A Survey of Distributed File Systems

M. Satyanarayanan
Department of Computer Science
Carnegie Mellon University

February 1989

Abstract

Abstract

This paper is a survey of the current state of the art in the design and implementation of distributed file systems. It consists of four major parts: an overview of background material, case studies of a number of contemporary file systems, identification of key design techniques, and an examination of current research issues. The systems surveyed are Sun NFS, Apollo Domain, Andrew, IBM AIX DS, AT&T RFS, and Sprite. The coverage of background material includes a taxonomy of file system issues, a brief history of distributed file systems, and a summary of empirical research on file properties. A comprehensive bibliography forms an important of the paper.

The author was supported in the writing of this paper by the National Science Foundation (Contract No. CCR-8657907), Defense Advanced Research Projects Agency (Order No. 4976, Contract F33615-84-K-1520) and the IBM Corporation (Faculty Development Award). The views and conclusions in this document are those of the author and do not represent the official policies of the funding agencies or Carnegie Mellon University.

1. Introduction

The sharing of data in distributed systems is already common and will become pervasive as these systems grow in scale and importance. Each user in a distributed system is potentially a creator as well as a consumer of data. A user may wish to make his actions contingent upon information from a remote site, or may wish to update remote information. Sometimes the physical movement of a user may require his data to be accessible elsewhere. In both scenarios, ease of data sharing considerably enhances the value of a distributed system to its community of users. The challenge is to provide this functionality in a secure, reliable, efficient and usable manner that is independent of the size and complexity of the distributed system.

This paper is a survey of the current state of the art in the design of *distributed file systems,* the most widely used class of mechanisms for sharing data. It consists of four major parts: a brief survey of *background* material, *case studies* of a number of contemporary file systems, an identification of the *key design techniques* in use today, and an examination of *research issues* that are likely to challenge us in the next decade.

2. Background

We begin by examining the basic abstraction realized by file systems, and proceed to develop a taxonomy of issues in their design. Section 2.2 then traces the origin and development of distributed file systems until the middle of the current decade, when the systems described in Section 3 came into use. A sizeable body of empirical data on file usage properties is available to us today. Section 2.3 summarizes these observations and shows how they have influenced the design of distributed file systems.

2.1. Basic Issues

Permanent storage is a fundamental abstraction in computing. It consists of a named set of objects that come into existence by explicit creation, are immune to temporary failures of the system, and persist until explicitly destroyed. The naming structure, the characteristics of the objects, and the set of operations associated with them characterize a specific refinement of the basic abstraction. A file system is one such refinement.

From the perspective of file system design, computing models can be classified into four levels. The set of design issues at any level subsumes those at lower levels. Consequently, the implementation of a file system for a higher level will have to be more sophisticated than one that is adequate for a lower level.

At the lowest level, exemplified by IBM PC-DOS [25] and Apple Macintosh [4], one user at a single site performs computations via a single process. A file system for this model must address four key issues. These include the *naming structure* of the file system, the application *programming interface*, the *mapping* of the file system abstraction on to physical storage media, and the *integrity* of the file system across power, hardware, media and software failures.

The next level, exemplified by OS/2 [33], involves a single user computing with multiple processes at one site. *Concurrency control* is now an important consideration at the programming interface and in the implementation of the file system. The survey by Bernstein and Goodman [6] treats this issue in depth.

The classic timesharing model, where multiple users share data and resources, constitutes the third level of the taxonomy. Mechanisms to specifiy and enforce *security* now become important. Unix [62] is the archetype of a timesharing file system.

Distributed file systems constitute the highest level of the taxonomy. Here multiple users who are physically dispersed in a network of autonomous computers share in the use of a common file system. A useful way to view such a system is to think of it as a distributed implementation of the timesharing file system abstraction. The challenge is in realizing this abstraction in an efficient, secure and robust manner. In addition, the issues of *file* *location* and *availability* assume significance.

The simplest approach to file location is to embed location information in names. Examples of this approach can be found in the Newcastle Connection [9], Cedar [74], and Vax/VMS [15]. But the static binding of name to location makes it inconvenient to move files between sites. It also requires users to remember machine names, a difficult feat in a large distributed environment. A better approach is to use *location transparency*, where the name of a file is devoid of location information. An explicit file location mechanism dynamically maps file names to storage sites.

Availability is of special significance because the usage site of data can be different from its storage site. Hence failure modes are substantially more complex in a distributed environment. *Replication*, the basic technique used to achieve high availability, introduces complications of its own. Since multiple copies of a file are present, changes have to be propagated to all the replicas. Such propagation has to be done in a consistent and efficient manner.

2.2. Evolution

User-initiated file transfer was the earliest form of remote file access. Although inconvenient and limited in functionality, it served as an important mechanism for sharing data in the early days of distributed computing. IFS on the Alto personal computers [86] and the Datanet file repository on the Arpanet [40] exemplify this approach.

A major step in the evolution of distributed file systems was the recognition that access to remote file could be made to resemble access to local files. This property, called *network transparency*, implies that any operation that can be performed on a local file may also be performed on a remote file. The extent to which an actual implementation meets this ideal is an important measure of quality. The Newcastle Connection and Cocanet [64] are two early examples of systems that provided network transparency. In both cases the name of the remote site was a prefix of a remote file name.

The decade from 1975 to 1985 saw a profusion of experimental file systems. Svobodova examines many of these in her comparative survey [82]. Systems such as Felix [19], XDFS [43], Alpine [8], Swallow [81] and Amoeba [45, 46] explored the issues of atomic transactions and concurrency control on remote files. The Cambridge file system [7] and the CMU-CFS file system [1] examined how the naming structure of a distributed file system could be separated from its function as a permanent storage repository. The latter also addressed access control, caching, and transparent file migration on to archival media. Cedar [74] was the first file system to practically demonstrate the viability of caching entire files. Many of its design decisions were motivated by its intended application as a base for program development.

Locus [56, 88] was a landmark system in two important ways. First, it identified location transparency as an important design criterion. Second it proposed replication, along with a mechanism for detecting inconsistency, to achieve high availability. Locus also provided support for atomic transactions on files and generalized the notion of transparent remote access to all aspects of the operating system. *Weighted voting*, an alternative way of using replication for availability, was demonstrated in Violet [21, 22].

The rapid decline of CPU and memory costs motivated research on workstations without local disks or other permanent storage media. In such a system, a *disk server* exports a low-level interface that emulates local disk operations. Diskless operation has been successfully demonstrated in systems such as V [13] and RVD [26]. Lazowska et al [29] present an in-depth analysis of the performance of diskless workstations. Since diskless operation impacts autonomy, scalability, availability and security, it has to be viewed as a fundamental design constraint. It remains to be seen whether these considerations, together with continuing improvements in disk technology, will eventually outweigh the cost benefits of diskless operation.

Distributed file systems are in widespread use today. Section 3 describes the most prominent of these systems. Each major vendor now supports a distributed file system, and users often view it as an indispensable component. But the process of evolution is far from complete. As elaborated in Section 5, the next decade is likely to see significant improvements in the functionality, usability and performance of distributed file systems.

2.3. Empirical Observations

A substantial amount of empirical investigation in the classic scientific mold has been done on file systems. The results of this work have been used to guide high-level design as well as to determine values of system parameters. For example, data on file sizes has been used in the efficient mapping of files to disk storage blocks. Information on the frequency of different file operations and the degree of read- and write-sharing of files has influenced the design of caching algorithms. Type-specific file reference information has been useful in file placement and in the design of replication mechanisms.

Empirical work on file systems involves many practical difficulties. The instrumentation usually requires modifications to the operating system. In addition, it has to impact system performance minimally. The total volume of data generated is usually large, and needs to be stored and processed efficiently.

In addition to the difficulty of collecting data, there are two basic concerns about its interpretation. *Generality* is one of these concerns. How specific are the observations to the system being observed? Data of widespread applicability is obviously of most value. Independent investigations have been made of a variety of academic and research environments. The systems examined include IBM MVS [60, 79, 76], DEC PDP-10 [66, 67], and Unix [50, 17, 18, 39]. Although these studies differ in their details, there is substantial overlap in the set of issues they investigate. Further, their results do not exhibit any serious contradictions. We thus have confidence in our understanding of file system characteristics in academic and research environments. Unfortunately there is little publicly available information from other kinds of environments.

The second concern relates to the *interdependency* of design and empirical observations. Are the observed properties an artifact of existing system design or are they intrinsic? Little is known about the influence of system design on file properties, although the existence of such influence is undeniable. For example, in a design that uses whole-file transfer, there is substantial disincentive to the creation of very large files. In the long run this may affect the observed file size distribution. It is therefore important to revalidate our understanding of file properties as new systems are built and existing systems mature.

