A Survey on Distributed File System Technology

J Blomer

CERN, CH-1211 Genève 23, Switzerland

E-mail: jblomer@cern.ch

Abstract. Distributed file systems provide a fundamental abstraction to location-transparent, permanent storage. They allow distributed processes to co-operate on hierarchically organized data beyond the life-time of each individual process. The great power of the file system interface lies in the fact that applications do not need to be modified in order to use distributed storage. On the other hand, the general and simple file system interface makes it notoriously difficult for a distributed file system to perform well under a variety of different workloads. This has lead to today's landscape with a number of popular distributed file systems, each tailored to a specific use case. Early distributed file systems merely execute file system calls on a remote server, which limits scalability and resilience to failures. Such limitations have been greatly reduced by modern techniques such as distributed hash tables, content-addressable storage, distributed consensus algorithms, or erasure codes. In the light of upcoming scientific data volumes at the exabyte scale, two trends are emerging. First, the previously monolithic design of distributed file systems is decomposed into services that independently provide a hierarchical namespace, data access, and distributed coordination. Secondly, the segregation of storage and computing resources yields to a storage architecture in which every compute node also participates in providing persistent storage.

1. Introduction

Distributed file systems provide persistent storage of unstructured data, which are organized in a hierarchical namespace of files that is shared among networked nodes. Files are explicitly created and they can survive the lifetime of processes and nodes until explicit deletion. As such they can be seen as the glue of a distributed computing infrastructure. Distributed file systems resemble the API of local file systems. To applications, it should be transparent whether data is stored on a local file system or on a distributed file system. This data model and the interface to applications distinguishes distributed file systems from other types of distributed storage such as databases.

Virtually all physics experiments store their data in distributed file systems. Large experiment collaborations, such as the experiment collaborations at the Large Hadron Collider (LHC), store data in a global federation of various cluster file systems rather than in a single, globally distributed file system. For LHC experiments, such globally federated and accessible storage sums up to more than 1 billion files and several hundred petabytes.

There is a variety of file systems available to choose from [1–14] and often it is not clear what are the particular strengths, weaknesses, and implications of using one distributed file system over the other. Several previous studies presented taxonomies, case studies, and performance comparisons on distributed file systems [15–20]. This survey is focused on the underlying building blocks of distributed file systems and what to expect from them with respect to physics applications.

The survey is mainly driven by the high-energy physics needs, which however cover many of the aspects relevant to distributed file systems in general.

2. How are Distributed File Systems Used?

Even though the file system interface is general and fits a broad spectrum of applications, most distributed file system implementations are optimized for a particular class of applications. For instance, the Andrew File System (AFS) is optimized for users' home directories [2], XrootD is optimized for high-throughput access to high-energy physics data sets [7], the Hadoop File System (HDFS) is designed as a storage layer for the MapReduce framework [10, 21], the CernVM File System (CVMFS) is optimized to distribute software binaries [12], and Lustre is optimized as a scratch space for cooperating applications on supercomputers [5]. These use cases differ both quantitatively and qualitatively. Consider a multi-dimensional vector describing different levels of properties or requirements for a particular class of data that consists of *data value*, *data confidentiality*, *redundancy*, *volume*, *median file size*, *change frequency*, and *request rate*. Every single use case above poses high requirements in only some of the dimensions. All of the use cases combined, however, would require a distributed file system with outstanding performance in every dimension. Moreover, some requirements contradict each other: a high level of redundancy (e. g. for recorded experiment data) inevitably reduces the write throughput in cases where redundancy is not needed (e. g. for a scratch area). The file system interface provides no standard way to specify quality of service properties for particular files or directories. Instead, we have to resort to using a number of distributed file systems, each with implicit quality of service guarantees and *mounted* at a well-known location (/afs, /eos, /cvmfs, /data, /scratch, ...). Quantitative file system studies, which are unfortunately rare, provide precise workload characterizations to guide file system implementers [22–24].

2.1. Application Integration

From the point of view of applications there are different levels of *integration* a distributed file system can provide. Most file systems provide a library with an interface that closely resembles POSIX file systems. The advantage is that the interface can be adapted to the use case at hand. For instance, the Google File System (GFS) extends POSIX semantics by an atomic append [25], a feature particularly useful for the merging phase of MapReduce jobs. A library interface comes at the cost of transparency; applications need to be developed and compiled for a particular distributed file system.

