A Survey of Distributed File Systems

Ping Guan, Martin Kuhl, Zhimin Li, Xiaoyu Liu
Dept. Computer Science & Engineering, University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114

Abstract: *An ideal distributed file system would be highly available, fault tolerant, scalable, high performance, transparent, provide tight file semantics, allow disconnected operation, be easy to maintain, have good security, and would work in heterogeneous environments. Unfortunately, existing distributed file systems can only achieve a small subset of these requirements. Many of these needs conflict, and compromises and trade-offs are required. This paper discusses these issues, and shows how they are addressed in different distributed file system implementations. Case studies of well known file systems, as well as newer experimental systems, are presented.*

1 Introduction

To appreciate and understand the challenges of designing a distributed file system, it is helpful to consider what an ideal distributed file system would be. First, it would provide high availability to all users at all times. The failure of one or more file servers or network links would not prevent a user from accessing his files. It would provide tight file semantics: a write to a file by any client would be immediately visible to all other clients. Atomic transactions would be supported. Fault recovery would be seamless and automatic. A minimum of human administration would be required. The system would function in heterogeneous environments. Individual users could operate disconnected from the rest of the system for extended periods and all file modifications performed would be automatically distributed to the rest of the system upon reconnection. System security would be tight: all users, clients, and servers would be authenticated, and network communications would be encrypted or some other means would be used to prevent eavesdropping. The system would be scalable: it would be possible to add any number of new users without having to reconfigure the existing hardware or software, and performance would be maintained. High bandwidth and low latency would be achieved for all file operations at all workstations. File naming would be independent of location and all users would see and use the same namespace.

Unfortunately, existing distributed file systems can only achieve a small subset of these requirements. Many of these needs conflict, and compromises and trade-offs are required. For instance, client caching of files is often used to provide high availability and good performance simultaneously. However, to ensure tight file semantics, some means are needed to maintain consistency between client caches. This requires mechanisms, such as clients checking with the server regularly or server call-backs to clients, that hurt performance and partially defeat the advantage of having a cache. Of course, one solution is for the system not to guarantee tight file semantics: clients read and write their caches, and any conflicts will not be discovered or resolved until the data is written back to the server.

File naming and name resolution is another area where conflicting requirements exist. One simple scheme is to concatenate the host name to the names of the files that are stored on that host. While this approach guarantees that a filename is unique systemwide, it conflicts with the goal of network transparency. Another problem with this approach is that moving a file from one host to another requires changing the filename and the applications accessing the file. A second approach is to mount remote directories onto local directories (make them a subtree). While this approach requires that the host of the remote directory be known, once it is mounted it can be referenced in a location transparent-manner. A third approach is to have a single global directory, where all files in the system belong to a single name space. This approach does not have the disadvantages of the above two naming schemes. However, because all filenames must be unique systemwide, it is not practical in systems that encompass heterogeneous environments or wide geographical areas, where a naming scheme suitable for one computing facility may be unsuitable for another.

Replication is the primary mechanism used in distributed file systems to enhance the availability of files. With replication, many copies of files are maintained at different servers. Inconsistency among replicas can result from a server failure, resulting in a file copy not being updated, or from a network partition, resulting in a situation where all the servers storing a copy of a file are not reachable from all clients. Several different schemes are possible for maintaining replica consistency. One approach is a weighted voting scheme. With this method, some number of votes and a timestamp are associated with each copy. A certain number of votes must be obtained before a read or a write can be performed. With this scheme it is not necessary to keep all the copies up-to-date as long as sufficient votes can be obtained to perform reads and writes. This feature provides for increased availability and fault tolerance during system failures. Another scheme to maintain replica consistency is to designate one or more processes as agents for controlling the access to replicas. In this approach, each filegroup has a designated site that enforces the global synchronization policy. File open and close requests are routed through the designated site to a storage site that has a copy of the requested file. The disadvantage of this approach is that the agent processes can become bottlenecks, hence it has poor scalability. The simplest approach is for the system not to guarantee consistency among replicas. When network partitions or other failures occur, copies can be updated and accessed independently. Any inconsistencies that result must be detected and resolved at a later time.

