A Survey of Distributed Storage for Unstructured Data : Technical Report LUSY-2013/1

Andrej Tolič, Andrej Brodnik

University of Ljubljana, Faculty of Computer and Information Science
`{andrej.tolic,andrej.brodnik}@fri.uni-lj.si`

Abstract. Computer systems such as clusters, grids and clouds rely heavily on techniques for storing vast amounts of unstructured data. Notions like performance, availability, scalability and security are among the main requirements. In this survey we present such a storage – distributed storage – and describe problems related to the design and implementation. The emphasis is put on distributed file systems, but we also present some other forms, such as key-value stores.

1 Introduction

Computer systems that process or store large amounts of data (search engines, cloud computing applications, data mining applications, scientific computing etc.) require a high-performance reliable and scalable infrastructure for storing data. An obvious choice is to build such a system as a distributed system that will leverage high-speed interconnects, power and storage of single machines. Since this does not mean we can *just* distribute a local file system, there is a whole systems research area dealing with design and implementation of distributed storage. This paper tries to review the most important contributions of the last 25 years or so combined with some introduction to file systems.

As the title suggests, we are considering storage systems for unstructured data, that is, systems that don't use data models. Such systems are distributed file systems, although there are some variations that do not expose (only) a file system, and more importantly, are built on top of different architectures.

We first start with introduction to file systems and review some fundamental papers regarding local file systems. This is followed by an short introduction to distributed systems. We then move to distributed storage systems, starting with early DFSs, which is followed by a subsection on changing architecture. Here, we survey two fundamental systems, Petal and NASD. Further on we consider shared-disk file systems, mainly popular in high-performance computers and later move on to object-based systems, probably the most attractive of the time. Due to using variable-length objects as a lower layer, they can be used to provide multi-interface (API) storage to clients, so we consider such systems in the last subsection. We conclude with future work suggestions.

2 Introduction to File Systems

Since the emphasis in this survey will be put on distributed file systems, it is necessary to first define a notion of a file system (FS), before moving on to distributed file systems (DFS).

We follow the description of a file system according to [42] although, when dealing with distributed storage, many authors name key-value stores and other forms of storage as file systems. In order to differentiate this other types from the "classical" file system, we call the later a POSIX FS (see below).

A POSIX file system consists of two parts: a collection of *files* each storing related data, and a *directory structure*, which organizes the files and provides information on them. The later is usually called metadata and can be file size, date of creation, access control list, extended attributes (xattrs) etc.

The POSIX standard for operating systems also defines FS interface and behavior. Local file systems mostly follow this standard, whereas DFS often deviate from it as a result of specific design decisions. Despite this, when a storage system exposes interface similar to that of a classical FS (hierarchical directory structure, file names, open/close and read/write operations etc.), even if not fully compliant, we say that we are dealing with a POSIX FS, or that a storage provides POSIX interface. This is important since many authors describe key-value stores and other forms that don't expose the POSIX interface as file systems.

It is important to realize that the internal structure of a file is almost always defined by the user (an application) and not the file system[1]. To the file system, the contents of the file are a sequence of bytes, but the operating system can support different file types. This is sometimes done by using an extension to the file name, or by special information embedded into a file. However, the **absence of a data model** is the main distinction of unstructured storage from a structured storage (a relational database).

[24] describes an implementation and an upgrade of a classical (local) file system for Unix OS also known as Unix File System (UFS) or **Berkeley Fast File System**. Concepts that the paper describes (from older versions) or introduces are found in modern file system. Among them are a superblock, inodes, free blocks list etc. Inodes using direct, single-, double- and triple-indirect blocks for file content allocation are presented. The need for high throughput due to frequent paging of data in and out of the file system is recognized and addressed by increasing block size from 512 to 4096 bytes, but still preserving space efficiency for small files by using block fragments. Some novelties seem trivial, but are important, such as duplicate copies of a superblock in case of failures. The locality of accesses is also investigated, for example, a "list directory" command accesses all of the inodes of one directory. The paper also discusses use of block-level caches or buffers, the analogy to caches used in DFSs.

[1] There are file systems that make distinction between, for example, text and binary files, or data and program files etc.

[11] presents a file system that uses a write-ahead **journal (log)**. This means that data is first written to a special part of a file system – the journal. At a later time (see group commit below), the changes are commited to the file system itself on the disk. The use of a journal greatly simplifies crash recovery, since unsuccessful writes to the file system can be replied when recovering, while unsuccessful writes to the journal can simply be ignored. At recovery the system only replays the last portion of the journal, while in non-journaling file system it would have to check the entire contents of a file system for possibly inconsistent data. Another novelty the paper presents is the *group commit* which is a logical consequence of having a journal. The file system caches several operations together into a single write. This greatly increases write performance. Of the two presented concepts, both are derived from database systems.

[33] presents a **log-structured** file system. It is a step further from a journaling file system since the log itself is used as a file system. Authors claim that such a system exhibits a performance increase of an order of magnitude for small-file writes, while matching the performance for large files in comparison to non-log-structured file systems of the time. The motivation was similar to the one for journaling file systems but with the intent to avoid double-write penalty of journaling file systems. The most important issue is managing free space, since the problem is how to make large extents of free space available for writing after many deletions and overwrites of small files. The solution is to divide the log into segments and periodically use segment cleaner (garbage collection technique) to compress and relocate live data and avoid fragmentation. Even though the most widely used local file systems are journaling, log-structured file systems became important with the advent of flash memory storage devices. The JFFS2 and YAFFS are both log-structured.

