Distributed File Systems Past, Present and Future
A Distributed File System for 2006

Fabio Kon – *f-kon@cs.uiuc.edu*

March 6, 1996

1 Introduction

We start in 1.1 by citing some of the first attempts of implementing a distributed file system. In section 2 we present the most famous systems in the academic community, namely NFS, AFS, and SPRITE and some other experimental systems which brought important scientific contributions: CODA, ZEBRA, HARP, FROLIC, and ECHO.

Based on the knowledge of these systems characteristics and on current trends on computer systems, we finish on section 3 presenting our conclusions and thoughts on how a Distributed File System for 2006 should be.

1.1 The Beginning

The major part of the work on Distributed File Systems was carried after the middle eighties. Nowadays there are tens of different implementations of this kind of system.

In the very beginning of the Distributed Systems history one can find the *Datacomputer* which used to support an FTP-like service for clients which did not have a large amount of local storage[Svo84]. It started running on a PDP-10 in the end of 1973.

The *Interim File Server* (IFS) was created two years later in the XEROX PARC. It was able to organize public and personal files on a hierarchical directory tree. The same research center brought the *Woodstock File Server* [SMB79]. In this system it was possible to access single pages of a file which allowed the use of diskless clients and over the network virtual memory.

In the following years, a lot of different systems like XDFS, LOCUS, SWALLOW, ACORN, and CMU's VICE were designed and implemented [Svo84].

After that, it was the time for the creation of the currently most used distributed file systems like Sun's NFS and CMU's AFS. In the following section we will describe not only these two systems but also some other systems which brought relevant contributions to the research in this field.

2 Case studies

2.1 NFS

The *Network File System* developed by Sun Microsystems is the most widely used distributed file systems in the UNIX world. Upon releasing the first versions of NFS in 1985, SUN made public the *NFS protocol* specification [SUN94] which allowed the implementation of NFS servers and clients by other vendors. This protocol defines an RPC interface using the external data representation standard XDR [SUN89]. Today it is possible to find NFS implementations for almost all existing computers and operating systems including non-UNIX systems like MVS, MacOS, OS/2 and even MS-DOS.

NFS servers are, by definition, stateless, i.e., they do not store information about the state of client accesses to its files. Should a server crash, no such information is lost and the recovery can be immediate.

On the other hand, stateless servers cannot control the concurrent accesses to its files and therefore NFS does not assure the consistency of its file system. Different clients can have different and conflicting copies of the same file or directory in their local cache.

When a file is updated by one client, this modifications may not be noticed by other clients during a period of up to 6 seconds. When a file is created or deleted, this fact can take up to 60 seconds to be perceived by other clients. If one needs a coherent sharing of information throughout the distributed system, some other mechanism – like message passing – must be used.

Also because of its stateless servers, NFS is not capable of managing locks nor atomic transactions.

The name space on each client can be different but, from the user point of view, it is location transparent. It is the job of the system administrator to determine how each client will view the directory structure. This can be done by editing the `/etc/fstab` file – which statically binds mount points to server directories – and by editing the *automounter* configuration files which allows dynamic bindings and some degree of replication on read-only directories.

The user level implementation based on RPC/XDR and the client caching policy make NFS's performance worse than most of the systems we describe bellow. Currently however, NFS is undoubtfully the most used distributed file system on networks of workstations.

2.2 ANDREW

The ANDREW project [HKM+88, Sat90, Tan92] started at the Carnegie Mellon University in 1983 with IBM's support. Its goal was to design and implement an ideal distributed file system for the academic environment which would allow the sharing of a common directory structure among thousands of client machines.

Besides scalability, the other important problem they had to face in order to accomplish that goal was security. With so many clients and users, it would be impossible to trust on every client and not weaken the security of the whole system. The security problem was solved – or, at least, restricted – by using the Kerberos protocol [SNS88] for mutual authentication between client and servers.

The scalability problem was address by taking a large amount of work from the server to the client. When a client opens a file under the ANDREW File System (AFS), the whole file[1] is fetched from the server. All subsequent `read` and `write` operations use a local copy stored on a local disk. Only when the file is closed – and if it was modified – it is transferred back to the server.

A consequence of this technique is that AFS presents *session semantics* rather than UNIX semantics. A client C_1 can only perceive an update to a file F made by another client C_2 if C_1 opens F after C_2 has closed it.

