COMPUTER

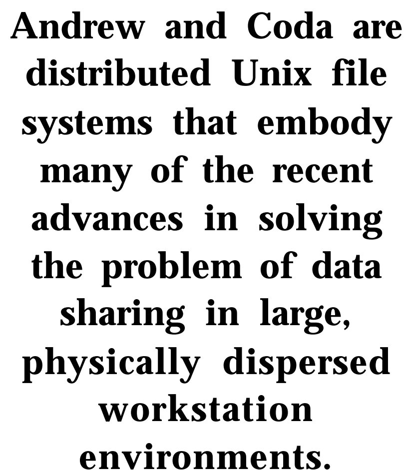
Scalable, Secure, and Highly Available Distributed File Access

Mahadev Satyanarayanan
Carnegie Mellon University

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

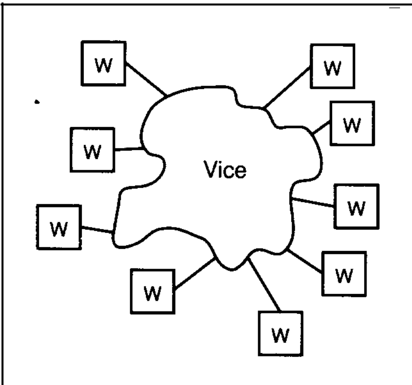


Figure 1. A high-level view of the Andrew architecture. The structure labeled "Vice" is a collection of trusted file servers and untrusted networks. The nodes labeled "W" are private or public workstations, or timesharing systems. Software in each such node makes the shared files in Vice appear as an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

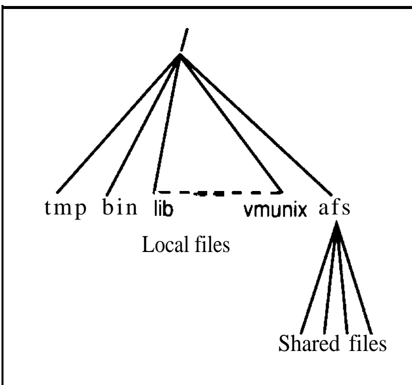


Figure 2. File system view at a workstation: how the shared files in Vice appear to a user. The subtree under the directory labeled "afs" is identical at all workstations. The other directories are local to each workstation. Symbolic links can be used to make local directories correspond to directories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

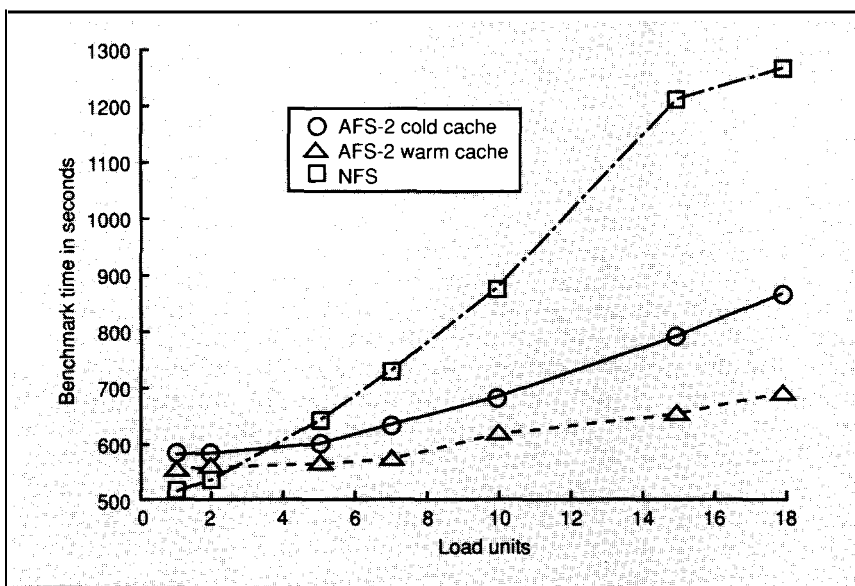


Figure 3. AFS-2 versus Sun NFS performance under load on identical client, server, and network hardware. A load unit consists of one client workstation running an instance of the Andrew benchmark. (Full details of the benchmark and experimental configuration can be found in Howard et al.,' from which 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-l'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.' 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