So called *interposition* systems introduce a layer of indirection that transparently redirects file system calls of an application to a library or an external process. The Parrot system creates a sandbox around a user-space process and intercepts its system calls [26]. The Fuse kernel-level file system redirects file system calls to a special user-land process ("upcall") [27]. Interposition systems come with a performance penalty for the indirection layer. This penalty is smaller for Fuse than for a pure user-level interposition system, but Fuse requires co-operation from the kernel.

Some distributed file systems are implemented as an extension of the operating system kernel (e. g. NFS [1], AFS, Lustre). That can provide better performance compared to interposition systems but the deployment is difficult and implementation errors typically crash the operating system kernel.

Distributed file systems do not fully comply with the POSIX file system standard. Each distributed file system needs to be tested with real applications. Functionality that is often poorly supported in distributed file systems is file locking, atomic renaming of files and directories, multiple hardlinks, and deletion of open files. Sometimes, deviations from the POSIX standard are subtle. HDFS, for instance, writes file sizes asynchronously and thus it returns the real size of a file only some time after the file has been written.

3. Architecture Evolution

The simplest architecture for a distributed file system is a single server that exports a local directory tree to a number of clients (e. g. NFSv3). This architecture is obviously limited by the capabilities of the exporting server.

An approach to overcome some of these limitations is to delegate ownership and responsibility of certain file system subtrees to different servers, as done by AFS. In order to provide access to remote servers, AFS allows for lose coupling of multiple file system trees ("cells"). Across cells, this architecture is not network-transparent: moving a file from one cell to another requires a change of path. It also involves a copy through the node which triggers the move, e. g. move is not a namespace-only operation. Furthermore, the partitioning of a file system tree is static and changing it requires administrative intervention.

In *object-based* file systems, data management and meta-data management is separated (e. g. GFS). Files are spread over a number of servers that handle read and write operations. A meta-data server maintains the directory tree and takes care of data placement. As long as meta-data load is much smaller than data operations (i. e. files are large), this architecture allows for *incremental scaling*. As the load increases, data servers can be added one by one with minimal administrative overhead.

The architecture is refined by *parallel file systems* (e. g. Lustre) that cut every file in small blocks and distribute the blocks over many nodes. Thus read and write operations are executed in parallel on multiple servers for better maximum throughput.

A *distributed meta-data* architecture (as for instance in Ceph [9]) overcomes the bottleneck of the central meta-data server. Distributed meta-data handling is more complex than distributed data handling because all meta-data servers are closely coupled and need to agree on a single state of the file system tree. That involves either distributed consensus algorithms [28, 29] or distributed transaction protocols such as two-phase commit.

Object-based architectures involve two servers (meta-data server, data server) for read and write operations. Instead of asking a meta-data server, *decentralized* or *peer-to-peer* file systems (e. g. GlusterFS [13]) let clients *compute* the location of data and meta-data by means of a distributed hash table. Zero-hop routing in a distributed hash table is restricted to a local network, however, in which every node is aware of every other node. On a global scale, tree based routing as being done by XrootD is simpler to implement and it shows better lookup performance than globally distributed hash tables. Furthermore, high peer churn (servers that frequently join and leave the network) pose a hard challenge on distributed hash tables.

3.1. Decomposition

There is a tendency of decomposition and modularization in distributed file systems. Examples are the offloading of authorization to Kerberos in AFS, the offloading of distributed consensus to Chubby [30] in GFS (resp. ZooKeeper [31] in HDFS), or the layered implementation of Ceph with the independent RADOS key-value store as building block beneath the file system. Another example is the separation of a distributed file system namespace and the data access. In the grid, for instance, the namespace is controlled by experiments' *file catalogs*, which, in combination with grid middleware, federates globally distributed cluster file systems.

For a globally distributed and administratively independent computing infrastructure used in high-energy physics, modularization is important because it allows for deployment of incremental improvements. A completely new file system takes many years to stabilize and roll-out throughout the grid.

4. Mechanisms and Techniques

A distributed file system should be fast and it should scale to many files, users, and nodes. At the same time, it should sustain hardware faults and recover gracefully from them and ensure the

integrity of the file system over long storage periods and long-distance network links. This section highlights techniques that are used to achieve these goals and that are particularly relevant for distributed file systems in high-energy physics.