Studies of file systems fall into two broad categories. Early studies [60, 79, 76, 66] were based on *static* analysis, using one or more snapshots of a file system. The data from these studies is unweighted. Later studies [67, 50, 17, 18, 39] are based on *dynamic* analysis, using continuous monitoring of a file system. These data are weighted by frequency of file usage.

Although these studies have all been done on timesharing file systems their results are assumed to hold for distributed file systems. This is based on the premise that user behavior and programming environment characteristics are the primary factors influencing file properties. A further assumption is that neither of these factors changes significantly in moving to a distributed environment. No studies have yet been done to validate these assumptions.

The most consistent observation in all the studies is the skewing of file sizes toward the low end. In other words, most files are small, typically in the neighborhood of 10 kilobytes. Another common observation is that read operations on files are much more frequent than write operations. Random accessing of a file is rare. A typical application program sequentially reads an entire file into its address space and then performs nonsequential processing on the in-memory data. A related observation is that a file is usually read in its entirety once it has been opened.

Averaged over all the files in a system, data appears to be highly mutable. The *functional lifetime* of a file, defined as the time interval between the most recent read and the most recent write, is skewed toward the low end. In other words, data in files tends to be overwritten often. Although the mean functional lifetime is small, the tail of the distribution is long, indicating the existence of files with long-lived data.

Most files are read and written by one user. When users share a file, it is usually the case that only one of them modifies it. Fine granularity read-write sharing of files is rare. It is important to emphasize that these are observations derived from research or academic environments. An environment with large collaborative projects or one that makes extensive use of databases may show substantially greater write-sharing of data.

File references show substantial temporal locality of reference. If a file is referenced there is a high probability it will be referenced again in the near future. Over short periods of time the set of referenced files is a very small subset of all files.

The characteristics described above apply to the file population as a whole. If one were to focus on files of a specific type their properties may differ significantly. For example, system programs tend to be stable and rarely modified. Consequently the average functional lifetime of system programs is much larger than the average over all files. Temporary files on the other hand show substantially shorter lifetimes. More fine-grained classification of files is also possible, as demonstrated by some of the investigations mentioned earlier [66, 17, 18].

3. Case Studies

In this section we examine three distributed file systems that are widely used today, focusing on their design goals, their naming and location mechanisms, their use of replication and caching, and the support they provide for security and system management. Due to constraints of space we only provide sufficient detail to highlight the differences and similarities of their designs. In addition, we touch upon the noteworthy features of three other contemporary file systems in Section 3.4.

3.1. Sun Network File System

3.1.1. Design Considerations

Since its introduction in 1985, the Sun Microsystems *Network File System (NFS)* has been widely used in industry and academia. In addition to its technical innovations it has played a significant educational role in exposing a large number of users to the benefits of a distributed file system. Other vendors now support NFS and a significant fraction of the user community perceives it to be a *de facto* standard.

Portability and heterogeneity are two considerations that have played a dominant role in the design of NFS. Although the original file system model was based on Unix, NFS has been ported to to non-Unix operating systems such as PC-DOS. To facilitate portability, Sun makes a careful distinction between the *NFS protocol*, and a specific implementation of an NFS server or client. The NFS protocol defines an RPC interface that allows a server to export local files for remote access. The protocol does not specify how the server should implement this interface, nor does it mandate how the interface should be used by a client.

Design details such as caching, replication, naming, and consistency guarantees may vary considerably in different NFS implementations. In order to focus our discussion, we restrict our attention to the implementation of NFS provided by Sun for its workstations that run the SunOS flavor of Unix. Unless otherwise specified, the term ‘‘NFS’’ will refer to this implementation in the rest of this paper. The term ‘‘NFS protocol’’ will continue to refer to the generic interface specification.

SunOS defines a level of indirection in the kernel that allows file system operations to be intercepted and transparently routed to a variety of local and remote file systems. This interface, often referred to as the *vnode interface* after the primary data structure it exports, has been incorporated into many other versions of Unix.

With a view to simplifying crash recovery on servers, the NFS protocol is designed to be *stateless*. Consequently, servers are not required to maintain contextual information about their clients. Each RPC request from a client contains all the information needed to satisfy the request. To some degree functionality and Unix compatibility have been sacrificed to meet this goal. Locking, for instance, is not supported by the NFS protocol, since locks would constitute state information on a server. SunOS does, however, provide a separate lock server to perform this function.

Sun workstations are often configured without a local disk. The ability to operate such workstations without significant performance degradation is another goal of NFS. Early versions of Sun workstations used a separate remote-disk network protocol to support diskless operation. This protocol is no longer necessary since the kernel now transforms all its device operations into file operations.

A high-level overview of NFS is presented by Walsh et al [89]. Details of its design and implementation are given by Sandberg et al [65]. Kleiman [27] describes the vnode interface, while Rosen et al [63] comment on the portability of NFS.

3.1.2. Naming and Location

The NFS paradigm treats workstations as peers, with no fundamental distinction between clients and servers. A workstation may be a server, exporting some of its files. It may also be a client, accessing files on other workstations. But it is common practice for installations to be configured so that a small number of nodes run as dedicated servers, while the others run as clients.

NFS clients are usually configured so that each sees a Unix file name space with a private root. Using an extension of the Unix *mount* mechanism, subtrees exported by NFS servers are individually bound to nodes of the root file system. This binding usually occurs when Unix is initialized, and remains in effect until explicitly modified. Since each workstation is free to configure its own name space there is no guarantee that all workstations at an installation have a common view of shared files. But collaborating groups of users usually configure their workstations to have the same name space. Location transparency is thus obtained by convention, rather than being a basic architectural feature of NFS.

Since name-to-site bindings are static, NFS does not require a dynamic file location mechanism. Each client maintains a table mapping remote subtrees to servers. The addition of new servers or the movement of files between servers renders the table obsolete. There is no mechanism built into NFS to propagate information about such changes.

3.1.3. Caching and Replication

NFS clients cache individual pages of remote files and directories in their main memory. They also cache the results of pathname to vnode translations. Local disks, even if present, are not used for caching.

When a client caches any block of a file, it also caches a timestamp indicating when the file was last modified on the server. To validate cached blocks of a file, the client compares its cached timestamp with the timestamp on the server. If the server timestamp is more recent, the client invalidates all cached blocks of the file and refetches them on demand. A validation check is always performed when a file is opened, and when the server is contacted to satisfy a cache miss. After a check, cached blocks are assumed valid for a finite interval of time, specified by the client when a remote file system is mounted. The first reference to any block of the file after this interval forces a validation check.

If a cached page is modified, it is marked as dirty and scheduled to be flushed to the server. The actual flushing is performed by an asynchronous kernel activity and will occur after some unspecified delay. However, the kernel does provide a guarantee that all dirty pages of a file will be flushed to the server before a close operation on the file completes.

Directories are cached for reading in a manner similar to files. Modifications to directories, however, are performed directly on the server. When a file is opened, a cache validation check is also performed on its parent directory. Files and directories can have different revalidation intervals, typical values being 3 seconds for files and 30 seconds for directories.

NFS performs network data transfers in large block sizes, typically 8 Kbytes, to improve performance. Read-ahead is employed to improve sequential access performance. Files corresponding to executable binaries are fetched in their entirety if they are smaller than a certain threshold.

As originally specified, NFS did not support data replication. More recent versions of NFS support replication via a mechanism called *Automounter* [20, 11]. Automounter allows remote mount points to be specified using a set of servers rather than a single server. The first time a client traverses such a mount point a request is issued to each server, and the earliest to respond is chosen as the remote mount site. All further requests at the client that cross the mount point are directed to this server. Propagation of modifications to replicas has to be done manually. This replication mechanism is intended primarily for frequently-read and rarely-written files such as system binaries.

3.1.4. Security

NFS uses the underlying Unix file protection mechanism on servers for access checks. Each RPC request from a client conveys the identity of the user on whose behalf the request is being made. The server temporarily assumes this identity, and file accesses that occur while servicing the request are checked exactly as if the user had logged in directly to the server. The standard Unix protection mechanism using user, group and world *mode bits* is used to specify protection policies on individual files and directories.

In the early versions of NFS, mutual trust was assumed between all participating machines. The identity of a user was determined by a client machine and accepted without further validation by a server. The level of security of an NFS site was effectively that of the least secure system in the environment. To reduce vulnerability, requests made on behalf of *root* (the Unix superuser) on a workstation were treated by the server as if they had come from a non-existent user, *nobody*. Root thus received the lowest level of privileges for remote files.

More recent versions of NFS can be configured to provide a higher level of security. DES-based mutual authentication is used to validate the client and the server on each RPC request. Since file data in RPC packets is not encrypted, NFS is still vulnerable to unauthorized release and modification of information if the network is not physically secure.

The common DES key needed for mutual authentication is obtained from information stored in a publicly readable database. Stored in this database for each user and server is a pair of keys suitable for public key encryption. One key of the pair is stored in the clear, while the other is stored encrypted with the login password of the user. Any two entities registered in the database can deduce a unique DES key for mutual authentication. Taylor [83, 84] describes the details of this mechanism.