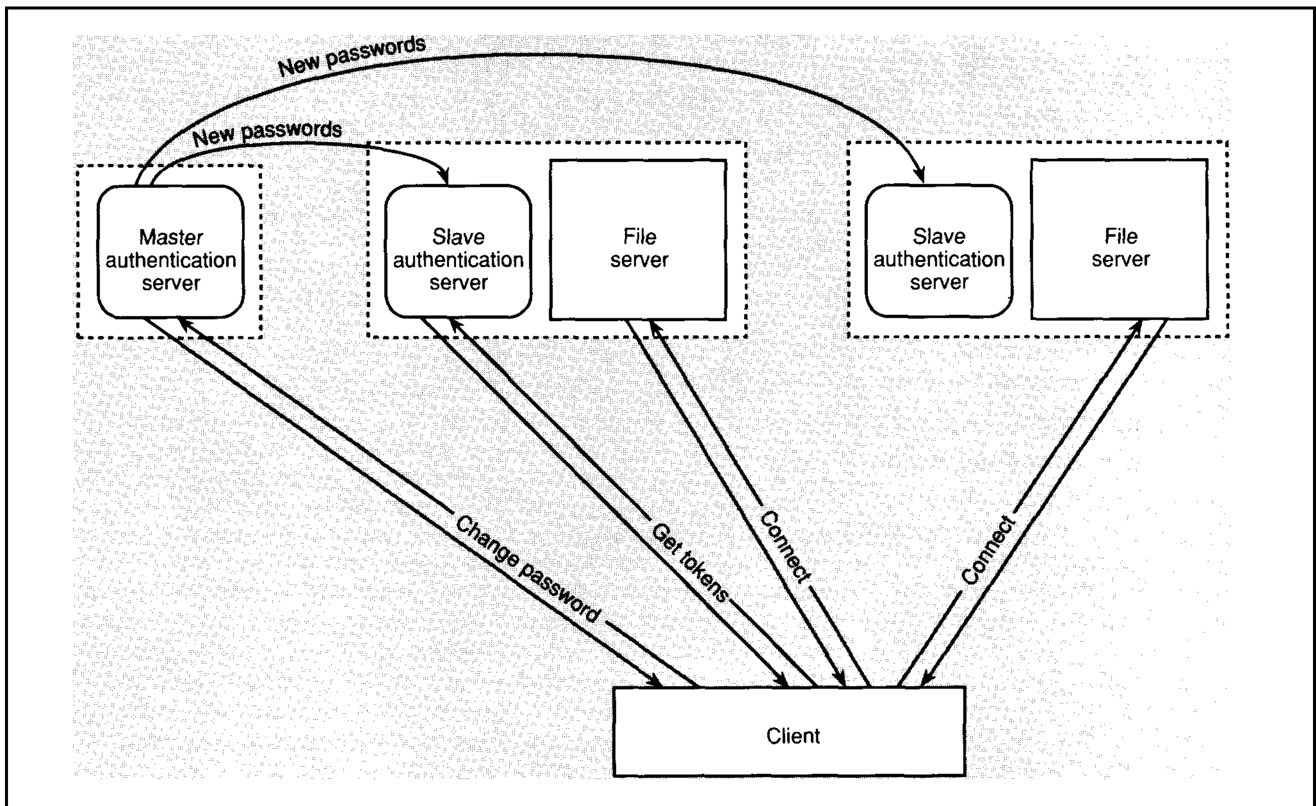


Figure 4. Major components and relationships involved in authentication in Andrew. Modifications such as password changes and additions of new users are made to the master authentication server, which distributes these changes to the slaves. When a user logs in, a client can obtain authentication tokens on the user's behalf from any slave authentication server. The client uses these tokens as needed to establish secure connections to file servers.

group is a set of other groups and users. Every group is associated with a unique user called its ***owner.***

AFS-1 and AFS-2 supported ***group inheritance,*** with a user's privileges being the cumulative privileges of all the groups it belonged to, either directly or indirectly. Modifications of the protection domain were made off line by system administrators and typically were reflected in the system once a day. In AFS-3, modifications are made directly by users to a protection server that immediately reflects the changes in the system. To simplify the implementation of the protection server, the initial release of AFS-3 does not support group inheritance. This may change in the future because group inheritance conceptually simplifies management of the protection domain.