AFS-2 brought the concept of a *callback* which allowed a client to open and close a file many times without having to access the server. When the client receives a file from a server, it also receives a callback which is a promise that the client is holding the most recent version of this file. A callback can be broken either by the client – when it updates the file – or by the server – when

[1] Indeed since AFS-3, files are transferred as needed on blocks of 64 Kbytes to reduce the open latency and to allow the access to files which would not fit into the local disk.

it receives a new version of the file from another client. The local copy can be used many times as far as the client holds a valid callback.

AFS supports a simple replication scheme which can be used for automatic backup of user files and for replication of read-only directories.

Currently there are more than 100 AFS cells all over the world giving its users the possibility of sharing files across different continents using a UNIX-like file system interface.

2.3 CODA

The CODA system [SKK+90], implemented in the early nineties, is a descendent of AFS. Its main goal was to provide access to a distributed file system from portable computers. CODA also implements automatic replication mechanisms not present on AFS.

Under CODA, each volume (a set of file system directories) is associated with a *volume storage group* (VSG), a set of servers that replicates the volume. The set of accessible servers from a particular group at a particular moment is called AVSG. The coherency among the several copies of a file is maintained using AFS-like callbacks.

When a client sends an updated file to a server, the update is forwarded to all AVSG servers using a multiRPC [SS90] mechanism. If a server that was down recovers from a crash, nothing is done at this moment to update its files. However, whenever a client sends a request for fetching a new file to its *preferred* server, it also sends a request for the file version number to all servers in the AVSG. Therefore, the client can find out if there is a server with an old version of the file and advise it to get an update copy.

Disconnected Operation

The most interesting CODA feature is the possibility of accessing distributed file system files without being connected to the network at all. If a file is locally cached on a portable computer disk, the user can read and update the file without the server permission.

When the portable computer is reconnected to the network, the system tries to propagate to appropriate servers the updates made during the disconnected period. Should conflicts occur – e.g. two different updates to the same file made by different clients – CODA provide some tools for the user to decide which update must prevail.

In order to allow access to files while disconnected, the CODA client tries to keep in its local disk the largest possible amount of files which the user might need if the connection with the server is lost. This is accomplished by a mechanism which assigns priorities to each file based on the last time the file was accessed by this client and an explicit list of "interesting" files provided by the user, the *hoard database*.

Once each 10 minutes, a process called *hoard walk* is initiated in order to bring to the local disk all files with larger priorities. The idea is that all client working set should be stored on the client disk. If the local disk is full and a new file must be copied into it, the cached file with the lowest priority is discarded from the local disk.

We can see that CODA does not even respect the session semantics. But this is something one must tolerate if one wants to have the extra availability provided by CODA.

2.4 SPRITE

The SPRITE Network Operating System [OCD+88] has been developed in the University of California at Berkeley since the middle eighties. It is not only a file system or a collection of network services but something closer to what one can call a *Distributed Operating System.* SPRITE has a transparently distributed file system and a mechanism for process migration [DO91] which can be transparent both to the process and to its user. Since this paper is on distributed file systems, we will only address SPRITE's file system.

The two main goals of the SPRITE file system designers were high performance and UNIX semantics on file sharing. The achieved performance was one of the best of their times, certainly better than NFS and AFS. The UNIX semantics was also achieved in the case where no failures occur.

High Performance

Previous experiments [O$^+$85] had shown that huge client and server caches could improve the performance considerably. The same experiments had shown that a large amount of file system work is devoted to temporary files whose lifetime is no more than some seconds.

In order to make both client and server caches as big as possible, SPRITE implements a scheme in which the file system and the virtual memory manager compete for physical memory. Usually, all physical memory is allocated for both systems and the file system cache can get up to tens of megabytes of the available memory. Indeed, SPRITE designers suggest that file servers should be dedicated machines with hundreds of megabytes of RAM which would be used almost entirely for file caching.

In order to reduce network and disk traffic SPRITE caches not only reads but also writes to the file system. Once every 30 seconds, all dirty cached blocks which were not modified in the last 30 seconds are sent to the server. A written piece of data can take up to 60 seconds to be sent to the server and then, up to 60 seconds to be written to disk. This procedure saves a lot of network and disk traffic. Temporary files, for example, are never sent to the server.

On the other hand, this delayed-write scheme makes the system much more sensitive to failures. When a user saves a file using a text editor, it can take up to 2 minutes to get to a safe place (the hard disk). The updates to the file may be completely lost if, meanwhile, the client or the server crashes.