3.1.5. System Management

Sun provides two mechanisms to assist system managers. One of these, the *Yellow Pages (YP)*, is a mechanism for maintaining key-value pairs. The keys and values are application-specific and are not interpreted by YP. A number of Unix databases such as those mapping usernames to passwords, hostnames to network addresses, and network services to Internet port numbers are stored in YP. YP provides read-only replication, with one master and many slaves. Lookups may be performed at any replica. Updates are performed at the master, which is responsible for propagating the changes to the slaves. YP provides a shared repository for system information that changes relatively infrequently and that does not require simultaneous updates at all replication sites. YP is usually in use at NFS installations, although this is not mandatory.

The *Automounter*, mentioned in Section 3.1.3 in the context of read-only replication, is another mechanism for simplifying system management. It allows a client to lazy-evaluate NFS mount points, thus avoiding the need to mount all remote files of interest when the client is initialized. Automounter can be used in conjunction with YP to substantially simplify the administrative overheads of server reconfiguration.

3.2. Apollo Domain File System

3.2.1. Design Considerations

The DOMAIN system, built by Apollo Computers Inc., is a distributed workstation environment whose development began in the early 1980s. The goal of this system was to provide a usable and efficient computing base for a close-knit team of collaborating individuals. Athough scale was not a dominant design consideration, large Apollo installations now exist. The largest of these is located at the Apollo corporate headquarters and encompasses over 3500 nodes.

Apollo workstations range in hardware capability from small, diskless units to large configurations with disks and other peripherals. The underlying network technology is a proprietary 12 Mbit token ring [31]. Installations may choose to treat some of their nodes as dedicated servers that run only system software, and other nodes as clients performing user computations. Such a dichotomy is only a matter of convention. The DOMAIN software treats all nodes as peers.

DOMAIN provides support for the distribution of typed files via an *Object Storage System (OSS).* A system-wide *Single Level Store (SLS)* that provides a mapped virtual-memory interface to objects is built on top of the OSS. The DOMAIN distributed file system is layered on the SLS and presents a Unix-like file interface to application programs. A facility called the *Open Systems Toolkit* [59] uses the file typing mechanism of the OSS to create an extensible I/O system. Users can write non-kernel code to interpret I/O operations. When a file is opened its type is determined and the code implementing I/O operations on that type of object is dynamically loaded by the system.

Levine [34] presents the design and rationale of the DOMAIN file system. Its goals include location transparency, data consistency, a system-enforced uniform naming scheme, and a uniform mechanism for access control. Full functionality, good performance and ease of administration are other stated goals of DOMAIN. In addition to the survey by Levine are other papers on the file system [32, 30], the overall architecture [31], an object-oriented development tool for distributed applications [16], and the user registry [3, 54].

3.2.2. Naming and Location

Every object in the system is uniquely named by a 64-bit identifier called its *UID.* Each Apollo workstation is given a unique node identifier at the time of its manufacture. This identifier forms one component of the UID of every object created at that workstation. The time at which the object was created forms another component. Together these two components guarantee uniqueness of UIDs. Location-specific information in UIDs does not violate the goal of location transparency since its sole function is to guarantee uniqueness.

At any instant of time an object has a *home* node associated with it. The OSS maps objects to their homes by using a *hint server.* As its name implies, the hint server performs the mapping using a number of heuristics. It is updated in normal system operation by many diverse components of the DOMAIN software as they discover the location of objects. A heuristic that is frequently successful is to assume that objects created at the same node are likely to be located together.

A distributed *naming server* that maps string names to UIDs is built on top of the OSS. This server provides a hierarchical, Unix-like, location transparent name space for all files and directories in the system. Directories in DOMAIN are merely objects that map name components to UIDs. The network-wide root directory of the name space is implemented as a replicated distributed database with a server instance at the site of each replica. The naming facility is a good source of hints for the hint manager, since objects are often co-located with their parent directory.

3.2.3. Caching and Replication

The DOMAIN system transparently caches data and attributes of objects at the usage node. Mapped virtual-memory accesses via the SLS interface and file accesses via the file system interface are both translated into object references at the OSS level. The latter manages a cache of individual pages of objects using a write-back scheme with periodic flushing of data to the home of the objects.

A timestamp is associated with each object indicating the time at its home node when it was last modified. Every cached page of the object contains this timestamp. The consistency of locally cached data pages is verified by comparing their timestamps with the timestamp of the object at the home node. Invalid pages are merely discarded. In the course of references to the object, missing pages are obtained by demand paging across the network to the home node. Fetch-ahead (currently 8KB) is used to improve sequential access performance.

Cache management in DOMAIN is integrated with its concurrency control mechanisms. Each node runs a *lock manager* that synchronizes accesses to all objects which have their home at that node. Two modes of locking are supported. One mode allows multiple distributed readers or a single writer to access the object. The other mode allows access to multiple readers and writers co-located at a single node. Lock managers do not queue requests. If a lock for an object cannot be granted immediately, the requesting node must periodically retry its request.

Cache validation is performed when an object is locked. When a write-lock on an object is released, an implicit *purify* operation is performed. This operation atomically flushes updated pages of an object to its home node. Application software is responsible for ensuring that objects are locked before being mapped into virtual memory or opened for file access. It is also responsible for releasing locks when appropriate.

DOMAIN does not support read-only or read-write replication of data. An object can have only one home at any instant of time. But replicated services such as a replicated user registry and a replicated naming service are supported by DOMAIN.

3.2.4. Security

Security in DOMAIN is predicated on the physical integrity of Apollo workstations and on the trustworthiness of the kernels on them. Since the network is also assumed to be secure, communications on it are sent in the clear. The network component of the kernel at each node uses a special field in the header of every packet to indicate whether the originator of the packet was a user-level program, or the kernel itself. This prevents user-level programs from masquerading as trusted system software.

A distributed user registry stores each user’s login password in encrypted form, as in Unix. When a user logs in on a node, the local kernel encrypts the password typed by the user, fetches his login entry from the registry, and validates the user by comparing the two encrypted passwords. Each instance of a logged-in user is associated with a unique identifier, called a *PPON*, that identifies the user, the project and organization he belongs to, and the node at which this login instance occurred. The PPON is used on all access checks on behalf of this logged-in instance of the user. Nodes cache recently-used registry information to enhance availability.

The user registry, called *RGY*, is a replicated database with one master site and multiple read-only slaves for availability. Each replica contains the entries for all users in the system. Updates are made at the master site which then propagates them asynchronously to the slave sites. Direct authentication to the master, using a Needham-Schroeder authentication handshake, is required before an update can be performed.

Protection policies are specified by access lists on objects. An access list entry maps a PPON to a set of rights. Components of PPONs can be wildcarded. If multiple entries are applicable in a given access check, the most specific matching entry overrides the others. Checking of access has been done at the home node of objects in some releases of DOMAIN, and at the usage node in other releases. These are logically equivalent, since the kernels trust each other.

3.2.5. System Management

Concern for ease of administration has been an important influence on the design of the DOMAIN user registry described in the previous section. Its design allows multiple mutually suspicious groups to use a single registry for system management information. Each group can have a distinct system administrator who is the only person who can manipulate entries pertaining to the group. Decentralized administration and specification of usage policies is effectively supported by this mechanism. The registry also supports heterogeneity, initially in the form of a client interface for Sun workstations.

An interactive tool, *edrgy*, provides a structured interface to the registry. It possesses substantial semantic knowledge of the contents of the registry and guides administrators. Edrgy detects and notifies administrators of potentially serious side effects of their actions.

3.3. Andrew File System

3.3.1. Design Considerations

Andrew is a distributed workstation environment that has been under development at Carnegie Mellon University since 1983. It combines the rich user interface characteristic of personal computing with the data sharing simplicity of timesharing. The primary data-sharing mechanism is a distributed file system spanning all the workstations. Using a set of trusted servers, collectively called *Vice*, the Andrew file system presents a homogeneous, location-transparent file name space to workstations. Clients and servers both run the 4.3 BSD version of Unix. It is a relatively heavyweight operation to configure a machine as an Andrew server. This is in contrast to systems such as Sun NFS, where it is trivial for any machine to export a subset of its local file system.

Scalability is the dominant design consideration in Andrew. Many design decisions in Andrew are influenced by its anticipated final size of 5000 to 10000 nodes, Careful design is necessary to provide good performance at large scale and to facilitate system administration. Scale also renders security a serious concern, since it has to be enforced rather than left to the good will of the user community.

The goals and directions of the Andrew project have been described by Morris et al [44]. The file system has been discussed extensively in papers focusing on architecture [70], performance [24], security [68], and the influence of scale [69]. The Andrew file system has undergone one complete revision, and a second revision is under way [77].