One group is distinguished by the name System:Administrators. Membership in this group endows special administrative privileges, including unrestricted access to any file in the system. The use of a System:Administrators group rather than a pseudo-user (such as "root" in Unix systems) has the advantage that the actual identity of the user exercising special privileges is available for use in audit trails.

Authentication. The Andrew RPC mechanism provides support for secure, authenticated communication between mutually suspicious clients and servers, by using a variant of the Needham and Schroeder private key algorithm.[4] When a user logs in on a workstation, his or her password is used to obtain tokens from an authentication server. These tokens are saved by Venus and used as needed to establish secure RPC connections to file servers on behalf of the user.

The level of indirection provided by tokens improves transparency and security. Venus can establish secure connections to file servers without users' having to supply a password each time a new server is contacted. Passwords do not have to be stored in the clear on workstations. Because tokens typically expire after 24 hours, the period during which lost tokens can cause damage is limited.

As shown in Figure 4, there are multiple instances of the authentication server, each running on a trusted Vice machine. One of the authentication servers, the master, responds to updates by users and system administrators and asynchronously propagates the updates to other servers. The latter are slaves and only respond to queries. This design provides robustness by allowing users to log in as long as any slave or the master is accessible.

For reasons of standardization, theAFS-3 developers plan to adopt the Kerberos authentication system.[5] Kerberos provides the functionality of the Andrew authentication mechanism and closely resembles it in design.

File system protection. Andrew uses an access list mechanism for file protection. The total rights specified for a user are the union of the rights specified for the user and for the groups he or she belongs to. Access lists are associated with directories rather than individual files. The reduction

in state obtained by this design decision provides conceptual simplicity that is valuable at large scale. 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 decouple the problems of rapid revocation and propagation of group membership information and are particularly valuable in a large distributed system.

Although Vice actually enforces protection on the basis of access lists, Venus superimposes an emulation of Unix protection semantics. The owner component of the Unix mode bits on a file indicate readability, writability, or executability. These bits, which indicate what can be done to the file rather than who can do it, are set and examined by Venus but ignored by Vice. The combination of access lists on directories and mode bits on files has proved to be an excellent compromise between protection at fine granularity, conceptual simplicity, and Unix compatibility.

Resource usage. A security violation in a distributed system can manifest itself as an unauthorized release or modification of information or as a denial of resources to legitimate users. Andrew's authentication and protection mechanisms guard against unauthorized release and modification of information. Although Andrew controls server disk usage through a per-volume quota mechanism, it does not control resources such as network bandwidth and server CPU cycles. In our experience, the absence of such controls has not proved to be a problem. What has been an occasional problem is the inconvenience to the owner of a workstation caused by the remote use of CPU cycles on that workstation. The paper on security in Andrew[3] elaborates on this issue.

High availability in Coda

The Coda file system, a descendant of AFS-2, is substantially more resilient to server and network failures. The ideal that Coda strives for is constant data availability, allowing a user to continue working regardless of failures elsewhere in the system. Coda provides users with the benefits of a shared data repository but allows them to rely entirely on local resources when that repository is partially or totally inaccessible.

When network partitions occur, Coda allows data to be updated in each partition but detects and confines conflicting updates as soon as possible after their occurrence. It also provides mechanisms to help users recover from such conflicts.

A related goal of Coda is to gracefully integrate the use of portable computers. At present, users manually copy relevant files from Vice, use the machine while isolated from the network, and manually copy updated files back to Vice upon reconnection. These users are effectively performing manual caching of files with write-back on reconnection. If one views the disconnection from Vice as a deliberately induced failure, it is clear that a mechanism for supporting portable machines in isolation is also a mechanism for fault tolerance.

By providing the ability to move seamlessly between zones of normal and disconnected operation, Coda may simplify the use of cordless network technologies such as cellular telephone, packet radio, or infrared communication in distributed file systems. Although such technologies provide client mobility, they often have intrinsic limitations such as short range, inability to operate inside steel-framed buildings, or line-of-sight constraints. These shortcomings are reduced in significance if clients are capable of temporary autonomous operation.

The design of Coda was presented in detail in a recent paper.[6] A large subset of the design has been implemented, and work is in progress to complete the implementation. One can sit down at a Coda workstation today and execute Unix applications without recompilation or relinking. Execution continues transparently when contact with a server is lost due to a crash or network failure. In the absence of failures, using a Coda workstation feels no different from using an AFS-2 workstation.