As systems grow in size, scalability can become a major problem. In traditional client/server distributed file systems, the server becomes a bottleneck as more clients are added, since all read misses and disk writes go through the central server. Cluster systems improve scalability by eliminating the central server and distributing the server functions across multiple machines. This maintains high performance as more workstations are added. Such systems generally have poor security, since all the machines must trust one another in order to maintain tight cooperation. As a result, the system must be contained in a secure physical environment, which limits its maximum size. Systems designed to handle thousands of workstations cannot rely on mutual trust between machines. Cluster systems can add clients outside the cluster by creating a traditional client/server interface between external clients and the cluster. This improves security, but also brings back all the scalability problems associated with traditional client/server systems.

Security is a major issue in distributed file systems, since the number of users is usually large, and the workstations are often in environments with limited physical security. Traditional client/server systems usually base security on the assumption that a limited number of trusted servers will communicate with a large number of untrusted workstations. Security is becoming more of a problem as the number and sophistication of computer users increases. Security will be discussed more in the case studies ahead.

Replication and back-ups have been the main methods of fault tolerance in traditional distributed file systems. More recently, striping of file data across multiple disks, with an addition disk used to store parity, has been used to increase disk bandwidth. This method also provides fault tolerance, since the data on a failed disk can be recovered from the data on the other disks and the parity data. Fault tolerance can have an adverse affect on performance, since it usually increases the complexity of write operations.

2 Case studies

2.1 NFS [7]

The Network File System developed by Sun Microsystems is the most widely used distributed file system in the Unix world. Upon releasing the first versions of NFS in 1985, Sun made public the NFS protocol specification, 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. Today it is possible to find NFS implementations for almost all existing computers and operating systems. NFS servers are by definition stateless: 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. Also because of its stateless servers, NFS is not capable of managing locks nor atomic transactions. The user level implementation based on RPC/XDR and the client caching policy make NFS's performance worse than most of the other systems described in this paper. Currently, however, NFS is undoubtedly the most widely used distributed file system on networks of workstations.

2.2 Andrew and Coda [8]

The Andrew file system was designed with the primary goals of scalability and security. Coda is similar to Andrew, but also seeks to provide high availability. The Andrew system consists of "Vice", a collection of file servers and networks, and the individual workstations attached to the networks. A process called "Venus", running on each workstation, mediates shared file access. Both Vice and Venus are invisible to workstation processes, which only see a Unix file system, one subtree of which is identical on all workstations.

In Andrew clients cache pathname prefix information and use it to direct file requests to appropriate servers. A mechanism known as a "callback" is used to notify a client when another workstation has modified a cached file. The RPC (remote procedure call) mechanism uses an optimized bulk-transfer protocol for file transfer. Data is organized in terms of volumes, a collection of files forming a partial subtree of the Vice name space. Volumes are usually small enough to allow multiple volumes per server disk partition and are easily moved between servers.

In 1988, work began on a new version of the Andrew file system called AFS-3. The revision was motivated by the need to provide decentralized system administration and by the desire to operate over wide area networks. AFS-3 supports multiple administrative cells, each with its own servers, workstations, system administrators, and users. The ability to decompose a distributed system into cells is important at large scale because it allows administrative responsibility to be delegated along institutional boundaries.

A consequence of large scale is that the casual attitude toward security typical of close-knit distributed environments is not acceptable. Rather than trusting thousands of workstations, Andrew predicates security on the integrity of the much smaller number of Vice servers. No user software is ever run on the servers. The protection domain in Andrew is composed of users and groups. AFS-1 and AFS-2 supported group inheritance, with a user's privileges being the cumulative privileges of all the groups it belongs to. The Andrew RPC mechanism provides support for secure, authenticated communication between mutually suspicious clients and servers. It uses a variant of the Needham and Schroeder private key algorithm. When a user logs on to a workstation, his password is used to obtain tokens from an authentication server. These tokens are saved by Venus and used as needed to establish secure RPC connections to file servers. The level of indirection provided by tokens improves transparency and security. Venus can establish secure connections to file servers without users' having to supply a password each time a new server is contacted, and passwords do not have to be stored in the clear on workstations. Andrew uses an access list mechanism for file protection.

The Coda file system is designed to provide constant data availability, allowing a user to continue working regardless of failures elsewhere in the system. Coda allows users to rely entirely on local resources when servers are inaccessible. Another goal of Coda is to integrate the use of portable computers. Coda provides failure resiliency through two distinct mechanisms. It uses server replication, or the storing of copies of files at multiple servers, and disconnected operation, a mode of execution in which the client relies solely on cached data. When network partitions occur, Coda allows data to be updated in each partition but detects and confines conflicting updates as soon as possible after their occurrence.

2.3 Echo [2]

Echo is a file system with the properties of global naming, global access, global security, and fault tolerance.