3.3.2. Naming and Location

The file name space on an Andrew workstation is partitioned into a *shared* and a *local* name space. The shared name space is location transparent and is identical on all workstations. The local name space is unique to each workstation and is relatively small. It only contains temporary files or files needed for workstation initialization. Users see a consistent image of their data when they move from one workstation to another, since their files are in the shared name space.

Both name spaces are hierarchically structured, as in Unix. The shared name space is partitioned into disjoint subtrees, and each such subtree is assigned to a single server, called its *custodian*. This assignment is relatively static, although reassignment for operational reasons is possible. Internally, Andrew uses 96-bit *file identifiers* to uniquely identify files. These identifiers are not visible to application programs.

Each server contains a copy of a fully replicated *location database* that maps files to custodians. This database is relatively small because custodianship is on subtrees, rather than on individual files. Temporary inaccuracies in the database are harmless, since forwarding information is left behind when data is moved from one server to another.

3.3.3. Caching and Replication

Files in the shared name space are cached on demand on the local disks of workstations. A cache manager, called *Venus*, runs on each workstation. When a file is opened, Venus checks the cache for the presence of a valid copy. If such a copy exists, the open request is treated as a local file open. Otherwise an up-to-date copy is fetched from the custodian. Read and write operations on an open file are directed to the cached copy. No network traffic is generated by such requests. If a cached file is modified, it is copied back to the custodian when the file closed.

Cache consistency is maintained by a mechanism called *callback.* When a file is cached from a server, the latter makes a note of this fact and promises to inform the client if the file is updated by someone else. Callbacks may be broken at will by the client or the server. The use of callback, rather than checking with the custodian on each open, substantially reduces client-server interactions. The latter mechanism was used in the first version of Andrew. Andrew caches large chunks of files, to reduce client-server interactions and to exploit bulk data transfer protocols. Earlier versions of Andrew cached entire files.

A mechanism orthogonal to caching is *read-only replication* of data that is frequently read but rarely modified. This is done to enhance availability and to evenly distribute server load. Subtrees that contain such data may have read-only replicas at multiple servers. But there is only one read-write replica and all updates are directed to it. Propagation of changes to the read-only replicas is done by an explicit operational procedure.

Concurrency control is provided in Andrew by emulation of the Unix *flock* system call. Lock and unlock operations on a file are performed directly on its custodian. If a client does not release a lock within 30 minutes, it is timed out by the server.

3.3.4. Security

The design of Andrew pays serious attention to security, while ensuring that the mechanisms for enforcing it do not inhibit legitimate use of the system [68]. Security is predicated on the integrity of a small number of Vice servers. These servers are physically secure, are accessible only to trusted operators, and run only trusted system software. Neither the network, nor workstations are trusted by Vice. Authentication and secure transmission mechanisms based on end-to-end encryption are used to provide secure access to servers from workstations.

It is still the responsibility of a user to ensure that he is not being compromised by malicious software on his workstation. To protect himself against Trojan horse attacks, a concerned user has to maintain the physical integrity of his workstation and to deny remote access to it via the network.

The protection domain in Andrew is composed of *users*, corresponding to human users, and *groups*, which are sets of users and other groups. Membership in a group is inherited, and a user accumulates the privileges of all the groups he belongs to directly and indirectly. Inheritance of membership simplifies the maintenance and administration of the protection domain. Membership in a special group called ‘‘System:Administrators’’ endows administrative privileges, including unrestricted access to any file in the system.

Andrew uses an *access list* mechanism for protection. The total rights specified for a user are the union of all the rights collectively specified for him and for all the groups of which he is a direct or indirect member. An access list can specify *negative rights*. An entry in a negative rights list indicates *denial* of the specified rights, with denial overriding possession in case of conflict. Negative rights are intended primarily a means of rapidly and selectively revoking access to critical files and directories.

For conceptual simplicity, Vice associates access lists with directories rather than files. The access list applies to all files in the directory, thus giving them uniform protection status. In addition, the three owner bits of the Unix *file mode* are used indicate readability, writability or executability. In Andrew, these bits indicate what can be done to the file rather than who can do it.

For reasons of compatibility, Andrew will replace its original authentication system with the Kerberos authentication system of Project Athena [78]. The two resemble each other closely in architecture, although they differ substantially in the details. Both use a two-step authentication scheme. When a user logs in to a workstation, his password is used to establish a secure communication channel to an *authentication server*. A pair of *authentication tokens* (in the case of Andrew), or an *authentication ticket* (in the case of Kerberos) are obtained from the authentication server and saved for future use. These are used, as needed, to establish secure RPC connections on behalf of the user to individual file servers. The authentication server has to run on a trusted machine in both systems. For robustness, there are multiple instances of this server. Only one server, the master, accepts updates. The others are slaves and respond only to queries. Changes are propagated to slaves by the master.

3.3.5. System Management

Operability is a major concern in Andrew on account of its scale. The system has to be easy for a small staff to run and administer. Regular operational procedures have to be performed in a manner that cause minimal disruption of service to users.

The operational mechanisms of Andrew are built around a data structuring primitive called a *volume* [75]. A volume is a collection of files forming a partial subtree of the Vice name space and having the same custodian. Volumes are glued together at *mount points* to form the complete name space.

There is usually one volume per user, as well as a number of volumes containing system software. Volume sizes are usually small enough to allow many volumes per disk partition on a server. Disk storage quotas are applied on a per-volume basis. Operations such as moving a volume from one server to another can be performed while the volume is still online. A read-only replica of a volume can be created by a *clone* operation. Such replicas can be used to improve availability and performance. Read-only volumes can also be used to implement an orderly release process for system software.

Volumes also form the basis of the backup and restoration mechanism. To backup a volume a frozen snapshot of its files is created by cloning. An asynchronous mechanism then transfers this clone to a staging machine from where it is dumped to tape. To handle the common case of accidental deletion by users, the cloned backup volume of each user’s files is made available as a read-only subtree in Vice.

Andrew has been extended to allow decentralized operation. A cooperating group of *cells* adhering to a standardized set of protocols and naming conventions [92] can jointly provide the image of a single file name space. Cross-cell authentication and translation of user identities in different administrative domains are key issues that have been addressed in this mechanism.

3.4. Other Contemporary Systems

Each of the three systems described in this section is important either because it is widely used or because it occupies a unique position in the space of distributed file system designs. In the interests of brevity, we only present condensed descriptions of the most distinctive aspects of these systems.

3.4.1. IBM AIX Distributed Services

As its name imples, *AIX Distributed Services (DS)* is a collection of distributed system services developed by IBM for its workstations running the AIX Operating System. AIX is a derivative of the System V version of Unix. The primary component of DS is a distributed file system whose design goals include strict emulation of Unix semantics, ability to efficiently support databases, and ease of administering a wide range of DS installation configurations.

A DS client can access remote files via an extension of the Unix mount mechanism. DS allows individual files and directories to be mounted, in contrast to distributed file systems that allow mounts only at the granularity of an entire subtree. A server does not have to explicitly export or advertise the files it wishes to share. Rather, all files are assumed to be remotely accessible, subject to access checks. Most file system operations behave identically on local and remote files. Two significant exceptions are the inability to access remote devices, and the inability to map remote files into the address space of a process. The latter restriction will be removed in a future release of DS.

DS uses client main memory as a write-through cache of individual pages of files. Clients notify servers of each open of a file for reading or writing. The behavior of the caching mechanism depends on whether it is in *read-only mode* (one or more clients reading and no clients writing), *async mode* (one client reading and writing), or *full-sync mode* (multiple clients writing). In read-only mode caching is enabled at all clients. In async mode caching is enabled only at the writer, with all other sites directing their read requests to the server. Client caching is disabled in full sync mode. Cache consistency is maintained by a mechanism reminiscent of the Andrew callback mechanism. Since a server is aware of all remote opens of its files, it can keep track of all clients that have opened a file since the last time it was modified. Before accepting the next open for modification, it notifies this list of clients and they invalidate all pages of the file that are in their caches.

DS uses virtual circuit communication based on the SNA LU6.2 protocol. Future versions of DS will also be able to run on the TCP/IP protocol. Node to node DES mutual authentication is provided as part of the LU6.2 implementation. Users and groups have 32-bit network-wide ids. DS translates these network ids into machine-specific Unix-compatible 16-bit ids. The Kerberos authentication mechanism will be supported as an option in the future.

Sauer et al [72] and Levitt [35] describe the design of DS. Sauer [71] presents a detailed description of the fine-granularity mount mechanism. Sauer et al [73] discuss the rationale for maintaining client state on servers for some aspects of DS while avoiding state for other aspects.

3.4.2. AT&T Remote File Sharing

Remote File Sharing (RFS) is a distributed file system developed by AT&T for its System V version of Unix, and is derived from an earlier implementation for Unix Edition 8 [90]. The most distinctive feature of RFS is its precise emulation of local Unix semantics for remote files. An operation on a remote file is indistinguishable from the corresponding operation on a local file. This aspect of RFS extends to areas such as concurrency control, write-sharing semantics and the ability to access and control remote devices.