A more recent paper is [30]. It analyzes several journaling file systems: ext3, IBM JFS, ReiserFS and Windows NTFS. The paper presents *semantic block-level analysis (SBA)*. Authors occupy the file system with carefully chosen workload patterns and observe not only the time needed but also the sequence of read and write operations. More important to this survey than analysis itself is the accompanying description of modern journaling file systems.

3 Distributed Systems

Distributed systems are an important part of modern IT infrastructure. A distributed system is a collection of *loosely coupled* processors (nodes, computers, hosts etc.) (see [42]). This means that processors do not share primary memory, as oposed to parallel systems. Reasons for building them are usually *resource sharing*, *computation speedup*, *reliability* and *communication*. An ideal distributed system should exhibit transparency, fault-tolerance and scalability. Storage systems which will be presented in this survey can be used by parallel systems as well, but are still generally implemented as distributed systems.

For example, a shared-memory supercomputer (parallel system) can use as a storage backend a distributed storage system built on a cluster (distributed system) of PCs. Or it may use a shared-disk file system that utilizes a distributed locking mechanism. In both these examples we are dealing with problems related to distributed computing. However, due to lack of space we will not cover the area, but just an exemplary issue that arises when designing a distributed storage. For a broader coverage, see [44] or [23].

An important problem with distributed systems is how can the participating machines **reach an agreement** on a value which was proposed by several participants. This is also called a **consensus** problem. Seminal work was done by Leslie Lamport with the introduction of **Paxos** protocol ([19]). Consensus is important in order to implement a *replicated state-machine*, a technique also proposed by Lamport on how to implement faul-tolerant distributed systems. State-machine replication was further reviewed in [38]. Paxos is used by a few systems we describe later, among them Google's Chubby locking service, Ceph, a fault-tolerant DFS. The main difference between Paxos and other consensus protocols, such as three-phase commit is it's resilience to failures. In ordinary commit protocols, if one node does not agree to the proposed value, the value is rejected, however, in paxos, only a majority is needed, thus paxos exhibits resilience to failure of almost half of the nodes.

An also important issue with distributes systems is that of byzantine fault-tolerance ([20]). Byzantine failures are ones where participants can fail without stopping. This means they continue working but exhibit erratic or possibly malicious behaviour. Therefore, designing byzantine fault-tolerant distributed systems is a complex task.

4 Distributed (Unstructured) Storage

In this section we survey several distributed storage systems. Some authors name all of these systems "file systems", even though some do not expose a POSIX interface. Furthermore, DFSs are sometimes called network, cluster, cloud, grid or parallel file systems among others. These expressions are sometimes used for a specific set of DFSs, but not necessarily.

4.1 Early Distributed File Systems

NFS ([35][43]) (Network File System) is probably the most widely used DFS. It was developed at Sun Microsystems in the mid-80s for use within groups of Sun workstations. It could be argued whether it is really a DFS, mainly due to lack of scalability. According to [42], a DFS is any file system where clients, servers and storage devices are dispersed among the machines of a distributed system. By this definition, NFS is a DFS, although it does not provide fault-tolerance, replication, parallel access and similar services provided by modern DFSs. Until version 4 (finalized in 2003) NFS was a stateless protocol. Client accessed remote files the same way as local files. NFSv4 ([40]) introduced states, client caches and mandatory locking (for non-UNIX systems). When client caches a file, the server *delegates* the responsibility for this particular file to the client, and all the other client, when accessing the particular file, are redirected by the server to the delegated client. NFS 4.1 ([41]), also known as pNFS (parallel NFS), introduces separation of meta-data and data. It defines file-, block- and object-level file layouts, the details of which are left to particular implementations.

File locking is in NFSv4 implemented with leases, where a client is granted a lease on a file for certain period of time and if it does not renew the lease, the lock is removed from a file. This is important in case of client failures, where file locks could otherwise pertain until the restart of the server. The technique of leases was introduced in [10].

AFS (Andrew File System), introduced in [26] and further discussed in [14], is one of the most studied DFSs. It was developed at Carnegie Mellon University, as part of the Andrew Project, another project to develop a distributed operating system (see Sprite). Authors emphasize the issue of scalability in comparison with NFS. If the later was designed to be used within smaller groups of workstations, authors of AFS targeted several thousand machines (and possibly more). One of the novelties is heavy use of caches and a callback mechanism. Callbacks provide cache consistency in a way later adopted by NFSv4, where callbacks are named delegations. There are two components in AFS, called Vice and Venus. Vice are the storage servers holding the actual data. These servers do not use local file systems for storage, but rather save data to raw partitions for performance reasons. They do, however, use local file system for storing metadata. Venus is a client-side component responsible for handling cache and contacting venus servers as needed. AFS caches entire files from servers. Only when opening or closing files is there any communication between clients and servers. Clients see a uniform name space. The distribution is achieved by partitioning name space into sub-trees and choosing a dedicated server to be responsible for a particular sub-tree. Sub-trees that contain frequently read, but rarely modified, files can have read-only replicas at other servers. These copies are created manually by system administrators. They are primarily used for system files in order to enhance availability and to distribute server load. AFS also provides security (ACLs, Kerberos client authentication) and a limited form of snapshots implemented as a copy-on-write backup mechanism.