Exhaustive experiments [BHK$^+$91, NWO88] have shown that SPRITE presented very good performance on local networks outperforming both NFS and AFS.

UNIX Semantics

Provided that neither the client nor the server goes down, the SPRITE file system users can access the file system concurrently and perceive the same kind of semantics found on a local UNIX machine. In other words, every call to the `read` system call returns the data written by the last `write` call to the same location.

This UNIX semantics is accomplished by disabling client caching when a file is "concurrent write shared". Concurrent write sharing is defined as a state in which more than one client has a particular file opened and at least one these clients has this file opened for writing.

When caching for a particular file is disabled all read and write operations must be done directly with the server and the UNIX semantics is obviously achieved. It is clear that when caching is disabled the performance is strongly affected. However, studies have shown [BHK$^+$91] that concurrent write sharing rarely occurs and therefore the gain of having a file system in which one can trust can compensate the loss in performance.

Differently from NFS, SPRITE servers are not stateless, they must keep track of which clients are doing what with its files. This can cause problems when a server crashes because it must rebuild its state upon rebooting. Mary Baker has worked hard on this topic and her PhD thesis [Bak94] presents some mechanisms that allowed the rebooting time for a SPARCStation 2 with 40 clients to be reduced to just 30 seconds (including the file system verification).

Name Space

Differently from the previously described systems, the SPRITE name space is the same for all system clients. The pathname resolution scheme is based on prefix tables, a mechanism presented in [WO86]. When a client needs to resolve a pathname, it looks for the largest prefix of this pathname on a local prefix table. There, it can find the server which handles the requested file or directory. It then sends to this server a handle corresponding to the found prefix plus the rest of the pathname.

If the server can resolve all the rest of the pathname, it returns to the client a file handle which is used on the subsequent accesses to this file. On the other hand, if it cannot resolve the pathname it just returns a code to the client indicating that.

If the server answers to the client that it cannot resolve the requested pathname, the client sends a broadcast to all network servers asking which is the server that handles this pathname. If a server replies, the client adds one more entry to its local prefix table. Otherwise, it gives an error message to the user.

Clients start with empty prefix tables and with a broadcast to find the root directory server. Other entries are added as need arises. If a client try to access a domain present in its prefix tables but does not receive a reply, it sends a new broadcast to all servers asking which is the server for that domain. This behavior allows the easy implementation of domain migration. When a domain migrates from one server to the other, its clients automatically adapt to the new configuration without the need for the administrator interference[2].

Differently from NFS and AFS which resolves pathnames component by component, the prefix tables allows SPRITE to access the appropriate server immediately (provided that the requested pathname is in the prefix table). NFS tries to reduce the overhead of analyzing pathnames component by component by caching directory information (something that SPRITE does not do).

The use of broadcasts in the SPRITE protocol seriously compromises its scalability. However, SPRITE was designed to work on local networks and presented a very good performance in this kind of environment.

It's also important to point out that SPRITE security mechanisms are very limited and sometimes even weaker than UNIX. One of these weaknesses arises from the fact that prefix tables allow a user to access a directory, say `/a/b/c/` without checking for the UNIX permission bits for directories `/a/` and `/a/b/` if the prefix `/a/b/c` was previously added to the prefix table.

2.5 ZEBRA

The ZEBRA system [HO92, HO93] has been developed at University of California at Berkeley since 1990. It joined two powerful ideas originally conceived for local file systems on a new distributed file system concept. These two concepts were Redundant Arrays of Inexpensive Disks (RAIDs) [L+92] and Log-structured File Systems [RO91].

In the last years, mainly after the RISC technology was introduced, the microprocessor speeds have increased much more than the disk access speeds [Ous90]. One of the consequences of this fact is that the disk access times started to be the major bottleneck on various applications and systems.

[2] Obviously, the administrator must reconfigure both servers involved on the migration.

The use of huge caches can avoid a lot of read accesses to the disk but has not the same effect on write accesses which cannot be cached for a long time. On log-structured file systems like SPRITE LFS [RO91] a large amount of small write operations can be grouped together and sequentially transferred to the disk avoiding unwanted seek times. Once a disk block is written it is not updated anymore[3]. Every new data is concatenated to the end of the log making the write operations much faster. ZEBRA uses the same idea to group writes to a distributed file system.