4.1. File System Integrity

Global file systems often need to transfer data via untrusted connections and still ensure integrity and authenticity of the data. Cryptographic hashes of the content of files are often used to ensure data integrity. Cryptographic hashes provide a short, constant length, unique identifier for data of any size. Collisions are virtually impossible to occur neither by chance nor by clever crafting, which makes cryptographic hashes a means to protect against data tampering. Many globally distributed file systems use cryptographic hashes in the form of content-addressable storage [12, 32–34], where the name of a file is derived from its cryptographic content hash. This allows for verification of the data independently of the meta-data. It also results in immutable data, which eliminates the problem of detecting stale cache entries and keeping cache consistency. Furthermore, redundant data and duplicated files are automatically de-duplicated, which in some use cases (backups, scientific software binaries) reduces the actual storage space utilization by many factors [12, 35].

Cryptographic hashes are also used to protect the integrity of the file system tree when combined with a Merkle tree [36]. In a Merkle tree, nodes recursively hash their children's cryptographic hashes so that the root hash uniquely identifies the state of the entire file system. Copies of this root hash created at various points in time provide access to previous snapshots of file systems, which effectively allows for backups and for versioned file systems. The hashes in the tree can also be cryptographically signed in order to ensure data authenticity of a file system or a subtree (*who* created the content). An elegant way to solve the problem of key distribution inherent to digital signatures is the encoding of the public key as part of the path name [37].

To protect against *silent corruption*—the probabilistic decay of physical storage media over time—simple checksums such as CRC32 provide an easy means. Checksums can be faster verified than cryptographic hashes, fast enough to compute them on the fly on every read access [10].

4.2. Fault-Tolerance

Fault-tolerance is an important property for a file system in high-energy physics because of the large scale of the storage systems and because mostly commodity hardware is used. Thus, hardware failures are not only the norm but they also tend to occur in a correlated manner. Power cuts are an example, or a failing controller with many connected drives.

Replication and *erasure codes* are the techniques used to avoid data loss and to continue operation in case of hardware failures. An engineering challenge is the placement of redundant data in such a way that the redundancy crosses multiple failure domains. The Ceph file system, for instance, can parse the physical layout of a data center together with policies to distribute redundant data on multiple racks or disk shelves. Another option is to keep a copy of all data at a remote data center. The second engineering challenge is to decide when to start a recovery, i. e. to distinguish between short-term glitches and permanent faults. Tight monitoring can help to distinguish between the two in some cases, but in general the only available technique are heartbeats between servers and properly tuned time-out values [38].

While replication is simple and fast, it also results in a large storage overhead. Most systems use a replication factor of three. Erasure codes, on the other hand, can be seen as more sophisticated, distributed RAID systems. Every file is chunked and additional redundancy blocks are computed. Typically the storage overhead of erasure codes is much smaller than a factor of two. Erasure codes, however, come at the cost of computational complexity. Modifications to a file as well as recovery from hardware faults is expensive because the redundancy blocks have to be recalculated and redistributed. Exploration of the trade-off between computational complexity and storage

overhead in erasure codes are an active research area. Only a few of today's open-source distributed file systems implement erasure codes [9, 11, 14].

In order to determine the required level of redundancy and in order to balance the resources assigned to recovery and the resources assigned to normal usage, it is necessary to *predict* the failure rate of storage systems. This is a hard problem. In large scale storage systems, the mean time to failure of components can be measured, and simulations or a Markov chain model can be used to reason about the failure rate of the system as a whole. Although state of the art, the failure probabilities returned by these approaches are often off by large factors [38].

4.3. Efficiency

Caching and file striping are standard techniques to improve the speed of distributed file systems. Caches can be located in memory, in flash memory, or on hard disks. They are transparently filled on access; opportunistic pre-placement of data sets is typically not part of distributed file systems but it is implemented in data management systems on top of distributed file systems. Cache sizes needs to be manually tuned according to the working set size of applications. Caches are most often managed per file system node. Co-operative caches between nodes in a local network have been discussed [33, 39, 40] but they are not implemented in today's production file systems. Some distributed file systems, however, support the notion of different *pools* with different performance characteristics (e. g. flash memory, hard drives, and tapes) and automatic migration of data sets between such pools [4, 8, 41].