In [28] authors describe The **Sprite** network operating system, developed at UC, Berkeley. An essential part of it is a network file system. The authors claim their system improves on transparency with regards to other systems, such as NFS. An important difference to NFS is that all of the machines should see the same directory hierarchy. Another important novelty is usage of *prefix tables* that are similar to IP routing tables. A client looks up a file using a full path name by finding the longest prefix match in the prefix table, and then contacting the server pointed to by the appropriate table entry. This server then strips away the prefix and continues to search directory hierarchy in order to locate the appropriate file. The third important contribution of this system is use of block-level in-memory **client and server caches**. AFS also uses caches, but they are stored on local disks.

[22], a survey paper, is one of the first to make the point, that a DFS should not be just a local file system extended to work over the network, but should be implemented as a distributed system with fault-tolerance and scalability in mind. Written in 1990, it discusses DFSs of the time – Unix United, Locus, Sun NFS, AFS and Sprite – but first presents DFS design issues. Authors name a local file system as *conventional*. Authors point out that any centralized solution has a single point of failure (SPOF) and a performance bottleneck. The solution is for a DFS to have *multiple and independent servers controlling multiple and independent storage devices*. The authors discuss several aspects of a DFS: location transparency, naming scheme and implementation details. The later include pathname translations, mount mechanisms and hints. Authors also discuss *sharing semantics*, a topic of interest for the clients, especially in large installations where simultaneous accesses to files are common. Caching issues (granularity, location, consistency etc.) are investigated along with fault-tolerance. The later is described as a functionality allowing client access to files in the presence of link, server or storage device failures. Authors also mention usage of file replicas for better performance, since DFS can service a client's request from the nearest replica.

[13] presents **WAFL** (Write Anywhere File Layout) protocol, developed by Netapp company. WAFL is not a DFS, but a file system layer used by Netapp in a NFS appliance. A distinguishing feature are snapshots, that provide read-only copies of files. WAFL holds up to 20 snapshots at a time, and periodically deletes them and creates new ones. Snapshotting is based on a **copy-on-write** technique, where, when creating a snapshot, the root inode is copied, and later when a block is changed, WAFL copies it and updates the pointers. All of the meta-data is contained in 3 special files, holding inodes, free blocks map and free inodes map. This is where WAFL gets its name - write anywhere, since because metadata is written in files, blocks of this files can be written anywhere on a disk, specifically in the same locations as the files contents, thus exploiting temporal locality and scheduling writes in a way to decrease RAID penalty of multiple writes due to parity.

[12] introduces **Zebra** Striped Network File System. It was designed as a replacement to be used in Stripe distributed OS. The main idea is to stripe data of one file among several servers while using parity (similar to RAID) to defend against node failures. It is not the first DFS to employ striping to increase large-file performance necessary for supercomputing applications, but it also employs techniques borrowed from log-structured file systems ([33]) in order to increase small-file performance by batching several small writes to a log, and then striping this log over several machines. Zebra also decouples metadata paths from data paths with the introduction of *file managers*. They behave like Stripe file servers, only that the contents of the file are pointers to file blocks stored on storage servers. Therefore, there are four components in a Zebra file system: clients, storage servers, file managers (metadata) and cleaners. The later are used to recover free space as a result of a log-based approach. The RAID-like technique is important since it enables installation of such a system using

commodity hardware, and presents a second alternative to faul-tolerance, the first being file replication. RAID-like techniques are increasingly important in modern DFSs, since plain file replication is inadequate due to high storage costs.

[2] introduces a paradigm shift from centralized servers (dedicated to a particular role) to a design, where all the server machines (cooperating as peers) can service all file system requests. This is very important for scalability, high-performance and fault-tolerance. Authors introduce a prototype DFS, called **xFS**, which borrows log-structured techniques from Zebra. xFS has the same components as Zebra, namely clients, storage servers, file managers and cleaners, but provides more flexibility as to which components does a particular machine run. There are some restrictions, for example, managers and cleaners must also be clients. xFS also introduces cooperative caching where portions of client memory are used as a global cache. The paper claims that a 32-node installation with 32 clients exhibits almost the same performance as it would if there were only one client. An important notion is that of stripe groups. They bound the striping to a particular group of servers which is important for large-scale installations and in contrast to Zebra. Since stripe groups seldom change, they are replicated globally (to all the machines). Stripe groups a similar to a concept of placement groups which we see later in Ceph ([34]).

