# Scalable, Secure, and Highly Available Distributed File Access

**Mahadev Satyanarayanan**
**Carnegie Mellon University**

**Andrew and Coda are distributed Unix file systems that embody many of the recent advances in solving the problem of data sharing in large, physically dispersed workstation environments.**

For the users of a distributed system to collaborate effectively, the ability to share data easily is vital. Over the last decade, distributed file systems based on the Unix model have been the subject of growing attention. They are now widely considered an effective means of sharing data in academic and research environments. This article presents a summary and historical perspective of work done by my colleagues, students, and I in designing and implementing such systems at Carnegie Mellon University.

This work began in 1983 in the context of Andrew, a joint project of CMU and IBM to develop a state-of-the-art computing facility for education and research at CMU. The project envisioned a dramatic increase in computing power made possible by the widespread deployment of powerful personal workstations. Our charter was to develop a mechanism that would enable the users of these workstations to collaborate and share data effectively. We decided to build a distributed file system for this purpose because it would provide the right balance between functionality and complexity for our usage environment.

It was clear from the outset that our distributed file system had to possess two critical attributes: It had to scale well, so that the system could grow to its anticipated final size of over 5,000 workstations. It also had to be secure, so that users could be confident of the privacy of their data. Neither of these attributes is likely to be present in a design by accident, nor can it be added as an afterthought. Rather, each attribute must be treated as a fundamental constraint and given careful attention during the design and implementation of a system.

Our design has evolved over time, resulting in three distinct versions of the Andrew file system, called AFS-1, AFS-2, and AFS-3. In this article "Andrew file system" or "Andrew" will be used as a collective term referring to all three versions.

As our user community became more dependent on Andrew, the availability of data in it became more important. Today, a single failure in Andrew can seriously inconvenience many users for significant periods. To address this problem, we began the design of an experimental file system called Coda in 1987. Intended for the same computing environment as Andrew, Coda retains Andrew's scalability and security characteristics while providing much higher availability.

## The Andrew architecture

The Andrew computing paradigm is a synthesis of the best features of personal computing and timesharing. It combines the flexible and visually rich user interface available in personal computing with the ease of information exchange typical of


0018-9162/90/0500-0009$01.00 © 1990 IEEE

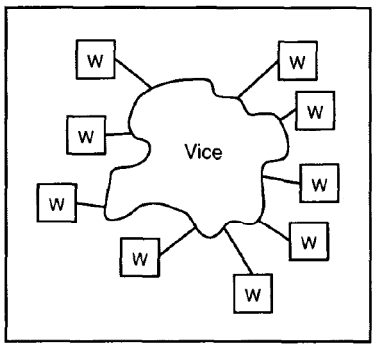


**Figure 1. A high-level view of the Andrew architecture. The structure labeled "Vice" is a collection of trusted file servers and untrusted networks. The nodes labeled "W" are private or public workstations, or timesharing systems. Software in each such node makes the shared files in Vice appear as an integral part of that node's file system.**

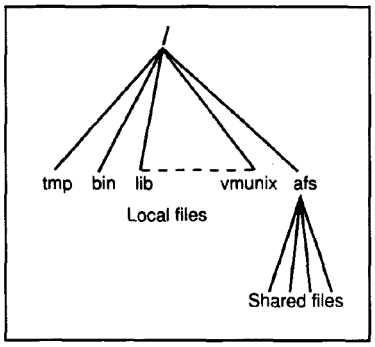


**Figure 2. File system view at a workstation: how the shared files in Vice appear to a user. The subtree under the directory labeled "afs" is identical at all workstations. The other directories are local to each workstation. Symbolic links can be used to make local directories correspond to directories in Vice.**

timesharing. A conceptual view of this model is shown in Figure 1.

The large, amoeba-like structure in the middle, called Vice, is the information-sharing backbone of the system. Although represented as a single entity, it actually consists of a collection of dedicated file servers and a complex local area network. User computing cycles are provided by workstations running the Unix operating system.

Data sharing in Andrew is supported by a distributed file system that appears as a single large subtree of the local file system on each workstation. The only files outside the shared subtree are temporary files and files essential for workstation initialization. A process called Venus, running on each workstation, mediates shared file access. Venus finds files in Vice, caches them locally, and performs emulation of Unix file system semantics. 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. Processes on two different workstations can read and write files in this subtree just as if they were running on a single timesharing system. Figure 2 depicts the file system view seen by a workstation user.