RFS uses the client-server model with virtual circuit communication based on Unix System V Streams [49] to provide easy portability across a variety of transport protocols. A server advertises each subtree it wishes to export, using a network-wide symbolic name for the root of the subtree. Clients explicitly import remote subtrees using symbolic names. A name server performs the translation of symbolic names to server addresses.

Accuracy of Unix system call emulation is achieved by executing all remote file system calls on the server. A client merely intercepts and forwards these calls to the server. The exact execution environment of the client is recreated on the server for the duration of a call, using information passed by the client in its request.

The initial version of RFS used no caching. It has since been extended to provide caching in client main memory, retaining the exact emulation of Unix semantics. Caching is used only for simple files, not for directories or devices. The cache is write-through, with consistency being checked on opens. With a single writer and multiple readers, caching is disabled at the readers. Caching is reenabled when the writer closes the file, or when the time interval since the last modification by the writer exceeds a predefined threshold. All caching in the system is disabled when there are multiple writers.

RFS clients and servers trust each other. Protection on files and directories is specified exactly as in Unix. A mechanism to map user and group identities allows files to be shared across administrative domains. RFS also provides a mechanism to restrict the privileges of remote users at a coarse granularity.

The rationale, architecture and implementation of RFS are described by Rifkin et al [61]. Bach et al [5] describe how RFS was extended to incorporate caching. Chartock [12] shows how RFS was made to coexist with Sun NFS, using the vnode interface. A comparison of Sun NFS and AT&T RFS is presented by Hatch et al [23].

3.4.3. Sprite Network File System

Sprite is an operating system for networked uniprocessor and multiprocessor workstations, designed at the University of California at Berkeley. The goals of Sprite include efficient use of large main memories, support for multiprocessor workstations, efficient network communication, and diskless operation. Besides a distributed file system, Sprite provides other distributed system facilities such as process migration.

Most workstations in a Sprite network are diskless. Although the design of Sprite does not make a rigid distinction between clients and servers, a few machines with disks are usually dedicated as file servers. These servers jointly present a location-transparent Unix file system interface to clients.

Clients do not have to explicitly import files from individual servers. Each server can respond to location queries, using *remote links* embedded in the file system at each server. Remote links are effectively pointers to files at other servers. Each client maintains a local *prefix table*, which maps pathname prefixes to servers. Substantial performance improvement is achieved by using the cached information in the prefix table for locating files.

Sprite is intended for use by a collection of collaborating users who are either incapable of subverting the kernels on workstations, or who trust each other. Consequently Sprite kernels trust each other, and communication between them is neither authenticated nor encrypted.

Exact emulation of Unix file system semantics is an important goal of Sprite. Whenever a client opens or closes a file for reading or writing, it notifies the server that stores the file. A Sprite client usually caches pages of a file, validating these pages each time the file is opened. Caching is disabled when multiple clients have a file open, and one or more of these clients have it open for writing. Once caching is disabled, it is reenabled only after all clients concurrently using the file have closed it. This strategy enables Sprite to provide consistency at the granularity of individual read and write operations.

Sprite provides location-transparent remote access to devices as well as files. To provide good performance under a wide variety of workloads, physical memory on a Sprite workstation is dynamically partitioned between the virtual memory subsystem and the file cache. Sprite uses ordinary file in the shared name space for paging . This simplifies process migration, since the backing files are visible at all other Sprite workstations in the environment.

Ousterhout et al [51] provide an overview of Sprite. Welch and Ousterhout [91] describe the prefix mechanism used for file location. A detailed performance analysis of caching in Sprite is presented by Nelson et al [48].

4. Mechanisms and Techniques

The purpose of this section is to highlight and discuss certain mechanisms that have been found to be of significant value in the design of distributed file systems. These mechanisms are of general applicability, except for mount points, which are Unix-specific. But even mount points are widely used since the majority of distributed file systems at the present time are based on Unix. The dominance of the Unix file system model is indeed remarkable.

4.1. Mount Points

The *mount* mechanism in Unix enables the glueing together of file name spaces to provide applications with a single, seamless, hierarchically structured, name space. On startup, the Unix file name space consists of a single *root file system.* Individual mount commands may then be issued to bind the root of an external file system to an internal or leaf node of the local name space. A mount on an internal node hides the original subtree beneath that node. To simplify the implementation, Unix imposes certain restrictions such as the inability to place hard links across mount points.

Mount was originally conceived as a mechanism to allow self-contained file systems on removable storage media to be added to or removed without reinitializing Unix. When performing a name lookup, the kernel uses an internal data structure called the *mount table* to direct its search to the appropriate storage device. A single lookup may span many devices if multiple mounts are in effect. In a distributed file system, the mount mechanism provides a natural hook on which to hang a remote subtree. There are two fundamentally different ways to use the mechanism, with numerous variants of each.

The simpler approach is used by systems such as NFS, where each client individually mounts subtrees from servers. There is no centralized management of mount information. Servers are unaware of where the subtrees exported by them have been mounted. Although this approach is easier to implement, it has the disadvantage that the shared name space is not guaranteed to be identical at all clients. Further, movement of files from one server to another requires each client to unmount and remount the affected subtree. In practice, systems that use this approach have usually had to provide auxiliary mechanisms (such as the Yellow Pages and Automounter in NFS) to automate and centralize mounts.

The alternative approach is to embed mount information in the data stored in the file servers. Andrew, for example, uses mount points embedded in volumes. Sprite uses remote links for a similar purpose. Using this approach, it is trivial to ensure that all clients see precisely the same shared file name space. Further, operational tasks such as moving files from one server to another only involve updating the mount information on the servers.

4.2. Caching at Clients

The caching of data at clients is undoubtedly the architectural feature that contributes most to performance in a distributed file system. Every distributed file system in serious use today uses some form of caching. Even AT&T's RFS, which initially avoided caching in the interests of strict Unix emulation, now uses it.

Caching exploits locality of reference. There is a high probability that file data will be reused soon after its first use. By obtaining a local copy of the data a client can avoid many further interactions with the server. Meta data, such as directories, protection and file status information, and file location information also exhibit locality of reference and are thus good candidates for caching.

A key issue in caching is the size of the cached units of data. Most distributed file systems cache individual pages of files. Early versions of Andrew cached entire files. Although this simplifies cache managment and offers simpler failure semantics it does suffer from the inability to access files that are larger than the client's cache. More recent versions of Andrew cache large portions (typically 64KB) of files. The unit of caching is closely related to the use of bulk transfer protocols, as discussed in Section 4.4.

In most systems clients maintain the cache in their main memory. Andrew is an exception in that it caches on its local disk, with a further level of caching in main memory. Besides providing larger cache sizes, disk caching preserves cache contents across system reboots.

The validation of cache contents can be done in two fundamentally different ways. One approach, used by most systems, is for the client to contact the server for validation. The alternative approach, used in Andrew and DS, is to have the server notify clients when cached data is about to be rendered stale. Although more complex to implement, the latter approach can produce substantial reductions in client-server traffic.

Existing systems use a wide spectrum of approaches in propagating modifications from client to server. In async mode, DS propagates changes to the server only when the file is explicitly flushed. Andrew propagates changes when a file is closed after writing. Sprite delays propagation until dirty cache pages have to be reclaimed or for a maximum of 30 seconds. Deferred propagation improves performance since data is often overwritten, but increases the possibility of server data being stale due a client crash.

References within a file also exhibit spatial locality. If a page of a file is read, there is substantial likelihood that succeeding pages will also be read. This property is exploited in many systems by using *read-ahead* of file data. The client can overlap the processing of one page with the fetching of the next page or set of pages from the server.

4.3. Hints

In the context of distributed systems, a *hint* [28] is a piece of information that can substantially improve performance if correct but has no semantically negative consequence if erroneous. For maximum performance benefit a hint should nearly always be correct. Terry [85] discusses the use of hints in detail and provides many examples of how they may be used in distributed systems.

By caching hints one can obtain substantial performance benefits without incurring the cost of maintaining cache consistency. Only information that is self-validating upon use is amenable to this strategy. One cannot, for instance, treat file data as a hint because the use of a cached copy of the data will not reveal whether it is current or stale.

Hints are most often used for file location information in distributed file systems. Sprite, for instance, caches mappings of pathname prefixes to servers. Similarly, Andrew caches individual entries from the volume location database. In these systems a client will use cached location information until a server rejects a request because it no longer stores the file referred to in the request. The client then obtains the new location of the file, and caches this information as a fresh hint. A more elaborate location scheme, incorporating a hint manager, is used by Apollo Domain.