[36] is an article on the evolution of the **Coda** DFS. Coda has its origins in the AFS file system and has been in development since 1987. Some of the most important additions are a better fault-tolerance and a support for *disconnected operation* (interesting for mobile computing). Since Coda is a descendant of AFS it already caches entire files. But when server link failures occur, clients keep writing changes to local disk, and when the link is up again, they try to replay all the changes to the appropriate servers. Of course, conflicts can arise, sometimes they can be resolved automatically, others, administrator intervention is needed. This is where *application specific resolution* comes into play. Authors suggest giving application developers the possibility to write app-specific software which the Coda can invoke to resolve conflicts. As far as replications is concerned, Coda uses so called volume groups to replicated the data. Versions are used in order to identify servers with out-of-date data. In case of failures, servers can easily be reintroduced with a few administrator commands. The systems takes care that the server gets the appropriate data copied to it upon rejoining the cluster. Coda holds volume and directory information, access control lists and file attribute information in raw partitions. These are accessed through a log based recoverable virtual memory package (RVM) for speed and consistency. Only file data resides in the files in server partitions. RVM has built in support for transactions - this means that in case of a server crash the system can be restored to a consistent state without much effort.

4.2 Changing architecture

[8] discusses differences between network-attached storage (NAS) and storage-area networks (SAN). NAS is a storage system that provides user access to files (think of `file open`, `file close`, `file read` operatons) where users and

NAS systems are separated by a network. Similarly, SAN provides users the access to block device (think of `get(block number)`, `put(block number,data)` operations) where users and SAN systems are separated by a network. To put it differently, NAS places a network between user and file system, while SAN places a network between file system and block device. According to [8], NAS systems are becoming more specialized (NAS clusters), while SAN systems are becoming more generalized (iSCSI), thus, these two approaches are converging.

Petal is introduced in [21]. It provides clients a block-oriented interface ie., virtual disks. Petal scales by splitting the controller function over a cluster of controllers, any one of which had access to consistent global state. According to [8], Petal could be viewed as a RAID system implemented on a symmetric multiprocessor, though it uses Lamport's Paxos algorithm for distributed consensus (see [19]) instead of shared memory for global state. Even though Petal was designed in 1996, it is interesting for use in virtual environments where we would like to provide block devices to particular virtual machines, so they can create local file systems on top of them. The idea of distributed virtual disks is later seen with Ceph (Rados Block Device), a system that provide distributed virtual disks as a layer on top of a different storage.

4.3 Shared-disk file systems

Shared-disk file systems (also known as cluster file systems) are a type of DFSs where clients share access to the same block-level device (physical or virtual), usually exposed by a SAN device. Clients all have shared-disk FS software with which they access shared storage. An important part of such software is a **distributed lock manager** (DLM) in order to control concurrent access to shared storage.

Most distributed file systems implement a **distributed lock manager** (DLM) similar to the one used in DEC's **VAX/VMS** system and introduced in [17] in 1986. The VMS lock manager allows cooperating processes to define shared resources and synchronize access to those resources. A resource can be any object an application cares to define. Each resource has a user-defined name by which it is referenced. The lock manager provides basic synchronization services to request a lock and release a lock. Each lock request specifies a locking mode, such as exclusive access, protected read, concurrent read, concurrent write, null, etc. If a process requests a lock that is incompatible with existing locks, the The data is divided into two parts: the resource lock descriptions and the resource lock directory system. Both are distributed. Each resource has a master node responsible for granting locks on the resource. The master maintains a list of granted locks and a queue of waiting requests for that resource. The second part of the database, the resource directory system, maps a resource name into the name of the master node for that resource. The directory database is distributed among nodes willing to share this overhead. Given a resource name, a node can trivially compute the responsible directory as a function of the name string and the number of directory nodes.

Another, but slightly different example of a DLM, is **Chubby** ([5]), a distributed lock service used by Google's distributed systems such as MapReduce, GFS (see below) and BigTable. The design emphasis is on scalability and reliability, and not high performance. Chubby implements locks, a reliable small-file storage system, and a session/lease mechanism in a single service.

Frangipani [45] is a shared-disk file system built on top of Petal. The disk being shared is not physical, but a virtual one provided by Petal. It consists of the following main components: Petal Server, Distributed Locking Service and the Frangipani File Server Module. The Petal Server is responsible for providing a common virtual disk interface to storage that is distributed in nature. The Petal device driver mimics the behaviour of a local disk, hiding its distributed nature. The Distributed Locking Service is responsible for enforcing consistency, thus changes made to the same block of data by multiple Frangipani servers are serialised ensuring data is always kept in a consistent state. The are two main types of locks, a read lock and a write lock. A read lock allows a server to read the data associated with the lock and cache it locally. If it is asked to release its lock, it must invalidate its cache. A write lock permits the server to read and write to the associated data. The third component is the Frangipani File Server Module, which interfaces with the kernel and the Petal device driver to provide a file system-like interface. Frangipani File Server communicates with the Distributed Locking Service to acquire locks and ensure consistency, and with Petal Servers for block-level storage capability.

Another well-know and commercially successful shared-disk file system is **GPFS** ([37]), General Parallel File System, from IBM. It is based on an earlier file system, called Vesta. One of distinct features of GPFS is *byte-range locking* which enables multiple clients to write to the same file in parallel and without compromising consistency. As the name suggests, GPFS supports parallel access which is achieved by striping data at a block level. Similar to VMS DLM, node wishing to write to a file contacts a designated node in order to obtain lock. There is also a designated *allocation manager*, a node charged with managing and allocating free space. GPFS also tracks failed disks or failed nodes. In case of communication failures, GPFS fences failed nodes so they cannot access shared disks. Since GPFS pays special attention to parallelism and scalability, it is used on some of the fastest supercomputers.