Our experience with the Andrew architecture over the past six years has been positive. It is simple and easily understood by naive users, and it permits efficient implementation. It also offers a number of benefits that are particularly valuable on a large scale:

- *Data sharing is simplified.* A workstation with a small disk can potentially access any file in Andrew by name. Since the file system is location transparent, users do not have to remember the machines on which files are currently located or where files were created. System administrators can move files from one server to another without inconveniencing users, who are completely unaware of such a move.
- *User mobility is supported.* A user can walk to any workstation in the system and access any file in the shared name space. A user's workstation is personal only in the sense that he owns it.
- *System administration is easier.* Operations staff can focus on the relatively small number of servers, ignoring the more numerous and physically dispersed clients. Adding a new workstation involves merely connecting it to the network and assigning it an address.
- *Better security is possible.* The servers in Vice are physically secure and run trusted system software. No user programs are executed on servers. Encryption-based authentication and transmission are used to enforce the security of server-workstation communication. Although individuals may tamper with the hardware and software on their workstations, their malicious actions cannot affect users at other workstations.
- *Client autonomy is improved.* Workstations can be turned off or physically relocated at any time without inconveniencing other users. Backup is needed only on the servers, since workstation disks are used merely as caches.

## Scalability in Andrew

A *scalable* distributed system is one that can easily cope with the addition of users and sites, its growth involving minimal expense, performance degradation, and administrative complexity. We have achieved these goals in Andrew by reducing static bindings to a bare minimum and by maximizing the number of active clients that can be supported by a server. The following sections describe the evolution of our design strategies for scalability in Andrew.

**AFS-1.** AFS-1 was a prototype with the primary functions of validating the Andrew file system architecture and providing rapid feedback on key design decisions. Each server contained a local file system mirroring the structure of the shared file system. Vice file status information, such as access lists, was stored in shadow directories. If a file was not on a server, the search for its name would end in a stub directory that identified the server containing that file. Since server processes could not share memory, their only means of sharing data structures was via the local file system.

Clients cached pathname prefix information and used it to direct file requests to appropriate servers. The Vice-Venus interface named files by their full pathnames. There was no notion of a low-level name, such as the *inode* in Unix.

Venus used a pessimistic approach to maintaining cache coherence. All cached copies of files were considered suspect. Before using a cached file, Venus would contact Vice to verify that it had the latest version. Each open of a file thus resulted in at least one interaction with a server, even if the file was already in the cache and up to date.

For the most part, we were pleased with AFS-1. Almost every application was able to use Vice files without recompilation or relinking. There were minor areas of incompatibility with standard Unix semantics, but these were never serious enough to discourage users.

## Design principles from Andrew and Coda

The design choices of Andrew and Coda were guided by a few simple principles. They were not specified a priori, but emerged in the course of our work. We share these principles and examples of their application in the hope that they will be useful to designers of other large-scale distributed systems. The principles should not be applied dogmatically but should be used to help crystallize thinking during the design process.

• **Workstations have the cycles to burn.** Whenever there is a choice between performing an operation on a workstation and performing it on a central resource, it is preferable to pick the workstation. This enhances the scalability of the design because it lessens the need to increase central resources as workstations are added.

The only functions performed by servers in Andrew and Coda are those critical to security, integrity, or location of data. Further, there is very little interserver traffic. Pathname translation is done on clients rather than on servers in AFS-2, AFS-3, and Coda. The parallel update protocol in Coda depends on the client to directly update all AVSG members, rather than updating one of them and letting it relay the update.

• **Cache whenever possible.** Scalability, user mobility, and site autonomy motivate this principle. Caching reduces contention on centralized resources and transparently makes data available wherever it is being used.

AFS-1 cached files and location information. AFS-2 also cached directories, as do AFS-3 and Coda. Caching is the basis of disconnected operation in Coda.

• **Exploit file usage properties.** Knowledge of the nature of file accesses in real systems allows better design choices to be made. Files can often be grouped into a small number of easily identifiable classes that reflect their access and modification patterns. These class-specific properties provide an opportunity for independent optimization and, hence, improved performance.

Almost one-third of the file references in a typical Unix system are to temporary files. Since such files are seldom shared, Andrew and Coda make them part of the local name space. The executable files of system programs are often read but rarely written. AFS-2, AFS-3, and Coda therefore support read-only replication of these files to improve performance and availability. Coda's use of an optimistic replication strategy is based on the premise that sequential write sharing of user files is rare.

