server. In this context, the storage servers are "dumb" compared to the OSD specifications [17]. Furthermore, the additional metadata overhead limits the scalability of these file systems.

5 Conclusion

This bibliographical study focused on the survey of Object-based file systems emerged as one tendency of file system design nowadays. We presented the strict moving from block-based interface to Object-based interface in storage architecture. The new interface, as a result potentially simplifies the design of file system, offloads almost management tasks to storage, and eliminates the trade-offs between security, data sharing and high performance. However, more works need to be done before we can introduce OSDs in industry.

We also presented several Object-based file system implementations, tried to group them in two classes. The first one exploits more about the capacity of OSD architecture but comes at a higher complexity. In contrast, given some assumptions, some relaxed constrains, the second one satisfies the demands while having a simpler design.

Furthermore, large-scale data management using objects likes object-based file system is an active research area. Outside the area of file system, other efforts consider using objects for large-scale data management without exposing a file system interface. Blobseer [11, 12] is such data management service.

BlobSeer [11, 12] is a blob (binary large object) management service specifically designed to deal with the dynamics of large scale distributed applications, who need read and update massive data amounts over very short periods of time. In this context, support is required for a large number of blobs, each of whom might reach a size in the order of TB. BlobSeer employs a powerful concurrency management scheme enabling a high number of clients to read and update the same blob simultaneously in a lock-free manner. Clients are offered transparent versioning and fine grain access to each blob.

In order to expose an additional interface for Blobseer service beside the current Blobseer's APIs and to see how to exploit Blobseer's features in the context of a DFS, the internship prepared by this bibliographical study will focus on Object-based file system thanked to Blobseer object management service. We have chosen integrating Blobseer and GFarm File system [1]. According to object-based file system architecture, a BlobSeer instance will be deployed in each storage node to accommodate objects. Delegated object management to Blobseer, Gfarm is free to expose fine-grain access, and versioning features which are crucial, and powerful in considering the requirement of data processing nowadays.

References

[1] *Gfarm v2: A grid file system that supports high-perfomance distributed and parallel data computing*. Proceedings of the 2004 Computing in High Energy and Nuclear Physics, 2004.

[2] D. Black A. Adamson B. Welch, B. Halevy and D. Noveck. pnfs operations summary. Technical report, IETF, 2004.

[3] Philip H. Carns, Walter, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[4] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[5] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 119–123, 2005.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[7] Apache Inc. http://hadoop.apache.org/core/docs/current/hdfs_design.html.

[8] Adrien Lebre, Renaud Lottiaux, Erich Focht, and Christine Morin. Reducing kernel development complexity in distributed environments. In *Euro-Par*, pages 576–586, 2008.

[9] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Comput. Surv.*, 22(4):321–374, December 1990.

[10] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *Communications Magazine, IEEE*, 41(8):84–90, 2003.

[11] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Distributed management of massive data. an efficient fine grain data access scheme. In *International Workshop on High-Performance Data Management in Grid Environment (HPDGrid 2008)*, Toulouse, 2008. Held in conjunction with VECPAR'08. Electronic proceedings.

[12] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling lock-free concurrent fine-grain access to massive distributed data: Application to supernovae detection. In *Poster Session - IEEE Cluster 2008*, Tsukuba, Japan, September 2008. Cluster/Grid.

[13] S. Carnevali A. Anagnostatos V. Hristidis S. Almukhaizim, K. Lakkas. Features and tradeoffs in distributed file systems, 2000.

[14] P. Schwan. Lustre: Building a file system for 1000-node clusters, 2003.

[15] M. Christie T. Fujita. Tgt: Framework for storage target drivers. In *Proceedings of the Ottawa Linux Symposium*, Ottwata, Canada, 2006.

[16] Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. The panasas activescale storage cluster - delivering scalable high bandwidth storage. *sc*, 00, 2004.

[17] R. O. Weber. Information technology—scsi object-based storage device commands -2 (osd-2). Technical report, INCITS Technical Committee T10/1729-D, 2007.

[18] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[19] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2004. IEEE Computer Society.