The name space of Echo is a tree with labeled edges and a single global common root. File names are paths from the root, consisting of a sequence of edges. The name space consists of volumes glued together at junctions. A volume is a subtree of the name space, residing on a set of servers. A junction is a leaf in one volume, containing the location and identity of a child volume. The junction is a static object, unlike an NFS mounting point. All clients in the system see the same volume in the same place in the name space. During name resolution, a file name is presented to an Echo file system switch, which calls the appropriate servers to access the volume. The global root volume is resolved by using the Domain Name Service (DNS) provided by the Internet. When an Echo client wants to access the root (DNS) volume, it uses the Echo name service clerk to talk to an Echo name server. This server either returns cached data or talks to the DNS servers on the client's behalf. Each clerk caches data from its servers. Most names are resolved entirely from the caches of clerk, without contacting any servers.

Echo implements UNIX semantics and provides uniform global access to the system. Changes made by one client are immediately visible to all the other clients. Echo provides a totally coherent cache (in main memory on the client). Because the updates are written-behind, the queued updates can be combined to a single stable storage operation for optimization. Echo uses a *token* scheme for file access. To get access to a file, a client must request a token from the server. The servers are responsible for detecting conflicting requests for tokens. If conflicts occur, the server calls back the clients to revoke the tokens. Echo uses *leases* to deal with client crashes that occur while the client is holding a token. A token is issued to a client with a lease. The token remains valid only as long as the lease has not expired, and it may be renewed by the client. If a client becomes detached by a network partition, the server will implicitly revoke all the tokens of the client.

The security of Echo is as good as a time-sharing system. Echo servers do not trust clients. All the communication between Echo servers and their clients is authenticated, and only RPC is used for the communications. The server always know the identity of the principal making or receiving an RPC call. These principals are identified by global names in the same name space that is used for file names. All Echo objects (files, directories, volumes, and servers) are protected by access control lists (ACL's), which specify what principals may perform what operations on the objects. An ACL is a set of names and access rights. The names in ACL's can be principals, or the names of other ACL's stored elsewhere in the global name space.

Echo is fault tolerant. Echo volumes are grouped into boxes, the basic unit of replication. Among the replicas of a box, one is elected as primary. All inquiries and updates are made at primary replica site. Echo uses a log for recording updates on each box's disk storage. The primary site updates its volatile data structures, writes a log record on its disk, and propagates the log record to all available replicas before returning to the client. Asynchronously, the servers write the updates to their disks, and remove the update records from the their logs. When the primary fails, a new primary is elected. The new primary reconciles with other active replicas and continues the service.

2.4 Serverless Network File Systems [1]

While traditional network file systems rely on a central server machine, a serverless system utilizes workstations cooperating as peers to provide all file services. Any machine in the system can store, cache, or control any block of data. The serverless approach uses this location independence, along with fast local area networks, to provide better performance and scalability than traditional file systems. Further, because any machine in the system can assume the responsibilities of a failed component, the serverless design also provides high availability via redundant data storage. A prototype serverless network file system called xFS has been developed at the University of California at Berkeley.

The recent introduction of switched local area networks such as ATM or Myrinet enables serverlessness by providing aggregate bandwidth that scales with the number of machines on the network. In contrast, shared media networks such as Ethernet or FDDI allow only one client or server to transmit at a time.

Current centralized file system designs fundamentally limit performance and availability since all read misses and disk writes go through the central server. Similarly, a central server represents a single point of failure, requiring server replication for high availability. xFS attempts to address these limitations in three ways. First, xFS dynamically distributes control processing across the system on a per file basis by utilizing a serverless management scheme. Second, xFS distributes its data storage across storage disks by implementing a software RAID using log-based network striping similar to Zebra's. Finally, xFS eliminates central server caching by using cooperative caching to harvest portions of client memory as a global file cache. The primary limitation of the xFS design is that it is only appropriate in a restricted environment, among machines that communicate over a fast network and that trust one another's kernels to enforce security.

xFS uses RAID style disk striping to provide high performance and highly available disk storage. A RAID partitions a stripe of data into N-1 data blocks and a parity block, the exclusive-OR of the corresponding bits of the data blocks. It stores each data and parity block on a different disk. The parallelism of RAID's multiple disks provides high bandwidth, while its parity storage provides fault tolerance: it can reconstruct the contents of a failed disk by taking the exclusive-OR of the remaining data blocks and the parity block. xFS uses single parity disk striping to achieve the same benefits.

xFS uses a log structured file system (LFS) because it provides high performance writes, simple recovery, and a flexible method to locate file data stored on disk. When used with a RAID, each segment of the log spans a RAID stripe and is committed as a unit to avoid the need to recompute parity.