• **Minimize systemwide knowledge and change.** In a large distributed system, it is difficult to be aware at all times of the entire state of the system. It is also difficult to update distributed or replicated data structures consistently. The scalability of a design is enhanced if it rarely requires global information to be monitored or atomically updated.

Workstations in Andrew and Coda monitor only the status of servers from which they have cached data. They do not require any knowledge of the rest of the system. File location information on Andrew and Coda servers changes relatively rarely. Caching by Venus, rather than file location changes in Vice, is used to deal with movement of users.

Coda integrates server replication (a relatively heavyweight mechanism) with caching to improve availability without losing scalability. Knowledge of a caching site is confined to servers with callbacks for the caching site. Coda does not depend on knowledge of systemwide topology, nor does it incorporate any algorithms requiring systemwide election or commitment.

Another instance of the application of this principle is the use of negative rights. Andrew provides rapid revocation by modifications of an access list at a single site rather than by systemwide change of a replicated protection database.

• **Trust the fewest possible entities.** A system whose security depends on the integrity of the fewest possible entities is more likely to remain secure as it grows.

Rather than trusting thousands of workstations, security in Andrew and Coda is predicated on the integrity of the much smaller number of Vice servers. The administrators of Vice need only ensure the physical security of these servers and the software they run. Responsibility for workstation integrity is delegated to the owner of each workstation. Andrew and Coda rely on end-to-end encryption rather than physical link security.

• **Batch if possible.** Grouping operations (and hence scalability) can improve throughput, although often at the cost of latency.

The transfer of files in large chunks in AFS-3 and in their entirety in AFS-1, AFS-2, and Coda is an instance of the application of this principle. More efficient network protocols can be used when data is transferred en masse rather than as individual pages. In Coda the second phase of the update protocol is deferred and batched. Latency is not increased in this case because control can be returned to application programs before the completion of the second phase.

AFS-1 was in use for about a year, from late 1984 to late 1985. At its peak usage, there were about 100 workstations and six servers. Performance was usually acceptable to about 20 active users per server. But sometimes a few intense users caused performance to degrade intolerably. The system turned out to be difficult to operate and maintain, especially because it provided few tools to help system administrators. The embedding of file location information in stub directories made it hard to move user files between servers.

**AFS-2.** The design of AFS-2 was based on our experience with AFS-1 as well as on extensive performance analysis.[1] We retained the strategy of workstations caching entire files from a collection of dedicated autonomous servers. But we made many changes in the realization of this architecture, especially in cache management, name resolution, communication, and server process structure.

A fundamental change in AFS-2 was the manner in which cache coherence was maintained. Instead of checking with a

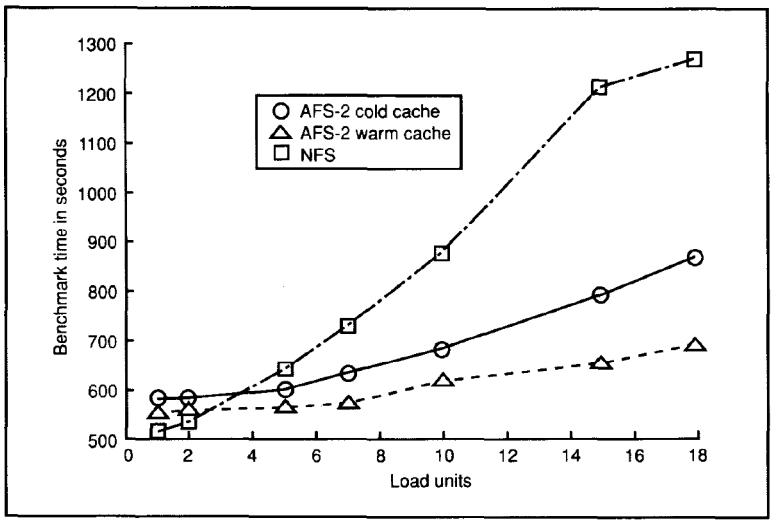


**Figure 3. AFS-2 versus Sun NFS performance under load on identical client, server, and network hardware. A load unit consists of one client workstation running an instance of the Andrew benchmark. (Full details of the benchmark and experimental configuration can be found in Howard et al.,[1] from which this graph is adapted.) As the graph clearly indicates, the performance of AFS-2, even with a cold cache, degrades much more slowly than that of NFS.**

server on each open, Venus now assumed that cache entries were valid unless otherwise notified. When a workstation cached a file or directory, the server promised to notify that workstation before allowing a modification by any other workstation. This promise, known as a *callback*, resulted in a considerable reduction in cache validation traffic.