4.4. Transferring Data in Bulk

Network communication overheads typically account for a major portion of the latency in a distributed file system. Although the transit time of small amounts of data across a local area network is insignificant, the delays caused by protocol processing can be substantial. Transferring data in bulk reduces this overhead at both the source and sink of the data. At the source, multiple packets are formatted and transmitted with one context switch. At the sink, an acknowledgement is avoided for each packet. Some bulk transfer protocols also make better use of the disks at the source and sink. Multiple blocks of data may often be obtained at the source with a single seek. Similarly, packets can be buffered and written *en masse* to the disk at the sink. In effect, the use of bulk transfer amortizes fixed protocol overheads over many consecutive pages of a file.

Bulk transfer protocols depend on spatial locality of reference within files for effectiveness. There is a very high probability that succeeding pages of a file will soon be referenced at the client if an earlier page is referenced. As mentioned in Section 2.3 there is substantial empirical evidence to indicate that files are read in their entirety once they are opened.

The degree to which bulk transfer is exploited varies from system to system. Andrew, for instance, is critically dependent on it for good performance. Early versions of the system transferred entire files, and the current version transfers files in 64KB chunks. Systems such as NFS and Sprite exploit bulk transfer by using very large packet sizes, typically 8KB. The latter systems depend on the link level protocol to fragment and reassemble smaller packets at the media access level. Bulk transfer protocols will increase in importance as distributed file systems spread across networks of wider geographic area and thus have greater inherent latency.

4.5. Encryption

Encryption is an indispensable building block for enforcing security in a distributed system. Voydock and Kent [87] classify threats to security as actions that cause unauthorized *release* of information, unauthorized *modification* of information, or unauthorized *denial of resources.* Encryption is primarily of value in preventing unauthorized release and modification of information. Because it is a national standard, DES [42] is the most commonly used form of private-key encryption.

The seminal work of Needham and Schroeder [47] on the use of encryption for authentication is the basis of all current security mechanisms in distributed file systems. At the heart of these mechanisms is a handshake protocol in which each party challenges the other to prove its identity. Possesion of a secret encryption key, known only to a legitimate client and server, is assumed to be *prima facie* evidence of authenticity. Thus two communicating entities that are mutually suspicious at the beginning end up confident of each other's identity, without ever transmitting their shared secret key in the clear.

This basic scheme is used in two distinct ways in current systems. The difference lies in the way user passwords are stored and used on servers. In the scheme used by Kerberos and Andrew, an authentication server that is physically secure maintains a list of user passwords in the clear. In contrast, the public key scheme used by Sun NFS maintains a publicly readable database of authentication keys that are encrypted with user passwords. The latter approach has the attractive characteristic that physical security of the authentication server is unnecessary.

Encryption is usually implemented end-to-end, at the RPC level. DS, in contrast, uses node-to-node encryption. In some systems, such as Andrew, encryption can be used to protect the data and headers of all packets exchanged after authentication. Other systems, such as Sun NFS, do not provide this capability.

A difficult nontechnical problem is justifying the cost of encryption hardware to management and users. Unlike extra memory, processor speed, or graphics capability, encryption devices do not provide tangible benefits to users. The importance of security is often perceived only after it is too late. At present, encryption hardware is viewed as an expensive frill. Hopefully, the emerging awareness that encryption is indispensable for security will make rapid, cheap encryption a universally available capability.

5. Current Issues

Distributed file systems continue to be the subject of considerable activity and innovation in industry and academia. Work is being done in the areas of *availability,* further *scaling*, support for *heterogeneity*, and *database access.* We briefly consider each of these issues in the following sections.

5.1. Availability

As reliance on distributed file systems increases, the problem of availability becomes acute. Today a single server crash or network partition can seriously inconvenience many users in a large distributed file system. There is a growing need for distributed file systems that are highly resilient to failures.

Availability is the focus of the Coda file system, currently being built at Carnegie Mellon University. Coda's goal is to provide the highest degree of availability in the face of all realistic failures, without significant loss of usability, performance or security. Two orthogonal mechanisms, *replication* and *disconnected operation*, are used to achieve this goal. Many key architectural features of Coda, such as the use of caching with callback, whole-file transfer, RPC-based authentication and encryption, and aggregation of data into volumes are inherited from Andrew.

Consistency, availability and performance tend to be mutually contradictory goals in a distributed system. Coda's strategy is to provide the highest availability at the best performance. It considers inconsistency tolerable if it is rare, occurs only under conditions of failure, is always detected, and is allowed to propagate as little as possible.

The relative infrequency of simultaneous write-sharing of files by multiple users makes this a viable strategy.

High availability is also a key concern of the Echo file system being built at the System Research Center of Digital Equipment Corporation. This design also uses replication, but its strategy differs substantially from that of Coda. At any time exactly one of the servers with a replica of a file is its *primary* site. Clients interact only with the primary site, which assumes the responsibility of propagating changes to the other replication sites. In case of partition, file updates are allowed only in the partition containing a majority of the replication sites. When the primary site is down, a new primary site is elected.

Other recent experimental efforts in this area include RNFS at Cornell University [41] and Gemini at the University of California at San Diego [14, 52, 53].

5.2. Scalability

Certain problems induced by scale have been exposed by the extensive use of large distributed file systems. One problem is the need for *decentralization.* The ability to delegate administrative responsibility along lines that parallel institutional boundaries is critical for smooth and efficient operation. The ideal model of decentralization is one in which users perceive the system as monolithic even when their accesses span many administrative domains. In practice, of course, most accesses from a client are likely to be directed to a server in the same administrative domain. As mentioned in Section 3 the Apollo and Andrew file systems have been extended to support decentralized administration.

Another aspect of scaling is the extension of the distributed file system paradigm over wide geographic areas. Virtually all distributed file systems today are designed with local area networks in mind. It is an open question whether such designs can be extended over networks with longer latencies and greater chances of network congestion. An effort is currently under way to extend Andrew to operate over a wide-area network. With its emphasis on caching and minimization of client-server interactions, the design of Andrew seems quite appropriate for such extension.

A basic question that arises at very large scale is whether a single hierarchically-organized name space is indeed the most appropriate model for sharing data. This paradigm, originally invented for timesharing systems of tens or hundreds of users, has been successfully extended to distributed file systems of a thousand or so nodes. Will it be the best model when there are two orders of magnitude or more nodes? Pathnames become longer and it becomes increasingly difficult to search for files whose name is not precisely known. The Quicksilver file system [10], currently under development at the IBM Almaden Research Center, addresses this issue. Its approach is to provide mechanisms for a user to customize his name space. Since the customization is location-transparent the user retains his context when he moves to any other node in the system. A similar approach has been proposed in the Plan 9 system at Bell Labs [57].

Network topology is becoming an increasingly important aspect of distributed systems. Large networks often have complex topologies, caused by a variety of factors. Electrical considerations limit the lengths of individual network segments and the density of machines on them. Maintenance and fault isolation are simplified if a network is decomposable. Administrative functions such as the assignment of unique host addresses can be decentralized if a network can be partitioned. Although distributed file systems mask underlying network complexity, performance inhomogeneities cannot be hidden. Routers, which introduce load-dependent transmission delays, are a common source of performance inhomogeneity. Uneven loading of subnets is another cause. The interaction between network topology and distributed system performance is still poorly understood. A preliminary investigation of these issues has been reported in the context of Andrew [37].

5.3. Heterogeneity

As a distributed system evolves it tends to grow more diverse. A variety of factors contribute to increased heterogeneity. First, a distributed system becomes an increasingly valuable resource as it grows in size and stores larger amounts of shared data. There is then considerable incentive and pressure to allow users outside the scope of the system to participate in the use of its resources. A second source of heterogeneity is the improvement in performance and decrease in cost of hardware over time. This makes it likely that the most effective hardware configurations will change over the period of growth of the system. Functional specialization is a third reason for heterogeneity. Certain combinations of hardware and software may be more appropriate than others for specific applications.

The distributed file system community has gained some experience with heterogeneity. Pinkerton et al describe an experimental file system at Washington [55] that focuses on heterogeneity. *TOPS* [80] is a product offered by Sun Microsystems that allows personal computers running the PC-DOS and Macintosh operating systems to share files. *PC-NFS*, also from Sun, allows PC-DOS applications to access files on an NFS server. A surrogate server mechanism in Andrew, called *PCServer* [58], enables PC-DOS applications to access files in Vice.

Coping with heterogeneity is inherently difficult because of the presence of multiple computational environments, each with its own notions of file naming and functionality. Since few general principles are applicable, the idiosyncrasies of each new system have to be accomodated by ad hoc mechanisms. Unfortunately heterogeneity cannot be ignored since it is likely to be more widespread in the future.

5.4. Database Access

As mentioned in the introductory section of this paper, a file system is a refinement of the permanent storage abstraction. A *database* is an alternative refinement and differs from a file system in two important ways. One difference is the *storage model* presented to application programs and users. A file system views the data in a file as an uninterpreted byte sequence. In contrast, a database encapsulates substantial information about the types and logical relationships of data items stored in it. It can ensure that constraints on values are satisfied and can enforce protection policies at fine granularity. The second fundamental distinction is in the area of naming. A file system provides access to a file *by name* whereas a database allows *associative access.* Data items can be accessed and modified in a database based on user-specified predicates.