To provide scalable control of disk metadata and cache consistency state, xFS splits management among metadata managers. xFS managers can dynamically alter the mapping from a file to its manager. Similarly, to provide scalable disk storage, xFS uses log-based network striping inspired by Zebra, but it dynamically clusters disks into stripe groups to allow the system to scale to a large number of storage servers. Finally, xFS replaces the server cache with cooperative caching that forwards data among client caches under control of the managers.

Preliminary test data indicates that the performance of xFS scales well with the addition of clients. However, the tests were microbenchmarks that provide significant parallelism and spread the load relatively evenly among xFS components. Real workloads will include hot spots that may limit the scalability of xFS or may require xFS to rely more heavily on its capacity to reconfigure responsibilities to avoid loaded machines. Also, a number of key features have not yet been implemented in the prototype. Additional work and stress testing will be needed before real users will want to trust their data to the system.

2.5 Harp [4]

Harp (Highly Available Reliable Persistent) is a replicated UNIX file system that is intended to be used within a network file service such as NFS or AFS. Harp provides highly available and reliable storage for files and guarantees that file operations are executed atomically in spite of concurrency and failure. All modifications to a file are reliably recorded at several server nodes; the number of nodes depends on how many failures the file is intended to survive.

Harp uses a novel variation of the primary copy replication technique. In this method, client calls are directed to a single primary server, which communicates with other backup servers and waits for them to respond before replying to the client. The system masks failures by performing a failover algorithm in which an inaccessible server is removed from service. When a primary performs an operation, it must inform enough backups to guarantee that the effects of that operation will survive all subsequent failovers.

Harp is one of the first implementations of a primary copy scheme that runs on conventional hardware. It has some novel features that allow it to perform well. Harp achieves good performance by recording the effects of modification operations in the log that resides in volatile memory; operations in the log are applied to the file system in the background. Essentially, it removes disk accesses from the critical path, replacing them with communication (from the primary to the backups), which is substantially faster if the servers are reasonably close together.

In using the log to record recent modifications, Harp is relying on a write-behind strategy, but the strategy is safe because log entries are not lost in failures. Each server is equipped with a small uninterruptible power supply (UPS) that allows it to run for a short while (e.g., a few minutes) after a power failure; the server uses this time to copy information in the log to disk. The combination of the volatile log and the UPS is one of the novel features of Harp. Information survives individual node failures because it exists in volatile memory at several nodes. It survives a power failure because of the UPS's.

As distributed systems become more and more prevalent, there is a growing dependence on file servers. However, existing file servers are unsatisfactory because they fail too often. Harp correct this problem. A file server can be switched to use Harp without file system users noticing the difference. Once Harp is in place, the file system will fail much less often than it did before, and even when it fails, it will usually do so more gracefully than an unreplicated system because operations are performed atomically (in the absence of a catastrophe).

In general, Harp provides high availability and good semantics with just a relatively small amount of additional hardware (the UPS's and the extra disks to hold extra file copies). Harp is especially well suited to Database applications.

2.6 Zebra [3]

Zebra has been designed to provide high performance and fault-tolerance. Zebra distributes file data across multiple file servers. Rather than striping each file separately, Zebra forms all the new data from each client into a single stream, which it then strips. Striping increases throughput and ensures that the loss of a single server does not affect the availability of the data. To do this, Zebra borrows from two innovations in the management of disk storage systems: Raid (Redundant Arrays of Inexpensive Disks) technology and log-structured file systems (LFS). RAID is a storage system architecture in which many small disks work together to provide increased performance and data availability. RAID technology allows Zebra to provide scalable file access performance while using parity instead of redundant copies to guard against server failures. The log-structured approach simplifies the parity implementation, reduces the impact of managing and storing parity, and allows clients to batch together small writes to improve the efficiency of writing to the server.

Zebra has several attractive properties:

Performance: Large files are read or written 4-5 times as fast as other network file systems, and small files are written 15-300 percent faster. In other file systems, only large files benefit from the striping.

Scalability: New disks or servers can be added incrementally to increase the system's bandwidth and capacity. This property is especially well suited for new styles of computing, such as multimedia and parallel computation, which require much greater throughput.