Striping, reading and writing a file in small blocks on many servers in parallel, works only for large files. For small files, (memory) caches can improve the read performance but the write performance on hard drives can suffer from the `seek` operations necessary to write data and meta-data to disk. Log-structured file systems provide near-optimal write performance for small and large files because all changes, new data and meta-data are appended to the physical medium [42, 43]. While log-structured data organization is popular in distributed key-value stores, it is only occasionally used in distributed file systems, possibly because most of them assume large sequential writes. In high-energy physics, this assumption is often valid but not always. Counter examples are the modifications to a file system namespace or the merging phase of a parallelized physics analysis with many small distributed parts of the final result. Moreover, log-structured data organization is an efficient means to manage storage space throughout the memory hierarchy, for DRAM, flash memory, and hard disks [44].

Dynamic workload adaptation is a technique used in the Ceph file system to change the mapping of meta-data to meta-data servers based on the server load. The file system tree is dynamically re-partitioned and re-distributed and single hot directories are split across multiple servers. The implementation is challenging though; a recent study still revealed stability problems [20].

5. Current Developments and Future Challenges

In the upcoming years, the computing landscape will move towards the *exascale*. That means data sets that routinely sum up to exabytes and supercomputers that provide computing power in the exaflop range, which are expected by 2020. Table 1 shows the development of storage bandwidth and capacity in the last 20 years. The rapid increase in storage capacity allows for exabyte data sets already today. At the same time, however, storage bandwidth developed not nearly at the same pace. While the gap between capacity and bandwidth widened by one to two orders of magnitude in the last 20 years, the bandwidth of Ethernet networks scaled at a similar pace than the capacity of hard drives.

Raicu et al. predict the collapse of exaflop supercomputing applications due to the limited storage bandwidth and the architecture of today's distributed file systems [46, 47]. They suggest to break the segregation between storage networks and compute networks and to build distributed file systems with the following characteristics.

Table 1. Development of capacity and bandwidth in the last 20 years. Method and entries marked † from Patterson [45]. Other numbers refer to Seagate ST6000NM0034 hard drive (2014), DDR2-400 DRAM (2004) and DDR4-3200 DRAM (2014), and the 100 GbitE IEEE 802.3bj standard (2014).

	Hard Disk Drives		**DRAM**		**Ethernet**
Year	Capacity	Bandwidth	Capacity	Bandwidth	Bandwidth
1993			16 Mibit/chip†	267 MiB/s†	
1994	4.3 GB†	9 MB/s†			
1995					100 Mbit/s†
2003	73.4 GB†	86 MB/s†			10 Gbit/s†
2004			512 Mibit/chip	3.2 GiB/s	
2014	6 TB	220 MB/s‡	8 Gibit/chip	25.6 GiB/s	100 Gbit/s
Increase	×1395	×24	×512	×98	×1000

‡http://www.storagereview.com/seagate_enterprise_capacity_6tb_35_sas_hdd_review_v4

(i) Every compute node becomes a part of the storage network, counting on the fact that the bisection bandwidth of the network connecting compute nodes is sufficiently high to move data.

(ii) Flash memory is explicitly used as a layer in the storage hierarchy and data locality is exposed in the file system, so that in many cases data can stay in local fast storage.

Similarly, Seagate Kinetic technology suggests to replace storage management servers by network-attached hard drives that can be directly addressed from applications [48]. And in the BigData community, a move has taken place to coalesce storage and compute nodes using GFS and MapReduce, resp. its open source counterpart Hadoop and recent real-time and in-memory enhancements such as Spark [49]. In high-energy physics, workflows are often executed in multiple stages (e. g. simulation and reconstruction, filter and analysis and merge) and often the result of stage n is explicitly written to a distributed file system only to be used as input for stage $n+1$. Investing in a computing framework and a distributed file system that exploits data locality and allows for keeping data at the node where they are produced would help to overcome the bandwidth limitations in today's and future storage hardware.

In the last few years, development efforts in distributed storage systems were mostly aimed at key-value stores, BLOB stores, and NoSQL databases rather than file systems. Due to their simpler interface, these technologies allow for very good scalability in local networks and fast provisioning of (mostly small) objects. As such, they become a building block of distributed file systems [9, 11] but they are not replacing them. They do not primarily address features necessary for data sharing, such as quotas, fine-grained access control, a standard interface to applications, or federation across multiple data centers.