Callback made it feasible for clients to cache directories and to translate pathnames locally. Without callbacks, the lookup of every component of a pathname would have generated a cache validation request. For reasons of integrity, directory modifications were made directly on servers, as in AFS-1. Each Vice file or directory in AFS-2 was identified by a unique fixed-length file identifier. Location information was contained in a slowly changing volume location database replicated on each server.

AFS-2 used a single process to service all clients of a server, thus reducing the context switching and paging overheads observed in AFS-1. A nonpreemptive lightweight process mechanism supported concurrency and provided a convenient programming abstraction on servers and clients. The RPC (remote procedure call) mechanism in AFS-2, which was integrated with the lightweight process mechanism, supported a very large number of active clients and used an optimized bulk-transfer protocol for file transfer.

Besides the changes we made for performance, we also eliminated AFS-1's inflexible mapping of Vice files to server disk storage. This change was the basis of a number of mechanisms that improved system operability. Vice data in AFS-2 was organized in terms of a data-structuring primitive called *a volume*, a collection of files forming a partial subtree of the Vice name space. Volumes were glued together at mount points to form the complete name space. Venus transparently recognized and crossed mount points during name resolution.

Volumes were usually small enough to allow many volumes per server disk partition. Volumes formed the basis of disk quotas. Each system user was typically assigned a volume, and each volume was assigned a quota. Easily moved between servers by system administrators, a volume could be used (even for update) while it was being moved.

Read-only replication of volumes made it possible to provide increased availability for frequently read but rarely updated files, such as system programs. The backup and restoration mechanism in AFS-2 also made use of volume primitives. To back up a volume, a read-only clone was first made. Then, an asynchronous mechanism transferred this frozen snapshot to a staging machine from which it was dumped to tape. To handle the common case of accidental deletion by users, the cloned backup volume of each user's files was made available as a read-only subtree of that user's home directory. Thus, users themselves could restore files within 24 hours by means of normal file operations.

AFS-2 was in use at CMU from late 1985 until mid-1989. Our experience confirmed that it was indeed an efficient and convenient system to use at large scale. Controlled experiments established that it performed better under load than other contemporary file systems.[1,2] Figure 3 presents the results of one such experiment.

**AFS-3.** In 1988, work began on a new version of the Andrew file system called AFS-3. (For ease of exposition, all changes made after the AFS-2 release described in Howard et al.[1] are described here as pertaining to AFS-3. In reality, the transition from AFS-2 to AFS-3 was gradual.) The revision was initiated at CMU and has been continued since mid-1989 at Transarc Corporation, a commercial venture involving many of the original implementers of AFS-3. The revision was motivated by the need to provide decentralized system administration, by the desire to operate over wide area networks, and by the goal of using industry standards in the implementation.

AFS-3 supports multiple administrative *cells*, each with its own servers, workstations, system administrators, and users. Each cell is a completely autonomous Andrew environment, but a federation of cells can cooperate in presenting users with a uniform, seamless filename space. The ability to decompose a distributed system into cells is important at large scale because it allows administrative responsibility to be delegated along lines that parallel institutional boundaries. This makes for smooth and efficient system operation.

The RPC protocol used in AFS-3 provides good performance across local and wide area networks. In conjunction with the cell mechanism, this network capability has made possible shared access to a common, nationwide file system, distributed over nodes such as MIT, the University of Michigan, and Dartmouth, as well as CMU.

Venus has been moved into the Unix

## Other contemporary distributed file systems

A testimonial to the importance of distributed file systems is the large number of efforts to build such systems in industry and academia. The following are some systems currently in use:

**Sun NFS** has been widely viewed as a de facto standard since its introduction in 1985. Portability and heterogeneity are the dominant considerations in its design. Although originally developed on Unix, it is now available for other operating systems such as MS-DOS.

**Apollo Domain** is a distributed workstation environment whose development began in the early 1980s. Since the system was originally intended for a close-knit team of collaborating individuals, scale was not a dominant design consideration. But large Apollo installations now exist.