Other important shared-disk file systems include Oracle's **OCFS**/OCFS2 (Oracle Cluster File System), Redhat's **GFS**/GFS2 (Global File System) and Vmware's **VMFS** (Virtual Machine File System).

4.4 Object-based file systems

One of the main ideas of object-based file systems (but not specific to them, since it is also present in some shared-disk FSs) is the separation of metadata and data. By using **variable-length objects** (with attributes) instead of blocks, we construct an object storage system that takes care of parallel access, replication (or erasure coding) etc. at the object level. These objects are stored locally by the object servers (often **called object-based storage devices** - OSD) usually as files on a local FS such as ext4, xfs, zfs etc. We then implement POSIX semantics as a layer on top of object storage. Or we could implement some other API, such as a key-value store. Typical operation is the following. The client contacts a metadata server when it wants to perform a file operation. The metadata server sends a layout (map) of where the data is located and the client communicates with the object storage nodes directly and possibly in parallel.

The seminal work on object-based storage occurred at Carnegie Mellon University's Parallel Data Lab (PDL) with the Network-Attached Secure Disks (**NASD**) project ([9]), where they emphasize variable-length data objects. NASD systems use central policy server (possibly a cluster of servers), while most commands and data transfers are directly between device and client. Upon approval by the central server, clients can access (directly and in parallel) all devices containing data of interest. There are many ways to ensure that the server in such asymmetric systems controls client access to storage. The most common trusts client operating systems to request and cache metadata as is done in Zebra, where each client temporarily functions as a server for the files whose metadata it has in cache. By storing file metadata in the device, NASD offers a file interface abstraction. However, it does not offer directory operations, because it assumes data is spread over multiple NASDs and that the file system's directories are globally defined. Nevertheless, a NASD policy server stores directory information in NASD files that clients can parse without the device recognizing the difference between a regular file and a directory.

Lustre ([39]) is an object-based DFS which has been designed for use in high-performance computing. Lustre is sometimes called a parallel file system since it supports striping data across several storage nodes and thus providing parallel access. There are two types of components in a Lustre cluster. Metadata servers contain the file system's directory layout, permissions and extended attributes for each object. First versions employed only one (and a backup) metadata server, later versions are designed to use many in order to distribute the load. Object Storage Targets are responsible for the storage and transfer of actual file data. Both types of service node can operate in pairs which automatically take over for each other in the event of failure. Each also runs an instance of the Lustre distributed lock manager which is based on the VAX/VMS design we described earlier. Lustre stores data as objects called containers that are very similar to files, but are not part of a directory tree.

PVFS (Parallel Virtual File System) is introduced in [6]. Similar to other object-based DFSs, PVFS has metadata servers and data servers. Since version 2, PVFS supports multiple metadata servers. The word virtual in the name means that it doesn't write directly to the object storage devices, but instead the data resides in another file systems that does the actual I/O operations to the storage devices. When data is written to PVFS, it is sent to the underlying file system in objects. The underlying file system then writes the data to the storage. PVFS allocates objects in a round-robin fashion (compare to CRUSH algorithm in Ceph, which we describe later). Another similar and modern file system is PanFS ([47]), developed by Panasas.

Possibly the most discussed modern DFS is the **Google file system** ([7]). Despite this fact, it was designed for a set of specific usage patterns, particularly writing large files which are seldom modified. It is a backbone for the MapReduce and BigTable platforms. Open-source equivalents are Apache Hadoop and its filesystem, HDFS. The interface it provides is not strict POSIX compliant but looks similar. There is also no caching involved at either side, client or server. GFS is designed around two types of components. The metadata server is called a *master*. According to the original paper, there is a single master, although it is quite possible that the GFS in use at Google today, supports multiple masters. Master's state is replicated on backup machines, in order to provide fast recovery after a crash. Object data servers are called *chunkservers*. GFS uses very large (64MB) objects called *chunks*. Chunkservers store chunks and provide additional integrity by calculating and storing checksums for 64 kb large blocks of chunks. Parallel access to multiple chunks which are replicated on multiple chunkservers is possible. GFS supports snapshots. Like AFS, it uses copy-on-write mechanism. When the master receives request for a snapshot it first revokes any outstanding leases on the chunks in the files it is about to snapshot. Then it logs the operation to disk, and applies this log to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snap- shot files point to the same chunks as the source files. The first time a client wants to write to a chunk after the snapshot operation, a copy is made on the chunkserver holding the original chunk, thus avoiding network transfer.

4.5 A Multi-interface Example

Before presenting the last DFS, Ceph, let us first look into APIs (interfaces) for storing unstructured data. Until now, we mainly dealt with file systems that provided POSIX or near-POSIX API to the user. However, we did encounter two examples where there was a different bottom layer on top of which was (or could be) a POSIX FS. One such example was Petal, where the bottom layer provided virtual block devices on top of witch we could design a shared-disk file system. A second example were object-based file systems, where the bottom layer was a distributed object store on top of which there was a POSIX FS layer. The question presents – is this the optimal API? How good is the hierarchical namespace? Why not just have **key-value** mappings? Can we implement key-value stores easily on top of object stores? How much of "on-top-of" layering makes sense and doesn't hinder performance? Or could we maybe extend POSIX API for better indexing and querying. The choice of interface influences the system design as well. If we do not require POSIX FS interface, a lot of metadata handling issues are gone. One such example is Facebook's photo storage system called Haystack, presented in [3], which implements an object-based storage with a very specific application in mind.