By stripping files through different servers – as RAIDs do within different disks on a single server – ZEBRA manages to evenly distribute the workload among them. Besides avoiding a situation in which a server is idle while another one is flooded with requests, an application running on ZEBRA can have various disk requests being processed on parallel by different servers.

A redundant logical server is maintained in order to keep the system availability on acceptable levels. So all system files are kept accessible even if one server goes down. If desired, the system can be configured to maintain a higher level of redundancy and therefore a higher availability.

Some tests have shown that ZEBRA's performance is much better than SPRITE and other systems'. However, most part of the measured workloads were built in such a way that this result would be obtained. More exhaustive measurements are required.

It seems that the major problems with this system is still the availability and scalability since it requires that two fundamental system managers be centralized on a single machine. More details about ZEBRA implementation can be found on the cited papers.

2.6 HARP

HARP (Highly Available, Reliable, Persistent file system) [LGJ+91] is an experimental system developed at MIT in the early nineties.

It offers a highly fault tolerant file service by adopting a primary-backup-witness pessimistic replication scheme. It runs on conventional hardware requiring only inexpensive uninterruptible power supplies (UPSs) for keeping the power up for a couple of minutes when there is a failure in the power supply.

Since any file system operation must be approved by a majority of servers the system performance is obviously worse than all previously described systems. Therefore, such a scheme should only be used if the integrity of the file system data is really essential.

2.7 FROLIC

FROLIC [PGZ92, SZ92] has been developed in the nineties at the University of Toronto. It is based on the assumption that file sharing on wide are networks may be much larger than what was expected by the AFS team.

The system is divided on clusters. Clients can only perform intra-cluster communication with local servers using conventional NFS. Servers, however, can perform inter-cluster communication in order to fulfill client requests. Servers dynamically create local replicas of remote files and the consistency control between replicas on different clusters obeys session semantics.

Experiments on a wide area Engineering company network [GZS94] showed major improvements on lowering the intra-cluster traffic.

[3] Indeed, it can be rewritten much later, after a "garbage collection" operation has been performed.

2.8 ECHO

The ECHO distributed file system [BHJ+93] was a piece of an ambitious project carried since 1988 at the DEC research center in Palo Alto. ECHO joined good qualities of systems like AFS (scalability), SPRITE (UNIX semantics), HARP (fault tolerance) and presented some new features like a coherent distributed file cache with directory write-behind.

Even though it presented the best system design of its time[4], its implementation was suspended in the beginning of 1992 when the project of porting TAOS (the operating system in which it was based) to a new hardware platform was canceled. ECHO was used by 60 users from January, 91 to middle 92.

The fault tolerance scheme [SBHM93] is based on the replication of the disks of a single server – which increases data integrity – and on the "replication of the servers of a single disk" – which increases the system availability. The latter requires a special hardware called *Autonet* to avoid that two servers access the same disk at the same time. A pessimistic replication algorithm – similar to the one implemented by HARP – is used.

ECHO was the first system to implement write-behind on all kind of data including directories [MBH+93]. In order to avoid the destruction of the file system integrity when a client or server crashes, ECHO assures that the written data will get to the disk on a well defined partial order which can be refined by the user by calling the new `forder(fildes)` system call.

The cache consistency is maintained by a very efficient and fault tolerant mechanism called *leases* [GC89, KM95]. This mechanism allows accessing cached files even when they are being write-shared by different clients.

Two of the most interesting ECHO features, however, were not completely implemented. The first one is the integration of the local naming service with the Internet domain name service (DNS). The name space was intended to be uniform on all ECHO clients in the world. For example, the pathname `/-/com/dec/src/dir/arq` would be used to access the file `arq` in the directory `dir` of the Internet node `src.dec.com`[5].

Ironically, the other not implemented feature may be the most important ECHO contribution. The idea is that, within a file server domain, some files called junctions would contain pointers to other servers directories. So, in this point, they are similar to NFS mount points and SPRITE remote links. However, the junctions would also contain some information about the protocol which should be used to access the corresponding sub-tree.

Therefore, the system administrator could specify what protocols and semantics each of the file system volumes would present. So, a directory with very important data which must not be lost and must be consistent could be replicated using a pessimistic algorithm. Not so important shared data could use an optimistic algorithm. Directories shared by distant clients could use session semantics. Directories whose files must be fast accessed could use write-behind and leases for all kind of information. In other words, each directory could use the most sensible protocol for its kind of data.

2.9 Comparative Charts

The following table presents a comparative summary of the previous systems most important characteristics.