**IBM AIX-DS** is a collection of distributed system services for the AIX operating system, a derivative of System V Unix. A distributed file system is the primary component of AIX-DS. Its goals include strict emulation of Unix semantics, ability to efficiently support databases, and ease of administering a wide range of installation configurations.

**AT&T RFS** is a distributed file system developed for System V Unix. Its most distinctive feature is precise emulation of local Unix semantics for remote files.

**Sprite** is an operating system for networked uniprocessor and multiprocessor workstations, designed at the University of California at Berkeley. The goals of the Sprite file system include efficient use of large main memory caches, diskless operation, and strict Unix emulation.

**Amoeba** is a distributed operating system built by the Free University and CWI (Mathematics Center) in Amsterdam. The first version of the distributed file system used optimistic concurrency control. The current version provides simpler semantics and has high performance as its primary objective.

**Echo** is a distributed file system currently being implemented at the System Research Center of Digital Equipment Corporation. It uses a primary site replication scheme, with reelection in case the primary site fails.

## Further reading


**Surveys**

Satyanarayanan, M., "A Survey of Distributed File Systems," in *Annual Review of Computer Science*, J.F. Traub et al., eds., Annual Reviews, Inc., Palo Alto, Calif., 1989.

Svobodova, L., "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys*, Vol. 16, No. 4, Dec. 1984.

**Individual systems**

**Amoeba**
van Renesse, R., H. van Staveren, and A.S. Tanenbaum, "The Performance of the Amoeba Distributed Operating System," *Software Practice and Experience*, Vol. 19, No. 3, Mar. 1989.

**Apollo Domain**
Levine, P., "The Apollo Domain Distributed File System" in *Theory and Practice of Distributed Operating Systems*, Y. Paker, J.-T. Banatre, and M. Bozyigit, eds., NATO ASI Series, Springer-Verlag, 1987.

**AT&T RFS**
Rifkin, A.P., et al., "RFS Architectural Overview" *Proc. Summer Usenix Conf.*, Atlanta, 1986, pp. 248-259.

**Echo**
Hisgen, A., et al., "Availability and Consistency Trade-Offs in the Echo Distributed File System," *Proc. Second IEEE Workshop on Workstation Operating Systems*, CS Press, Los Alamitos, Calif., Order No. 2003, Sept. 1989.

**IBM AIX-DS**
Sauer, C.H., et al., "RT PC Distributed Services Overview," *ACM Operating Systems Review*, Vol. 21, No. 3, July 1987, pp. 18-29.

**Sprite**
Ousterhout, J.K., et al., "The Sprite Network Operating System," *Computer*, Vol. 21, No. 2, Feb. 1988, pp. 23-36.

**Sun NFS**
Sandberg, R., et al., "Design and Implementation of the Sun Network File System," *Proc. Summer Usenix Conf.*, Portland, 1985, pp. 119-130.


kernel in order to use the *vnode* file intercept mechanism from Sun Microsystems, a de facto industry standard. The change also makes it possible for Venus to cache files in large chunks (currently 64 Kbytes) rather than in their entirety. This feature reduces file-open latency and allows a workstation to access files too large to fit on its local disk cache.

## Security in Andrew

A consequence of large scale is that the casual attitude toward security typical of close-knit distributed environments is not acceptable. Andrew provides mechanisms to enforce security, but we have taken care to ensure that these mechanisms do not inhibit legitimate use of the system. Of course, mechanisms alone cannot guarantee security; an installation also must follow proper administrative and operational procedures.

A fundamental question is who enforces security. 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 servers. Workstations may be owned privately or located in public areas. Andrew assumes that the hardware and software on workstations may be modified in arbitrary ways.

This section summarizes the main aspects of security in Andrew, pointing out the changes that occurred as the system evolved. These changes have been small compared to the changes for scalability. More details on security in Andrew can be found in an earlier work.[3]

**Protection domain.** The protection domain in Andrew is composed of *users* and *groups*. A user is an entity, usually a human, that can authenticate itself to Vice, be held responsible for its actions, and be charged for resource consumption. A

DOCKET ALARM

# Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

## Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

## Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

## Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

## API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

**LAW FIRMS**
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

**FINANCIAL INSTITUTIONS**
Litigation and bankruptcy checks for companies and debtors.

**E-DISCOVERY AND LEGAL VENDORS**
Sync your system to PACER to automate legal marketing.