Cost-effective servers: Storage servers do not need to be high-performance machines or have special purpose hardware, since the performance of the system can be increased by adding more servers. Zebra transfers information to storage servers in large stripe fragments, and the servers do not interpret the contents of stripes; therefore, the server implementation is simple and efficient.

Availability: By combining ideas from RAID and LFS, Zebra can use simple mechanisms to manage parity for each stripe. The system can continue operation while one of the storage servers is unavailable and can reconstruct lost data in the event of a total failure of a server or disk.

On the down side, the current design of Zebra does little to provide security and protection for the files it stores. Malicious clients cannot overwrite existing files, but they can read files for which they should not have permission. The addition of security identifiers to file blocks and access control lists to the storage servers would greatly improve Zebra's security.

2.7 Frangipani [9]

Frangipani is a new scalable distributed file system that manages a collection of disks on multiple machines as a single shared pool of storage. It has a two-layer structure. The lower layer is Petal, which is a distributed storage system, providing incrementally scalable, highly available, easy-to-manage virtual disks. The upper layer is the Frangipani file system running on top of a single shared Petal virtual disk, using a distributed lock service to ensure coherence.

Petal consists of a collection of network connected servers that cooperatively manage a pool of physical disks. To Petal clients, this collection is a block-level storage system that provides large abstract containers called virtual disks. A virtual disk is a sparse 64-bit byte storage space globally accessible to all Petal clients. The Petal client interface is implemented as a kernel device driver. This makes Petal look like an ordinary local disk to higher layers of the system. The driver is responsible for contacting the correct Petal server and failing over to another when necessary.

The Frangipani file system is a client of Petal. The Frangipani file server module on each machine runs within the kernel. It accesses Petal virtual disks using the local Petal device driver. All the file servers access the same file system data structures on the shared Petal disk. The Frangipani servers have no need to communicate directly with one another, they communicate only with Petal and the lock service. This keeps file addition, deletion, and recovery simple.

Frangipani implements UNIX semantics and uniform global access to the system. Changes made to a file or directory on one machine are immediately visible on all others. The Frangipani cache is implemented in main memory, using the buffer pool of the kernel. Multiple-readers/single-writer locks are used to synchronize the current accesses to the shared on-disk data. The lock service is responsible for detecting conflicting lock requests: the current holder of the lock is asked to release or downgrade it to remove the conflict. Frangipani also uses *leases* to handle client crashes that occur when the client is holding a lock.

Frangipani is fault tolerant. It uses write-ahead redo logging of metadata; user data is not logged. Each Frangipani server has its own private log in Petal. When a Frangipani file server needs to make a metadata update, it first creates a record and appends it to its log in memory. These log records are periodically written out to Petal. If a Frangipani server crashes, the recovery daemon is implicitly given ownership of the failed server's log and locks. The daemon examines each record, carrying out each described update that is not already complete. After log processing is finished, the recovery daemon releases all its locks and frees the log. As long as the underlying Petal volume remains available, the system tolerates an unlimited number of Frangipani server failures.

Frangipani is scalable. To add a new server, the server needs only to be told which Petal virtual disk to use and where to find the lock service. The new server contacts the lock service to obtain a lease, determines which portion of the log space to use from the lease identifier, and goes into operation. The other servers will adapt to the presence of the new one automatically. Removing a Frangipani server is simple. It is adequate to simply shut the server off, but it is preferable for the server to flush all its dirty data and release its locks before halting. Addition and removal are both done without touching the other Frangipani servers in the system. The addition of a disk to Petal and the addition of a Petal server are completely handled by Petal, and is transparent to the Frangipani file system.

It is assumed that the Frangipani system is to be under a common administration domain. Frangipani servers trust one another, the Petal servers, and the lock service. Hence it can be viewed as a cluster file system. It is possible to export the Frangipani file system to untrusted machines outside an administration domain, but security is a problem. Alternatively, a client/server configuration can be used, with only the trusted Frangipani servers communicating with Petal and the lock service. Remote, untrusted clients talk to the Frangipani servers through a separate network and have no direct access to the Petal servers.

2.8 Secure File System [5,6]

SFS (secure file system) is a fairly new file system designed and developed by the MIT Laboratory for Computer Science. The goal is to design a secure network file system that can span the Internet. To meet the challenges of security and the diversity of the Internet, in designing SFS, security, extensibility, and portability were ranked over performance.