[4] ECHO may possibly be the best distributed file system designed until the present moment.

[5] The symbol `/-/` just indicates that the following address is not local and must be resolved by the DNS module.

	Sun NFS	**SPRITE**	**ZEBRA**	**ANDREW**	**CODA**	**FROLIC**	**ECHO**	**HARP**
Client CACHE LOCATION	Main Memory	Main Memory	Main Memory	Local Disk	Local Disk	Main Memory	Main Memory	Main Memory
REPLICATION	POOR. Just for read–only directories	POOR. Just for read–only directories	POOR. Just for read–only directories	POOR. Just for read–only directories	GOOD. Overhead is distributed among clients.	GOOD. Dynamic replication based on clusters.	EXCELLENT. Tolerance to server and disk crashes.	GOOD. Tolerance to server crashes.
CONSISTENCY	POOR. Concurrent access generate unpredictable results.	EXCELLENT. UNIX semantics.	EXCELLENT. UNIX semantics.	FAIR. Session semantics.	POOR. Session semantics weakened by optimistic replication.	POOR. Session semantics on intercluster access and *NFS semantics* inside each cluster.	EXCELLENT. Uses *leases* to assure UNIX semantics.	POOR. Uses NFS for client–server communication.
SCALABILITY	POOR. Servers saturate rapidly.	POOR. Protocol requires broadcasts.	FAIR. Distributes workload among servers.	EXCELLENT. Ideal for wide area networks with low degree of file sharing.	EXCELLENT. Ideal for wide area networks with low degree of file sharing.	EXCELLENT. Even for wide area networks with higher degree of file sharing.	FAIR. Interact with Internet DNS.	POOR. Generate even more overhead than NFS.
PERFORMANCE	POOR. Inefficient protocol.	GOOD. Kernel implementation. File contents delayed writes. Huge cache.	EXCELLENT. File striping. Log file system.	FAIR. Large latency on non–cached files, though.	GOOD. Looks for the "closest" replica.	GOOD. Clients talk only to local servers.	EXCELLENT. Delayed writes for all kind of information.	POOR.
DATA PERSISTENCE when fault occurs	POOR. Delayed writes may cause loss of data.	POOR. Long delayed writes may cause loss of data.	GOOD. Tolerates up to one disk or server crash in each stripe.	FAIR. Automatic backup tools.	GOOD.	GOOD.	GOOD.	EXCELLENT.
AVAILABILITY	POOR.	FAIR. Fast recovery from crashes.	GOOD. Tolerates client and server crashes. Fast crash recovery.	POOR.	EXCELLENT. Optimistic replication. Disconnected operations.	GOOD.	Theoretically EXCELLENT. Bad implementation brought problems.	GOOD.
SECURITY	POOR. Servers trust on clients.	POOR. Servers trust on clients.	POOR. Servers trust on clients.	GOOD. Access control lists. Kerberos authentication between client and servers.	GOOD. Access control lists. Kerberos authentication between client and servers.	POOR.	GOOD. Access control lists. Encrypted authentication.	POOR.
Some of the PLATFORMS	All relevant.	SPARCstations and DECstations.	DECstation 5000/200	Workstations by SGI, HP, Next, DCE, IBM, SUN, and VAX.	IBM RT, DECstations 3100 and 5000, and IBM/PC 386/486.	Prototype developed on SPARCstations' NFS.	FIREFLIES (VAX experimental multiprocessor).	Prototype developed on MicroVAX 3500's NFS.
Some HARDWARE characteristics.	Allow diskless clients.	Allow diskless clients.	Allow diskless clients.		Allow the use of portable computers. Replication requires extra disks and servers.	Replication requires extra disks and servers.	Uses special board for server replication. Replication requires extra disks and servers.	Requires UPS. Replication requires extra disks and servers.

The following diagram describes some of the tradeoffs between pairs of distributed file system characteristics and mechanisms. A continuous arrow from x to y roughly means that by increasing x one can expect an increase on y. A non-continuous arrow from x to y means that a increase on x usually leads to a decrease on y. The labels on the arrows contain the name of a system where the corresponding tradeoff can be observed.

An interesting relation exists between replication and efficiency. Optimistic replication may lead to an increase on efficiency while pessimistic replication usually leads to a decrease on efficiency. The arrow from heterogeneity to scalability just signs that using a system that can run on a large number of different platforms – like NFS – it is easier to build a larger distributed system because more machines would be available. NFS is well known for its poor scalability.