Neither the difference in storage model nor in naming makes it *a priori* more difficult to build distributed databases than file systems. However the circumstances that lead to the use of a database are often precisely those that make distribution of data difficult. The use of databases is most common in applications where data has to be concurrently shared for reading and writing by a large number of users. These applications usually demand strict consistency of data as well as atomicity of groups of operations. Although the total quantity of data in the database may be large, the granularity of access and update is usually quite fine. It is this combination of application characteristics that makes the implementation of distributed databases substantially harder than the implementation of distributed file systems.

Distributing a database is particularly difficult at large scale. In its most general form the problem seems hopelessly difficult. A database is conceptually a focal point for enforcing concurrency control and atomicity properties. If the control structures to enforce these properties are physically distributed, the resulting network protocols have to be substantially more complex. The feasibility of fully distributing data and control at small scale has been demonstrated by systems such as R* [36]. But extension to larger distributed systems is not trivial.

A less ambitious approach attempts to provide *distributed access* to data on a single large database server. Although the data itself is located at a single site, transparent access to this data is possible from many sites. In this model the database requirements of a large distributed system are met by a small number of powerful database servers each exporting a standardized network interface. An example of such a system, *Scylla*, has been demonstrated at

Carnegie Mellon University by integrating an off-the-shelf relational database system, Informix, with an RPC package. Similar approaches have recently been announced by Microsoft [2], Oracle [38] and other vendors.

6. Conclusion

Since the earliest days of distributed computing, file systems have been the most important and widely-used form of shared permanent storage. The continuing interest in distributed file systems bears testimony to the robustness of this model of data sharing. We now understand how to implement distributed file systems that span a few hundred to a few thousand nodes. But scaling beyond that will be a formidable challenge. As elaborated in the preceding section, availability, heterogeneity and support for databases will also be key issues. Security will continue to be a serious concern and may, in fact, turn out to be the bane of large distributed systems. Regardless of the specific technical direction taken by distributed file systems in the next decade, there is little doubt that it will be an area of considerable ferment in industry and academia.

Acknowledgements

Brent Callaghan, Michael Kazar, James Kistler, Paul Leach, Paul Levine, John Ousterhout, Charlie Sauer, Ellen Siegel, Chris Silveri, Carl Smith, Warren Smith, Alfred Spector and Peter Weinberger each made valuable comments on this paper and contributed to its technical accuracy and readability.

References

[1] Accetta, M., Robertson, G., Satyanarayanan, M., Thompson, M.
The Design of a Network-based Central File System.
Technical Report CMU-CS-80-134, Department of Computer Science, Carnegie Mellon University, 1980.

[2] Adler, M.
Developing SQL Server Database Applications Through DB-Library.
Microsoft Systems Journal 3(6), November, 1988.

[3] *Heterogeneous Registry Client Side Release Document*
Apollo Computer Inc., 1988.
Part No. 015182-A0.

[4] Apple Computer, Inc.
Inside Macintosh, Volume II.
Addison Wesley, 1985.

[5] Bach, M.J., Luppi, M.W., Melamed, A.S., Yueh, K.
A Remote-File Cache for RFS.
In *Summer Usenix Conference Proceedings, Phoenix*. 1987.

[6] Bernstein, P.A., Goodman, N.
Concurrency Control in Distributed Database Systems.
Computing Surveys 13(2), June, 1981.

[7] Birrell, A.D., Needham, R.M.
A Universal File Server.
IEEE Transactions on Software Engineering SE-6(5), September, 1980.

[8] Brown, M.R., Kolling, K.N., Taft, E.A.
The Alpine File System.
ACM Transactions on Computer Systems 3(4), November, 1985.

[9] Brownbridge, D.R., Marshall, L.F., Randell, B.
The Newcastle Connection.
Software Practice and Experience 12, 1982.

[10] Cabrera, L.F., Wyllie, J.
QuickSilver Distributed File Services: An Architecture for Horizontal Growth.
In *Proceedings of the 2nd IEEE Conference on Computer Workstations, Santa Clara*. March, 1988.
Also available as Tech Report RJ5578, April 1987, Computer Science Department, IBM Almaden Research Center.

[11] Callaghan, B., Lyon, T.
The Automounter.
In *Winter Usenix Conference Proceedings, San Diego*. 1989.

[12] Chartock, H.
RFS in SunOS.
In *Summer Usenix Conference Proceedings, Phoenix*. 1987.

[13] Cheriton, D.R., Zwaenepoel, W.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proceedings of the 9th ACM Symposium on Operating System Principles, Bretton Woods*. October, 1983.

[14] Davcev, D., Burkhard, W.A.
Consistency and Recovery Control for Replicated Files.
In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island*. December, 1985.

[15] *VMS System Software Handbook*
Digital Equipment Corporation, 1985.

[16] Dineen, T.H., Leach, P.J., Mishkin, N.W., Pato, J.N., Wyant, G.L.
The Network Computing Architecture and System: An Environment for Developing Distributed Applications.
In *Summer Usenix Conference Proceedings, Phoenix*. 1987.

[17] Floyd, R.
Short-Term File Reference Patterns in a Unix Environment.
Technical Report TR-177, Department of Computer Science, University of Rochester, 1986.

[18] Floyd, R.
Directory Reference Patterns in a Unix Environment.
Technical Report TR-179, Department of Computer Science, University of Rochester, 1986.

[19] Fridrich, M., Older, W.
The FELIX File Server.
In *Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar*. December, 1981.

[20] Garlick, L., Lyon, R., Delzompo, L., Callaghan, B.
The Open Network Computing Environment.
SunTechnology 1(2), Spring, 1988.

[21] Gifford, D.K.
Violet, an Experimental Decentralized System.
Technical Report CSL-79-12, Xerox Corporation, Palo Alto Research Center, September, 1979.

[22] Gifford, D.K.
Weighted Voting for Replicated Data.
Technical Report CSL-79-14, Xerox Corporation, Palo Alto Research Center, September, 1979.

[23] Hatch, M.J., Katz, M., Rees, J..
AT&T's RFS and Sun's RFS.
Unix/World II(11), December, 1985.

[24] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6(1), February, 1988.

[25] *Disk Operating System, Version 2.1*
IBM Corporation, 1983.
1502343.

[26] IBM Corporation.
The Remote Virtual Disk Subsystem
1987.
In *Academic Operating System, Volume III.*.

[27] Kleiman, S.R.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
In *Summer Usenix Conference Proceedings, Atlanta*. 1986.

[28] Lampson, B.W.
Hints for Computer System Designers.
In *Proceedings of the 9th ACM Symposium on Operating System Principles, Bretton Woods*. October, 1983.

[29] Lazowska, E.D., Zahorjan, J., Cheriton, D.R., Zwaenepoel, W.
File Access Performance of Diskless Workstations.
ACM Transactions on Computer Systems 4(3), 1986.

[30] Leach, P.J., Stumpf, B.L., Hamilton, J.A., Levine, P.H.
UIDs as Internal Names in a Distributed File System.
In *Proceedings of the Symposium on Principles of Distributed Computing, Ottawa*. August, 1982.

[31] Leach, P.J., Levine, P.H., Douros, B.P., Hamilton, J.A., Nelson, D.L., Stumpf, B.L.
The Architecture of an Integrated Local Network.
IEEE Journal on Selected Areas in Communications 1(5), November, 1983.

[32] Leach, P.J., Levine, P.H., Hamilton, J.A., Stumpf, B.L.
The File System of an Integrated Local Network.
In *Proceedings of the ACM Computer Science Conference, New Orleans*. March, 1985.

[33] Letwin, G.
Inside OS/2.
Microsoft Press, 1988.

[34] Levine, P.H.
The Apollo DOMAIN Distributed File System.
In Paker, Y., Banatre, J-P, Bozyigit, M. (editors), *NATO ASI Series: Theory and Practice of Distributed Operating Systems*. Springer-Verlag, 1987.

[35] Levitt, J.
The IBM RT Gets Connected.
Byte 12(12), 1987.

[36] Lindsay, B.G., Haas, L.M., Mohan, C., Wilms, P.F., Yost, R.A.
Computation and Communication in R*: A Distributed Database Manager.
ACM Transactions on Computer Systems 2(1), February, 1984.

[37] Lorence, M., Satyanarayanan, M.
IPWatch: A Tool for Monitoring Network Locality.
In *Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Mallorca*. September, 1988.
Also available as Tech. Rept.CMU-CS-88-176, Department of Computer Science, Carnegie Mellon University.

[38] Mace, S.
Oracle, Fox, XDB Detail Strategies for Database Servers.
Infoworld November 21, 1988.