A distributed file system that offers multiple APIs is **Ceph** ([34]). It is actually a distributed object storage, on top of which are block-, object- and file-level access APIs. A main component of Ceph is RADOS ([46]), a reliable autonomic distributed object store that provides object-based storage. In Ceph, responsibility for data migration, replication, failure detection and failure recovery is with RADOS. This enables Ceph to expose the storage through three different interfaces – rados block devices (similar to Petal virtual disks), POSIX FS and key-value store, similar to Amazon S3 ([1]) and Openstack Swift ([27]), which can be exposed through HTTP-based protocols (radosgw) or accessed by a librados library. Architecture is shown in figure 1 while cluster layout, which is an example of typical object-based file system with separation of metadata and data paths, is shown in figure 2. In the paper, RADOS uses a special local file

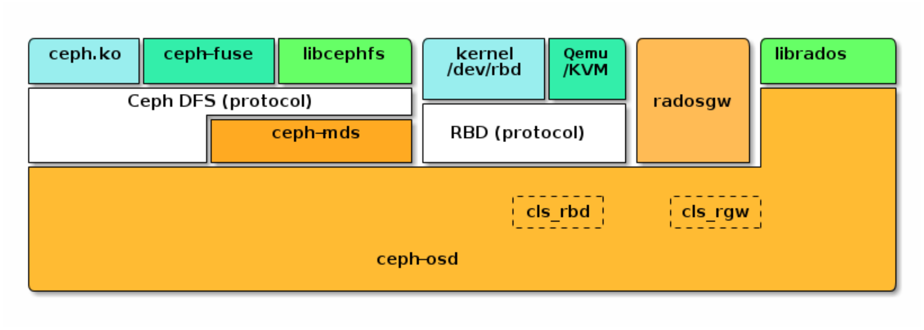

Fig. 1. Architecture of Ceph.

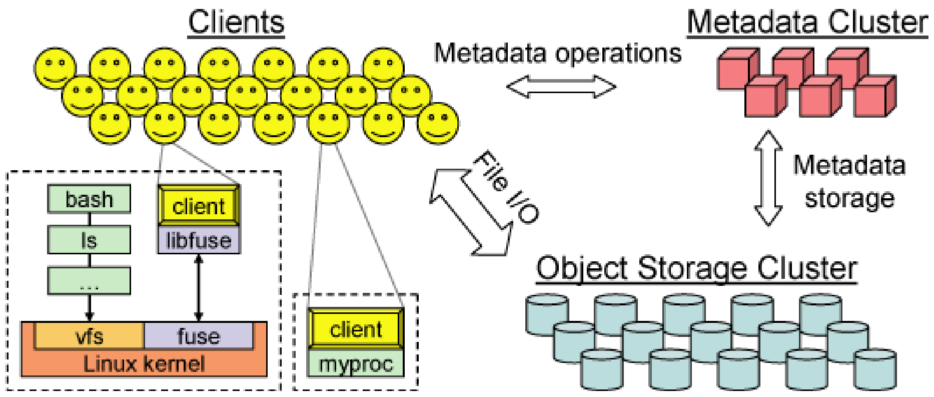

Fig. 2. Layout of a Ceph system.

system to store objects on OSDs, called EBOFS. However, currently available implementations use already existing local file systems. The authors discovered that, btrfs, a local file system currently still in development, offers all the properties needed. At the moment they advise the use of XFS, although any other local FS that supports extended attributes could be used. Ceph cluster consists of metadata servers (MDSs), object-storage devices (OSDs) and cluster monitors (MONs). The later run a variant of Paxos ([19]) protocol in order to achieve consensus on the state of the cluster. State in this case is a *cluster map*, a sort of cluster description. Due to Paxos' need for majority of votes, there always has to be an odd number of MONs. In contrast to most other object-based file systems which replace long per-file block lists with shorter object lists, Ceph eliminates allocation lists entirely. A special-purpose data distribution function CRUSH maps objects to storage devices. This allows any node to calculate (rather than look up) the name and location of objects, thus reducing the metadata cluster workload. CRUSH uses cluster map, but this map is a fairly static structure and changes only when nodes join or leave the cluster. Since metadata operations make up as much as half of typical POSIX file system workloads ([32]) metadata management is critical to overall system performance. For the purpose of providing POSIX interface, Ceph uses dynamic subtree partitioning which distributes responsibility for managing the file system namespace hierarchy among tens or even hundreds of MDSs.

A similar architecture (shown on figure 3) to that of Ceph is the one used by parallel NFS, also called **pNFS** ([41]) which we briefly mentioned earlier in the paper when describing classical NFS. However, here, the pNFS takes care for the metadata and appropriate communication with storage devices regarding this metadata. Data paths have to be taken care of separately, since with pNFS, the data does not pass through the NFS server. Clients and the data storage system communicate directly through block-, file- or object-based protocols.

Let us briefly mention two other examples. A similar approach to placement calculation (instead of look up) is taken by **GlusterFS** ([31]), a modern DFS developed with support from Redhat.