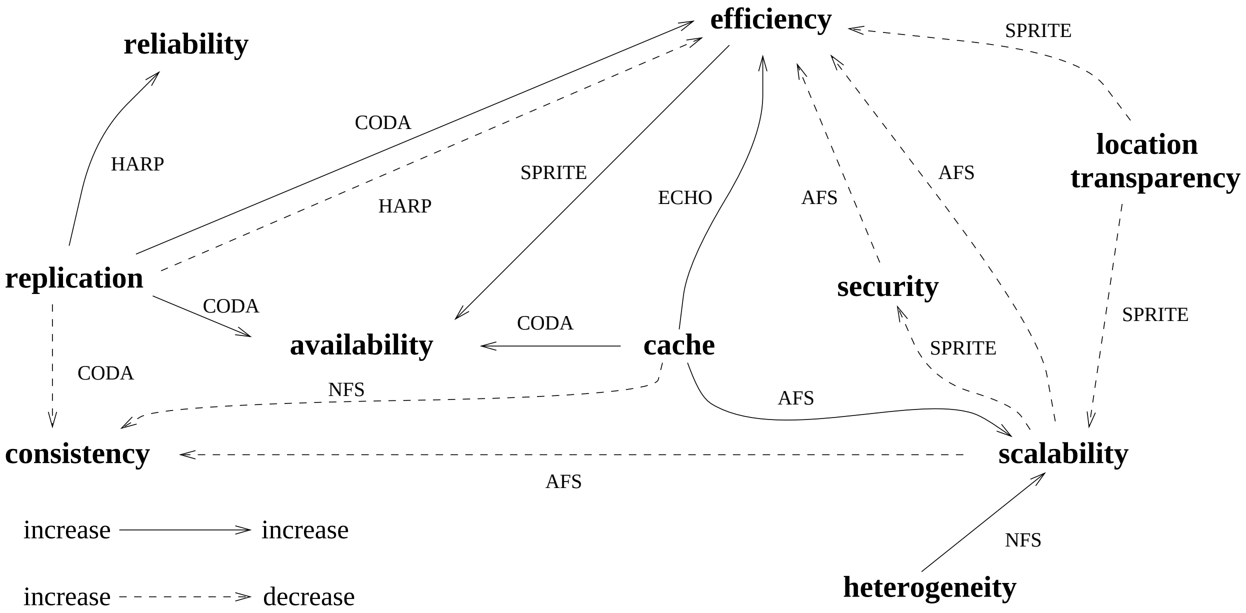

Figure 1: Distributed file systems tradeoffs

3 The Future

Current distributed file systems presents some problems which the research community should try to solve. Besides, new kind of applications – mainly that which deals with multimedia and "continuous data" like video and audio – are bringing new requirements to local and distributed file system design.

Now, we enumerate some of the most important features a future distributed file system should support.

- **Transparency:** the file system must make easier user interaction and programmers' efforts by hiding aspects like data location, network type, operating system type, failure occurrences.

- **Global scale:** AFS-like systems give users in different continents the possibility of sharing files. In some years, all computers on Earth will be connected and the file system can be used as a tool for sharing information between them.

- **UNIX semantics:** this semantics provides a simple tool for synchronization and data sharing among processes on different machines. Most current systems don't provide this feature claiming that files are rarely shared by different clients. However, aren't files rarely shared **because** the systems don't provided coherency? The future will answer.

- **Automatic replication:** not only increases the system availability and reliability but also can offer the user more than one server allowing a choice based on efficiency and workload distribution.
- **Striping:** evenly distributes the workload and allow parallel transfers of single files.
- **Aggressive caching:** increases system performance by making the cache hit ratio as big as possible.
- **Log-structure:** increase data writing performance and decrease file system verification time.
- **Automatic compression:** by making use of idle CPU cycles decreases the storage space requirements and the data transfer times [Dou93].
- **Multimedia support:** multimedia applications deal with huge amounts of information which can currently get to terabytes of data and transfer rates of hundreds of megabytes per second. UNIX-like file systems are not proper for this kind of task. Alternatives like [MP91] designed at UCSD must be considered in order to decrease latency and increase transfer speed and file sizes.
- **Adaptation:** by monitoring the workload, the system should try to make future accesses the most efficient as possible.
- **Protocol flexibility:** as done by ECHO's junctions shown in 2.8 the system administrator must be able to determine which protocol will be used in each piece of the system name space.