Design overview. The Coda design retains key features of AFS-2 that contribute to scalability and security:

- Clients cache entire files on their local disks. From the perspective of Coda, whole-file transfer also offers a degree of intrinsic resiliency. Once a file is cached and open at a client, it is immune to server and network failures. Caching on local disks is also consistent with our goal of supporting portable machines.
- Cache coherence is maintained by the use of callbacks.
- Clients dynamically find files on servers and cache location information.
- Token-based authentication and end-to-end encryption are used as the basis of security.

Coda provides failure resiliency through two distinct mechanisms. It uses *server replication,* or the storing of copies of files at multiple servers, to provide a highly available shared storage repository. When no server can be contacted, the client resorts to *disconnected operation,* a mode of execution in which the client relies solely on cached data. Neither mechanism is adequate alone. While server replication increases the availability of all shared data, it does not help if all servers fail or if all are inaccessible due to a network failure adjacent to a client. On the other hand, permanent disconnected operation is infeasible. The disk storage capacity of a client is a small fraction of the total shared data. Permanent disconnected operation is also inconsistent with the Andrew model of treating each client's disk merely as a cache. Key advantages of the Andrew architecture, namely mobility and a user's ability to treat any workstation as his or her own, are lost.

From a user's perspective, transitions between these complementary mechanisms are seamless. A client relies on server replication as long as it remains in contact with at least one server. It treats disconnected operation as a measure of last resort and reverts to normal operation at the earliest opportunity. A portable client that is isolated from the network is effectively operating in disconnected mode.

When network partitions occur, Coda allows data to be updated in each partition but detects and confines conflicting updates as soon as possible after their occurrence. It also provides mechanisms to help

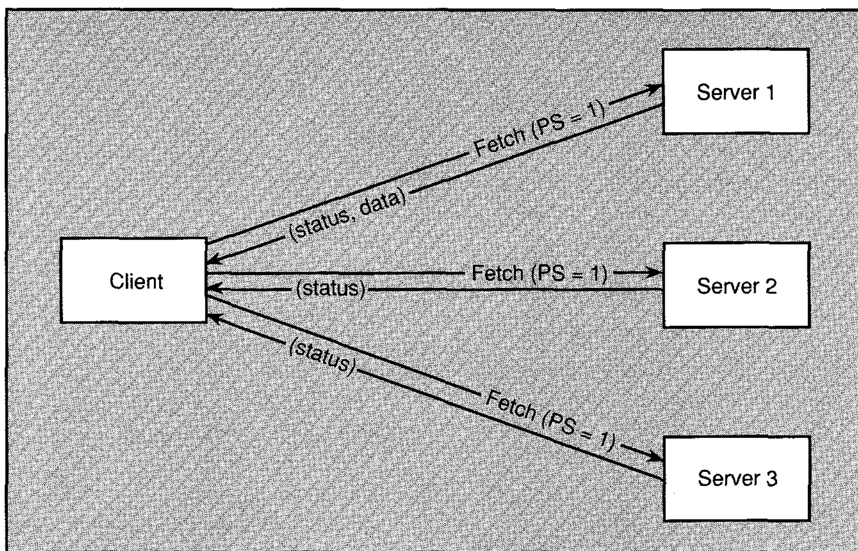


Figure 5. Servicing a cache miss in Coda: the events that follow from a cache miss at the client. Both data and status are fetched from Server 1, which is the preferred server (PS). Only status is fetched from Server 2 and Server 3. The calls to all three servers occur in parallel.

users recover from such conflicts, This strategy is *optimistic,* in contrast to a *pessimistic* strategy that would preserve strict consistency by disallowing updates in all but one partition. We chose an optimistic strategy for two reasons: First, we saw no clean way to support disconnected operation with a pessimistic strategy. Second, it is widely believed that sequential write sharing between users is relatively infrequent in Unix environments, so conflicting updates are likely to be rare.

Coda provides a scalable and highly available approximation of Unix semantics. We arrived at this semantics on the basis of our positive experience with AFS-2. In the absence of failures, Coda and AFS-2 semantics are identical. On open, the latest copy of a file in the system is cached from Vice. Read and write operations are made to the cached copy. On close, the modified file is propagated to Vice. Future opens anywhere in the system will see the new copy of the file. In the presence of failures, Coda and AFS-2 semantics differ. An open or close in AFS-2 would fail if the server responsible for the file was inaccessible. In Coda, an open fails only on a cache miss during disconnected operation or if aconflict is detected. A close fails only if a conflict is detected.