Another active object-based DFS is **XtreemFS** ([15]), developed with funding from European Union as a part of the XtreemOS distributed operating system. It uses a lease protocol ([16]) based on Paxos in order to coordinate leases in a fault-tolerant way.

5 Conclusion and Future work

As promised we surveyed some of the most important work needed to understand modern distributed file systems. Since there a plenty such systems, we tried to mention only some of the more important ones, but some also important were probably left out.

Also, since we were dealing with unstructured storage, we did not look into semi-structured storage systems, mainly NoSQL databases although future research might lead us into that direction. Key-value stores – not in a NoSQL sense, but an Amazon S3 sense – were mentioned where they were provided as

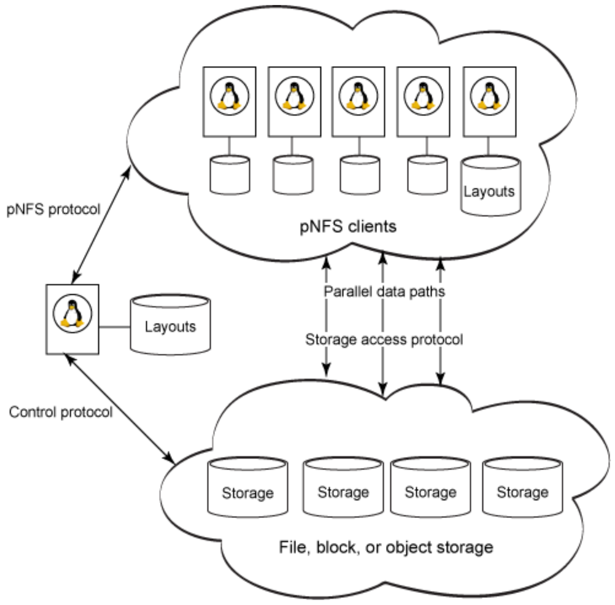

Fig. 3. Architecture of pNFS.

an additional interface or access method. Since there aren't many non-NoSQL key-value systems around and not much is known about ones that are (Amazon S3), we did not address them separately.

Here we list some of the future work. Mainly it is about going deeper into existing systems and dealing with specific issues. Some of them are the following:

- A better review of security mechanisms. A survey article [25] deals with decentralized access control in DFSs .
- A review of wide-area distributed file systems, such as OceanStore ([18]) and Farsite ([4]).
- Problem of deduplication. At what level (block, object, file or some other) do we do it an how.
- Adequacy of TCP protocol for transfers.
- A completely different API, specifically for indexing and searching. How does it change the architecture and does it reach into structured world.
- How does virtualization fit into all this? What kinds of file systems are the most appropriate, should VMs be aware of using DFSs and "act accordingly".
- Erasure codes as an alternative to n-way replication.
- Replica placement strategies.
- Replacing block-devices (HDDs) on storage servers with RAM – RAMcloud ([29]).

References

1. Amazon: Amazon S3 (2012), http://aws.amazon.com/s3/
2. Anderson, T.E., Dahlin, M.D., Neefe, J.M., Patterson, D.A., Roselli, D.S., Wang, R.Y.: Serverless Network File Systems. ACM Transactions on Computer Systems 14(1), 41–79 (Feb 1996)
3. Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P.: Finding a needle in Haystack: facebook's photo storage. In: OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation. pp. 1–8 (Oct 2010)
4. Bolosky, W.J., Douceur, J.R., Howell, J.: The Farsite project. ACM SIGOPS Operating Systems Review 41(2), 17–26 (Apr 2007)
5. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th symposium on Operating systems design and implementation - OSDI '06. pp. 335–350 (Nov 2006)
6. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: a parallel file system for linux clusters. In: ALS'00 Proceedings of the 4th annual Linux Showcase & Conference - Volume 4. p. 28 (Oct 2000)
7. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google File System. ACM SIGOPS Operating Systems Review 37(5), 29 (Dec 2003)
8. Gibson, G.A., Van Meter, R.: Network attached storage architecture. Communications of the ACM 43(11), 37–45 (Nov 2000)
9. Gibson, G.A., Zelenka, J., Nagle, D.F., Amiri, K., Butler, J., Chang, F.W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D.: A cost-effective, high-bandwidth storage architecture. ACM SIGPLAN Notices 33(11), 92–103 (Nov 1998)
10. Gray, C., Cheriton, D.: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In: Proceedings of the twelfth ACM symposium on Operating systems principles - SOSP '89. pp. 202–210. ACM Press, New York, New York, USA (1989)
11. Hagmann, R.: Reimplementing the Cedar file system using logging and group commit. In: Proceedings of the eleventh ACM Symposium on Operating systems principles - SOSP '87. vol. 21, pp. 155–162. ACM Press, New York, New York, USA (Nov 1987)
12. Hartman, J.H., Ousterhout, J.K.: The Zebra striped network file system. ACM Transactions on Computer Systems 13(3), 274–310 (Aug 1995)
13. Hitz, D., Lau, J., Malcolm, M.: File system design for an NFS file server appliance. In: WTEC'94 Proceedings of the USENIX Winter 1994 Technical Conference on USENIX. p. 19 (Jan 1994)
14. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.a., Satyanarayanan, M., Sidebotham, R.N., West, M.J.: Scale and performance in a distributed file system. ACM Transactions on Computer Systems 6(1), 51–81 (Feb 1988)
15. Hupfeld, F., Cortes, T., Kolbeck, B., Stender, J., Focht, E., Hess, M., Malo, J., Marti, J., Cesario, E.: The XtreemFS architecture—a case for object-based file systems in Grids. Concurrency and Computation: Practice & Experience 20(17), 2049–2060 (Dec 2008)
16. Hupfeld, F., Kolbeck, B., Stender, J., Högqvist, M., Cortes, T., Marti, J., Malo, J.: FaTLease. In: Proceedings of the 17th international symposium on High performance distributed computing - HPDC '08. p. 1. ACM Press, New York, New York, USA (Jun 2008)
17. Kronenberg, N.P., Levy, H.M., Strecker, W.D.: VAXcluster: a closely-coupled distributed system. ACM Transactions on Computer Systems 4(2), 130–146 (May 1986)