SFS splits overall security into two pieces: file system security and key management. SFS proper provides only file system security. In this way, it allows users to choose a key management policy according to their own security needs so that a more diverse range of uses can be supported. To achieve this, SFS adopted a modular design and self-certifying pathnames-filenames that inherently specify all information necessary to communicate securely with remote file servers: namely, a network address and a public key. By using separate programs for user authentication, one can add new user authentication protocols to SFS without modifying the actual file system software, and using self-certifying pathnames eliminates the need of separate key management to map file names to encryption keys like other file systems.

To prevent attackers from reading or modifying files without permission, and to allow programs to get the correct contents of whatever files they ask for, SFS cryptographically enforces all access control. Users cannot read, modify, delete, or otherwise tamper with files without possessing an appropriate secret key, unless anonymous access is explicitly permitted. SFS also cryptographically guarantees that the results of file system operations come from the appropriate server or private key owner. In the case that a server's private key is compromised, SFS provides two mechanisms, key revocation and HostID blocking, to prevent users from accessing its old (bad) self-certifying pathname, that may lead to a fake server, run by a malicious attacker.

Unlike any previous file system, SFS achieves its security without granting control of the namespace to any naming authority. Clients automatic mount remote file systems when a user references a self-certifying pathname for the first time. Therefore, new file servers are immediately accessible securely by all client machines without the need to reconfigure clients or register new servers in a centrally or hierarchically maintained database.

The SFS protocol includes two parts: a general-purpose read-write protocol that allows clients and read-write servers to communicate over a low-level secure channel, and a read-only protocol for heavily accessed public data with stringent security requirements. The read-only protocol eliminates the expensive key exchange at connection setup time and uses precomputed digital signatures to prove the contents of a file system to client machines. This allows the server to serve more clients at the same time.

SFS uses a lease-based cache mechanism where clients must hold a server-granted lease, with a explicit expiration time, before accessing any file (even if it is in the cache). Server generated unsolicited messages are used to synchronize read and write requests on shared files. This enables the use of a write-behind cache, because servers can directly ask write-lease holders for information before asking them to release the lease. It also enables better cache consistency and semantics than NFS provides and a longer attribute cache time. And because servers can inform reading clients of changes, readers don't have to check for changes as often. A large file or device for backing store is used to enable a huge cache, offsetting the cost of decryption.

SFS sacrificed performance for security and portability. Portability is achieved by running in user space and speaking an existing network file system protocol (NFS) to the local machine. Compared to NFS, which runs in the kernel, SFS pays a substantial cost for its user-level implementation and software encryption, which should have lead to worst performance. However, SFS has two important factors that mitigate these effects. First, by using an asynchronous RPC library which allows multiple outstanding requests, SFS enables overlapping of the latency of RPCs. Second, SFS has better attribute and access caching than NFS, which reduces the number of RPC calls that actually need to go over the network.

3 Conclusion

Distributed file systems are complex systems with many conflicting requirements. Since all the ideal requirements cannot be satisfied simultaneously, trade offs and compromises are required in the design. Design decisions must take into account the environments and applications for which the system will be used. As systems continue to grow in size, scalability and security have become more important. Many experimental systems have been developed in the last ten years to address these issues, but the issues remain problematic. Much of the future work in this field will probably be focused on these two key issues.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, R. Y. Wang, *Serverless Network File Systems*, Computer Science Division, University of California at Berkeley, 1995.

[2] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, G. Swart, *The Echo Distributed File System*, Digital Systems Research Center, Palo Alto, CA, 1993.

[3] J. H. Hartman and J. K. Ousterhout, *The Zebra Striped Network File System*, Sun Microsystems Laboratories, Inc., 1995.

[4] B. Liskov, S. Ghemaway, R. Gruber, P. Johnson, L. Shira, M. Williams, *Replication in the Harp File System*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, 1991.

[5] D. Mazieres, *Security and Decentralized Control in the SFS Global File System*, Master's thesis, Massachusetts Institute of Technology, 1997.

[6] D. Mazieres, M. Kaminsky, M. F. Kaashoek, E. Witchel, *Separating key management from file system security*, MIT Laboratory for Computer Science, 1999.

[7] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, R. Lyon, *Design and Implementation of the Sun Network Filesystem*, Sun Microsystems, Inc., Mountain View, CA, 1985.

[8] M. Satyanarayanan, *Scalable, Secure, and Highly Available Distributed File Access*, Carnegie Mellon University, 1990.

[9] C. A. Thekkath, T. Mann, E. K. Lee, *Frangipani: A Scalable Distributed File System*, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, 1997.