[39] Majumdar, S., Bunt, R.B.
Measurement and Analysis of Locality Phases in File Referencing Behaviour.
In *Proceedings of Performance '86 and ACM Sigmetrics 1986, Raleigh*. May, 1986.

[40] Marill, T., Stern, D.
The Datacomputer -- A Network Data Utility.
In *Proceedings of AFIPS National Computer Conference*. May, 1975.

[41] Marzullo, K., Schmuck, F.
Supplying High Availability with a Standard Network File System.
Technical Report 87-888, Department of Computer Science, Cornell University, December, 1987.

[42] Meyer, C.H., Matyas, S.M.
Cryptography: A New Dimension in Computer Data Security.
John Wiley & Sons, 1982.

[43] Mitchell, J.G., Dion, J.
A Comparison of Two Network-based File Servers.
Communications of the ACM 25(4), April, 1982.

[44] Morris, J. H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S., Smith, F.D.
Andrew: A Distributed Personal Computing Environment.
Communications of the ACM 29(3), March, 1986.

[45] Mullender, S.J., Tanenbaum, A.S.
A Distributed File Service based on Optimistic Concurrency Control.
In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island*. December, 1985.

[46] Mullender, S.J., Tanenbaum, A.S.
The Design of a Capability-based Operating System.
The Computer Journal 29(4), March, 1986.

[47] Needham, R.M, Schroeder, M.D.
Using Encryption for Authentication in Large Networks of Computers.
Communications of the ACM 21(12), December, 1978.

[48] Nelson, M.N., Welch, B.B., Ousterhout, J.K.
Caching in the Sprite Network File System.
ACM Transactions on Computer Systems 6(1), February, 1988.

[49] Olander, D.J., McGrath, G.J., Israel, R.K.
A Framework for Networking in System V.
In *Summer Usenix Conference Proceedings, Atlanta*. 1986.

[50] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J.
A Trace-Driven Analysis of the Unix 4.2 BSD File System.
In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island*. December, 1985.

[51] Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., Welch, B.B.
The Sprite Network Operating System.
Computer 21(2), February, 1988.

[52] Paris, J-F.
Voting with Witnesses: A Consistency Scheme for Replicated Files.
Technical Report, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, San Diego, 1986.

[53] Paris, J-F.
Voting with a Variable Number of Copies.
Technical Report, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, San Diego, 1986.

[54] Pato, J.N., Martin, E., Davis, B.
A User Account Registration System for a Large (Heterogeneous) UNIX Network.
In *Winter Usenix Conference Proceedings, Dallas*. 1988.

[55] Pinkerton, C.B., Lazowska, E.D., Notkin, D., Zahorjan, J.
A Heterogeneous Remote File System.
Technical Report 88-08-08, Department of Computer Science, University of Washington, August, 1988.

[56] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., Thiel, G.
LOCUS, A Network Transparent, High Reliability Distributed System.
In *Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar*. December, 1981.

[57] Presotto, D.L.
Plan 9 from Bell Labs -- The Network.
In *European Unix User Group Conference Proceedings, London*. April, 1988.

[58] Raper, L.K.
The CMU PC Server Project.
Technical Report CMU-ITC-051, Information Technology Center, Carnegie Mellon University, February, 1986.

[59] Rees, J., Levine, P.H., Mishkin, N., Leach, P.J.
An Extensible I/O System.
In *Summer Usenix Conference Proceedings, Atlanta*. 1986.

[60] Revelle, R.
An Empirical Study of File Reference Patterns.
Technical Report RJ 1557, IBM Research Division, April, 1975.

[61] Rifkin, A.P., Forbes, M.P., Hamilton, R.L., Sabrio, M., Shah, S., Yueh, K.
RFS Architectural Overview.
In *Summer Usenix Conference Proceedings, Atlanta*. 1986.

[62] Ritchie, D.M., Thompson, K.
The Unix Time Sharing System.
Communications of the ACM 17(7), July, 1974.

[63] Rosen, M.B., Wilde, M.J., Fraser-Campbell, B.
NFS Portability.
In *Summer Usenix Conference Proceedings, Atlanta*. 1986.

[64] Rowe, L.A., Birman, K.P.
A Local Network Based on the Unix Operating System.
IEEE Transactions on Software Engineering SE-8(2), March, 1982.

[65] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.
Design and Implementation of the Sun Network Filesystem.
In *Summer Usenix Conference Proceedings, Portland*. 1985.

[66] Satyanarayanan, M.
A Study of File Sizes and Functional Lifetimes.
In *Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar*. December, 1981.

[67] Satyanarayanan, M.
A Synthetic Driver for File System Simulations.
In *Proceedings of the International Conference on Modelling Techniques and Tools for Performance Analysis, Paris*. May, 1984.

[68] Satyanarayanan, M.
Integrating Security in a Large Distributed Environment.
Technical Report CMU-CS-87-179, Department of Computer Science, Carnegie Mellon University, 1987.

[69] Satyanarayanan, M.
On the Influence of Scale in a Distributed System.
In *Proceedings of the 10th International Conference on Software Engineering, Singapore*. April, 1988.

[70] Satyanarayanan, M., Howard, J.H., Nichols, D.N., Sidebotham, R.N., Spector, A.Z., West, M.J.
The ITC Distributed File System: Principles and Design.
In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island*. December, 1985.

[71] Sauer, C.H.
Presenting a Single System Image with Fine Granularity Mounts.
login: 13(4), July/August 1988, 1988.

[72] Sauer, C.H., Johnson, D.W., Loucks, L.K., Shaheen-Gouda, A.A., Smith, T.A.
RT PC Distributed Services Overview.
Operating Systems Review 21(3), July, 1987.

[73] Sauer, C.H., Johnson, D.W., Loucks, L.K., Shaheen-Gouda, A.A., Smith, T.A.
Statelessness and Statefulness in Distributed Services.
In *Proceedings of UniForum 1988, Dallas*. February, 1988.

[74] Schroeder, M.D., Gifford, D.K., Needham, R.M.
A Caching File System for a Programmer's Workstation.
In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island*. December, 1985.

[75] Sidebotham, R.N.
Volumes: The Andrew File System Data Structuring Primitive.
In *European Unix User Group Conference Proceedings*. August, 1986.
Also available as Technical Report CMU-ITC-053, Information Technology Center, Carnegie Mellon University.

[76] Smith, A.J.
Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms.
IEEE Transactions on Software Engineering 7(4), July, 1981.

[77] Spector, A.Z., Kazar, M.L.
Wide Area File Service and the AFS Experimental System.
Unix Review 7(3), March, 1989.

[78] Steiner, J.G., Neuman, C., Schiller, J.I.
Kerberos: An Authentication Service for Open Network Systems.
In *Winter Usenix Conference Proceedings, Dallas*. 1988.

[79] Stritter, E.P.
File Migration.
PhD thesis, Stanford University, 1977.

[80] Stroud, G.
Introduction to TOPS.
SunTechnology 1(2), Spring, 1988.

[81] Svobodova, L.
A Reliable Object-oriented Data Repository for a Distributed Computer System.
In *Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar*. December, 1981.

[82] Svobodova, L.
File Servers for Network-Based Distributed Systems.
Computing Surveys 16(4), December, 1984.

[83] Taylor, B.
Secure Networking in the Sun Environment.
In *Summer Usenix Conference Proceedings, Atlanta*. 1986.

[84] Taylor, B.
A Framework for Network Security.
SunTechnology 1(2), Spring, 1988.

[85] Terry, D.B.
Caching Hints in Distributed Systems.
IEEE Transactions in Software Engineering SE-13(1), January, 1987.

[86] Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., Boggs, D.R.
Alto: A Personal Computer.
In Siewiorek, D.P., Bell., C.G., Newell, A. (editors), *Computer structures: Principles and Examples*. McGraw-Hill Book Company, 1981.

[87] Voydock, V.L., Kent, S.T.
Security Mechanisms in High-Level Network Protocols.
Computing Surveys 15(2), June, 1983.

[88] Walker, B., Popek, G., English, R., Kline, C., Thiel, G.
The LOCUS Distributed Operating System.
In *Proceedings of the 9th ACM Symposium on Operating System Principles, Bretton Woods*. October, 1983.

[89] Walsh, D., Lyon, B., Sager, G., Chang, J.M., Goldberg, D., Kleiman, S., Lyon, T., Sandberg, R., Weiss, P.
Overview of the Sun Network Filesystem.
In *Winter Usenix Conference Proceedings, Dallas*. 1985.

[90] Weinberger, P.J.
The Version 8 Network File System.
In *Summer Usenix Conference Proceedings, Salt Lake City*. 1984.

[91] Welch, B., Ousterhout, J.
Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System.
In *Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge*. May, 1986.

[92] Zayas, E.R., Everhart, C.F.
Design and Specification of the Cellular Andrew Environment.
Technical Report CMU-ITC-070, Information Technology Center, Carnegie Mellon University, June, 1988.

Table of Contents