Some recent distributed file system developments were aimed at storing and sharing personal data [50, 51]. Such systems explore extensions of the pure file system interface in order to support meta-data searches and instant dissemination of file system subtrees and document collections. These systems are decentralized and more flexible approaches to the problems addressed by Dropbox and AFS.

6. Conclusion

Distributed file systems provide a relatively well-defined and general purpose interface for applications to use large-scale persistent storage. The hierarchical namespace provides a natural and flexible way to store large amounts of unstructured data. The implementation of distributed file systems, however, is always tailored to a particular class of applications. Even though a large number of distributed file systems is available today, some practical and important use cases are still uncovered. For instance, a distributed file system for a cluster of commodity servers that provides at least half of the aggregated throughput of the hard drives, utilize at least 90 % of the available capacity, without central components and fault-tolerant to a small number of hardware failures would fit the storage needs of many small and mid-sized physics experiments [52].

For large-scale distributed file systems targeted at anticipated physics data sets in the exabyte range, the relatively small bandwidth of storage hardware poses a hard challenge. Approaches towards decentralized file systems that coalesce storage nodes and compute nodes and that make explicit use of locality and flash memory offer a possible solution to overcome such limits. Quantitative studies on the file system usage of physics applications should guide the selection and development of building blocks for future distributed file systems in physics experiments.

References

[1] Sandberg R, Goldberg D, Kleiman S, Walsh D and Lyon B 1985 *Proc. of the Summer USENIX conference* pp 119–130

[2] Morris J H, Satyanarayanan M, Conner M H, Howard J H, Rosenthal D S H and Smith F D 1986 *Communications of the ACM* **29** 184–201

[3] Carns P H, III W B L, Ross R B and Thakur R 2000 *Proc. 4th Annual Linux Showcase and Conference (ALS'00)* pp 317–328

[4] Schmuck F and Haskin R 2002 *Proc. 1st USENIX conf. on File Storage and Technologies (FAST'02)* pp 231–244

[5] Schwan P 2003 *Proc. of the 2003 Linux Symposium* pp 380–386

[6] Nagle D, Serenyi D and Matthews A 2004 *Proc. of the 2004 ACM/IEEE conf. on SuperComputing (SC'04)*

[7] Dorigo A, Elmer P, Furano F and Hanushevsky A 2005 *WSEAS Transactions on Computers* **4** 348–353

[8] Fuhrmann P and Gülzow V 2006 *dCache, Storage System for the Future* (Springer) pp 1106–1113 (*Lecture Notes in Computer Science* no 4128)

[9] Weil S A 2007 *Ceph: reliable, scalable, and high-performance distributed storage* Ph.D. thesis University of California Santa Cruz

[10] Shvachko K, Kuang H, Radia S and Chansler R 2010 *Proc. of the 26th IEEE Sympoisum on Mass Storage and Technologies (MSST'10)* pp 1–10

[11] Peters A J and Janyst L 2011 *Journal of Physics: Conference Series* **331**

[12] Blomer J, Aguado-Sanchez C, Buncic P and Harutyunyan A 2011 *Journal of Physics: Conference Series* **331**

[13] Davies A and Orsaria A 2013 *Linux Journal*

[14] Ovsiannikov M, Rus S, Reeves D, Sutter P, Rao S and Kelly J 2013 *Proc. of the VLDB Endowment* vol 6 pp 1092 – 1101

[15] Satyanarayanan M 1990 *Annual Review of Computer Science* **4** 73–104

[16] Guan P, Kuhl M, Li Z and Liu X 2000 A survey of distributed file systems University of California, San Diego

[17] Agarwal P and Li H C 2003 A survey of secure, fault-tolerant distributed file systems http://www.cs.utexas.edu/users/browne/cs395f2003/projects/LiAgarwalReport.pdf

[18] Thanh T D, Mohan S, Choi E, Kim S and Kim P 2008 *Proc. int. conf. on Networked Computing and Advanced Information Management (NCM'08)* pp 144 – 149

[19] Depardon B, Séguin C and Mahec G L 2013 Analysis of six distributed file systems Tech. Rep. hal-00789086 Université de Picardie Jules Verne

[20] Donvito G, Marzulli G and Diacono D 2014 *Journal of Physics: Conference Series* **513**

[21] Dean J and Ghemawa S 2008 *Communications of the ACM* **51** 107–114

[22] Doraimani S and Iamnitchi A 2008 *Proc. 17th int. symposium on High performance distributed computing* pp 153–164