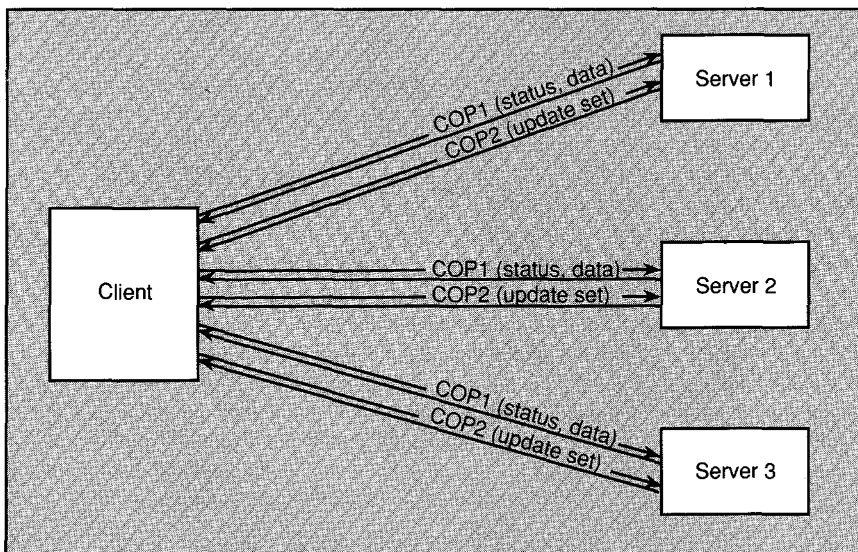


Figure 6. A store operation in Coda: the two phases of the Coda update protocol. In the first phase, COP1, the three servers are sent new status and data in parallel. In the later asynchronous phase, COP2, the update set is sent to these servers. COP2 also occurs in parallel and can he piggybacked on the next COP1 to these servers.

Server replication. The unit of replication in Coda is a volume. A *replicated volume* consists of several physical volumes, or replicas, that are managed as one logical volume by the system. Individual replicas are not normally visible to users. The set of servers with replicas of a volume constitutes its *volume storage group* (VSG). The degree of replication and the identity of the replication sites are specified when a volume is created. Although these parameters can be changed later, we do not anticipate such changes to be frequent. For every volume from which it has cached data, Venus keeps track of the subset of the VSG that is currently accessible. This subset is called the *accessible VSG* (AVSG). Different clients may have different AVSGs for the same volume at a given instant. Venus performs periodic probes to detect shrinking or enlargement of the AVSGs from which it has cached data. These probes are relatively infrequent, occurring once every 10 minutes in our current implementation.

Coda integrates server replication with caching, using a variant of the read-one, write-all strategy. This variant can be characterized as read-one-data, read-all-status, write-all. In the common case of a cache hit on valid data, Venus avoids contacting the servers altogether. When servicing a cache miss, Venus obtains data from one member of its AVSG, known as *thepreferred server.* The PS can be chosen at random or on the basis of performance criteria such as physical proximity, server load, or server CPU power. Although data is transferred only from one server, Venus contacts the other servers to collect their versions and other status information. Venus uses this information to check whether the accessible replicas are equivalent. If the replicas are in conflict, the

system call that triggered the cache miss is aborted. If the replicas are not in conflict but some replicas are stale, the AVSG is notified asynchronously that a refresh is necessary. In the special case where the data on the PS is stale, a new PS is selected and the fetch is repeated. The message exchange in the normal case, where there is no conflict and the PS has the latest copy of the data, is illustrated in Figure 5. A callback is established with the PS as a side-effect of successfully fetching the data.

When a file is closed after modification, it is transferred to all members of the AVSG. This approach is simple to implement and maximizes the probability that every replication site has current data at all times. Server CPU load is minimized because the burden of data propagation is on the client rather than the servers. This in turn improves scalability, since the server CPU is the bottleneck in many distributed file systems. Operations that update directories, such as creating a new directory or removing a file, are also written through to all AVSG members.