18. Kubiatowicz, J., Wells, C., Zhao, B., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H.: OceanStore. ACM SIGOPS Operating Systems Review 34(5), 190–201 (Dec 2000)
19. Lamport, L.: The Part-Time Parliament. ACM Transactions on Computer Systems 16(2), 133–169 (May 1998)
20. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (Jul 1982), http://dl.acm.org/citation.cfm?id=357172.357176
21. Lee, E.K., Thekkath, C.A.: Petal. ACM SIGPLAN Notices 31(9), 84–92 (Sep 1996)
22. Levy, E., Silberschatz, A.: Distributed file systems: concepts and examples. ACM Computing Surveys 22(4), 321–374 (Dec 1990)
23. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (Jan 1996)
24. McKusick, M.K., Joy, W.N., Leffler, S.J., Fabry, R.S.: A fast file system for UNIX. ACM Transactions on Computer Systems 2(3), 181–197 (Aug 1984)
25. Miltchev, S., Smith, J.M., Prevelakis, V., Keromytis, A., Ioannidis, S.: Decentralized access control in distributed file systems. ACM Computing Surveys 40(3), 1–30 (Aug 2008)
26. Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S., Smith, F.D.: Andrew: a distributed personal computing environment. Communications of the ACM 29(3), 184–201 (Mar 1986)
27. Openstack: Openstack Swift Storage (2012), http://www.openstack.org/software/openstack-storage/
28. Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., Welch, B.: The Sprite network operating system. Computer 21 (1988)
29. Ousterhout, J., Parulkar, G., Rosenblum, M., Rumble, S.M., Stratmann, E., Stutsman, R., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D.: The case for RAMCloud. Communications of the ACM 54(7), 121 (Jul 2011)
30. Prabhakaran, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Analysis and evolution of journaling file systems. In: ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference. p. 8 (Apr 2005)
31. Redhat: GlusterFS (2012), http://www.gluster.org/
32. Roselli, D., Lorch, J.R., Anderson, T.E.: A comparison of file system workloads. ATEC '00 Proceedings of the annual conference on USENIX Annual Technical Conference p. 4 (Jun 2000)
33. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Transactions on Computer Systems 10(1), 26–52 (Feb 1992)
34. Sage A. Weil, S.A.B.: Ceph: A scalable, high-performance distributed file system. Proceedings of the 7th symposium on Operating systems design and implementation - OSDI '06 p. 14 (2006)
35. Sandberg, R., Golgberg, D., Kleiman, S., Walsh, D., Lyon, B.: Design and implementation of the Sun network filesystem. In: Proceedings of the Summer 1985 USENIX Conference. pp. 379–390 (Dec 1988)
36. Satyanarayanan, M.: The Evolution of Coda. ACM Transactions on Computer Systems 20(2), 85–124 (May 2002)
37. Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: FAST '02 Proceedings of the 1st USENIX Conference on File and Storage Technologies. p. 19 (Jan 2002)

38. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys 22(4), 299–319 (Dec 1990)
39. Schwan, P.: Lustre: Building a File System for 1,000-node Clusters. In: Proceedings of the 2003 Linux Symposium (2003)
40. Shepler, S., Eisler, M., Robinson, D., Callaghan, B., Thurlow, R., Noveck, D., Beame, C.: Network File System (NFS) Version 4, http://tools.ietf.org/html/rfc3530
41. Shepler, S., Noveck, D., Eisler, M.: Network File System (NFS) Version 4.1, http://tools.ietf.org/html/rfc5661
42. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, Seventh Edition. John Wiley & Sons (2007)
43. Staubach, P., Pawlowski, B., Callaghan, B.: NFS Version 3 Protocol Specification, http://tools.ietf.org/html/rfc1813
44. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms (2nd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (Oct 2006)
45. Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani. ACM SIGOPS Operating Systems Review 31(5), 224–237 (Dec 1997)
46. Weil, S.A., Leung, A.W., Brandt, S.A., Maltzahn, C.: RADOS. In: Proceedings of the 2nd international workshop on Petascale data storage held in conjunction with Supercomputing '07 - PDSW '07. p. 35. ACM Press, New York, New York, USA (Nov 2007)
47. Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., Zhou, B.: Scalable performance of the Panasas parallel file system. In: FAST'08 Proceedings of the 6th USENIX Conference on File and Storage Technologies. p. 2 (Feb 2008)