[23] Leung A W, Pasupathy S, Goodson G and Miller E L 2008 *Proc. of the USENIX Annual Technical Conference* pp 213–226

[24] Dayal S 2008 Characterizing hec storage systems at rest Tech. Rep. CMU-PDL-08-109 Carnegie Mellon University

[25] Ghemawat S, Gobioff H and Leung S T 2003 *ACM SIGOPS Operating Systems Review* **37** 29–43
[26] Thain D and Livny M 2005 *Scalable Computing: Practice and Experience* **6** 9
[27] Henk C and Szeredi M Filesystem in Userspace (FUSE) http://fuse.sourceforge.net URL http://fuse.sourceforge.net/
[28] Lamport L 1998 *ACM Transactions on Computer Systems* **16** 133–169
[29] Ongaro D and Ousterhout J 2014 *Proc. of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)* pp 305–319
[30] Burrows M 2006 *Proc. 7th symposium on Operating systems design and implementation* pp 335–350
[31] Hunt P, Konar M, Junqueira F P and Reed B 2010 *Proc. of the 2010 USENIX annual technical conference*
[32] Kubiatowicz J, Bindel D, Chen Y, Czerwinski S, Eaton P, Geels D, Gummadi R, Rhea S, Weatherspoon H, Weimer W, Wells C and Zhao B 2000 *ACM SIGPLAN Notices* **35** 190–201
[33] Dabek F, Kaashoek M F, Karger D, Morris R and Stoica I 2001 *ACM SIGOPS Operating Systems Review* **35** 202–215
[34] Kutzner K 2008 *The Decentralized File System Igor-FS as an Application for Overlay-Networks* Ph.D. thesis University of Karlsruhe
[35] Quinlan S and Dorward S 2002 *Proc. of the 1st USENIX Conf. on File and Storage Technologies (FAST'02)* pp 89–102
[36] Merkle R C 1988 *A Digital Signature Based on a Conventional Encryption Function* (*Lecture Notes in Computer Science* vol 293) (Springer) pp 369–378
[37] Mazières D, Kaminsky M, Kaashoek M F and Witchel E 1999 *ACM SIGOPS Operating Systems Review* **34** 124–139
[38] Ford D, Labell F, Popovici F I, Stockly M, Truong V A, Barroso L, Grimes C and Quinlan S 2010 *Proc. 9th symposium on Operating Systems Design and Implementation (OSDI'10)*
[39] Nelson M N, Welch B B and Ousterhout J K 1988 *ACM Transactions on Computer Systems* **6** 134–154 ISSN 07342071
[40] Annapureddy S, Freedman M J and Mazières D 2005 *Proc. of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)* pp 129–142
[41] Lo Presti G, Barring O, Earl A, Rioja R M G, Ponce S, Taurelli G, Waldron D and Santos M C D 2007 *Proc. of the 24th IEEE Conference on Mass Storage Systems and Technologies* pp 275–280
[42] Rosenblum M and Osterhout J K 1991 *ACM SIGOPS Operating Systems Review* **25**
[43] Hartman J H and Osterhout J K 1995 *ACM Transactions on Computer Systems* **13** 274–310
[44] Rumble S M, Kejriwal A and Ousterhout J 2014 *Proc. 12th USENIX Conference on File and Storage Technologies (FAST'14)*
[45] Patterson D A 2004 *Communications of the ACM* **47**
[46] Raicu I, Foster I T and Beckman P 2011 *Proc. 3rd int. workshop on Large-scale system and application performance (LSAP'11)* pp 11–18
[47] Zhao D, Zhang Z, Zhou X, Li T, Wang K, Kimpe D, Carns P, Ross R and Raicu I 2014 *Proc. of the 3rd IEEE int. conf. on Big Data (BigData'14)*
[48] Seagate Kinetic open storage documentation wiki https://developers.seagate.com/display/KV/Kinetic+Open+Storage+Documentation+Wiki
[49] Engle C, Lupher A, Xin R, Zaharia M, Li H, Shenker S and Stoica I 2012 *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*
[50] Fitzpatrick B *et al.* Camlistore http://camlistore.org
[51] Mashtizadeh A J, Bittau A, Huang Y F and Mazières D 2013 *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP'13)*
[52] Allen B 2009 A distributed file system wish-list Talk at Max Planck Institut für Informatik, Saarbrücken