As shown in section 2.9 there are some tradeoffs among most of this features. Tradeoffs that make the work of implementing a system containing all desired properties much harder. However, our role as researchers is to defy them and build a Distributed File System for 2006.

References

[Bak94] Mary Baker. *Fast Crash Recovery in Distributed File Systems.* PhD thesis, University of California, Berkeley, CA 94720, January 1994. Technical Report UCB/CSD 94/787.

[BHJ+93] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The echo distributed file system. Technical Report #111, DIGITAL Equipment Corporation Systems Research Center, Palo Alto, CA, September 1993.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 198–212, Pacific Grove, CA, October 1991.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice & Experience*, 21(8), August 1991.

[Dou93] Fred Douglis. On the role of compression in distributed systems. *ACM Operating Systems Review*, 27(2):88–93, 1993.

[GC89] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 202–210, December 1989.

[GZS94] Deepinder S. Gill, Songnian Zhou, and Harjinder S. Sandhu. A case study of file system workload in a large-scale distributed environment. Technical Report 296, Computer Science Research Institute - University of Toronto, 1994.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[HO92] John H. Hartman and John K. Ousterhout. Zebra: A striped network file system. In *Proceedings of the USENIX File Systems Workshop*, pages 71–78, Ann Arbor, Michigan, May 1992. USENIX.

[HO93] John H. Hartman and John K. Ousterhout. The zebra striped network file system. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 29–43, Asheville, NC, December 1993. ACM.

[KM95] Fabio Kon and Arnaldo Mandel. SODA: A lease-based consistent distributed file system. In Sociedade Brasileira de Computação, editor, *XIII Brazilian Symposium on Computer Networks*, 1995. Also Technical Report #9503 (MAC-IME/USP). Available at ftp://ftp.ime.usp.br:/pub/reports/comp/rt-mac-9503.ps.gz.

[L+92] Edward K. Lee et al. RAID-II: A scalable storage architecture for high bandwidth network file service. February 1992.

[LGJ+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 226–38, 1991.

[MBH+93] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. Technical Report #103, DIGITAL Equipment Corporation Systems Research Center, Palo Alto, CA, June 1993.

[MP91] Keith Muller and Joseph Pasquale. A high performance multi-structured file system design. *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 56–67, 1991.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[O+85] J. Ousterhout et al. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, December 1985. ACM.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Summer USENIX '90*, pages 247–256, Anaheim, CA, June 1990.

[PGZ92] James Y.C. Pang, Deepinder S. Gill, and Songnian Zhou. Implementation and performance of cluster-based file replication in large-scale distributed systems. Technical report, Computer Science Research Institute - University of Toronto, August 1992.

[RO91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[Sat90] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, pages 9–21, May 1990.

[SBHM93] Garret Swart, Andrew Birrell, Andy Hisgen, and Timothy Mann. Availability in the echo file system. Technical Report #112, DIGITAL Equipment Corporation Systems Research Center, Palo Alto, CA, September 1993.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasai, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–59, April 1990.

[SMB79] Daniel Swinehart, Gene McDaniel, and David Boggs. WFS: A simple shared file system for a distributed envoronment. In *Proceedings of the 7th ACM Symposium on Operating System Principles*, pages 9–17, December 1979.

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. Available at ftp://athena-dist.mit.edu, file /pub/kerberos/doc/usenix.{txt,PS}, March 1988.

[SS90] M. Satyanarayanan and Ellen H. Siegel. Parallel communication in a large distributed environment. *IEEE Transactions on Computers*, 39:328–48, 1990.

[SUN89] SUN Microsystems, Inc. XDR: External data representation standard. RFC 1014, Network Information Center, SRI International, 1989.

[SUN94] SUN Microsystems, Inc. NFS: Network file system version 3 protocol specification. Available at ftp://ftp.uu.net/networking/ip/nfs/NFS3.spec.ps.Z, February 1994.

[Svo84] Liba Svobodova. File servers for network-based distributed systems. *ACM Computing Surveys*, 16(4):353–398, December 1984.

[SZ92] Harjinder S. Sandhu and Songnian Zhou. Cluster-based file replication in large-scale distributed systems. *ACM SIGMETRICS - Performance Evaluation Review*, 20(1):91–102, June 1992.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[WO86] B. B. Welch and J. K. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed filesystem. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 184–189, Boston, Mass., May 1986. IEEE.