Because its replication scheme is optimistic, Coda checks for existing conflicts on each server operation. The update protocol also guarantees eventual detection of new conflicts caused by the update. This protocol consists of two phases, COP1 and COP2, where COP stands for Coda *optimistic protocol.* The first phase performs the semantic part of the operation, such as transferring file contents, making a directory entry, or changing an access list. Each server verifies that its copy does not conflict with the client's copy before performing the update. The second phase distributes to the servers a data structure called the *update set,* which summarizes the client's knowledge of who performed the COP1 operation. The update set maintains the version information used in conflict detection. Figure 6 illustrates the message exchange in a store operation (which corresponds to a file close).

Two protocol optimizations improve performance: First, latency is reduced by Venus's returning control to the user after completion of COP1 and performing COP2 asynchronously. Second, network and server CPU loads can be reduced by Venus's piggybacking the asynchronous COP2 messages on subsequent COP1 calls to the same VSG.

At present, a server performs no explicit remote actions upon recovery from a crash. Rather, it depends on clients to notify it of stale or conflicting data. Although this lazy strategy does not violate Coda's consistency guarantees, it does increase the chances of a future conflict. A better approach, which we plan to adopt in the future, is for a recovering server to contact other servers to bring itself up to date.

Each server operation in Coda typically involves multiple servers. If the operation were carried out sequentially, latency would increase significantly. Venus therefore communicates with replication sites in parallel, using a parallel RPC mechanism. This mechanism has been extended to use hardware multicast support, if available, to reduce the latency and network load caused by shipping large files to multiple sites. Shipping a large file to three servers in our current implementation typically takes about 10 percent longer than shipping it to one server.

Operation latency is usually a major concern with replication schemes, but server replication in Coda has worked well. Controlled experiments on identical client and server hardware show that under light loads Coda's performance is within five percent of the performance of Andrew's current release. Thus, the cost of replication is primarily the storage cost for additional replicas at the servers. The current implementation of Coda does not perform quite as well under heavy load. Our measurements indicate specific areas for improvement, and we are confident that these changes will result in an implementation with significantly better performance under load.

Disconnected operation. Disconnected operation begins at a Coda workstation when no member of a VSG is accessible. Clients view it as a temporary state and revert to normal operation at the earliest opportunity. A client may be operating in disconnected mode with respect to some volumes but not others. Disconnected operation is transparent to a user unless a cache miss occurs. A cache miss normally aborts the system call that triggered the reference, but it is possible to arrange for such system calls to block. Return to normal operation is also transparent, unless a conflict is detected.

To reduce the chances of a cache miss during disconnected operation, Coda allows a user to specify a prioritized list of files and directories that Venus should strive to retain in the cache. Objects of the highest priority level are "sticky" and must be retained at all times. As long as the local disk is large enough to accommodate all sticky files and directories, the user can always access them. Since it is often difficult to know exactly what file references are generated by a certain set of high-level user actions, Coda provides the ability for a user to bracket a sequence of high-level actions and for Venus to note the file references generated during these actions. The implementer of an application can also provide a list of files that should be made sticky for the application to work when disconnected.

Disconnected operation with respect to a particular volume ends when Venus reestablishes connection with any member of the volume's VSG. Reconnection results from a successful probe—either one of Venus's periodic probes or one manually induced by a user-level command. The transition from disconnected operation invokes a process of reintegration. For each cached file or directory that has been created, deleted, or modified on the client during disconnected operation, Venus executes a sequence of update operations to make AVSG replicas identical to the cached copy. Reintegration proceeds top-down, from the root to the leaves of modified subtrees.

Update operations during reintegration may fail for one of two reasons. First, there may be no authentication tokens that Venus can use to communicate securely with AVSG members. Users whose tokens expire during disconnected operation may forestall reintegration until they have reacquired valid tokens to minimize this possibility. Second, conflicts may be detected. Given our model in which servers rather than clients are dependable storage repositories, we felt that the proper approach to handling both of these situations was to find a temporary home on servers for the data in question and to rely on a user to resolve the problem later.

The temporary repository is realized as a *covolume* for every replica of every volume in Coda. Covolumes are similar in spirit to lost+found directories in Unix. They have a flat name space derived from the original low-level identifiers of the objects they contain. Covolumes are not directly visible to users but are accessed indirectly through a repair tool as described in the next section. *Migrate* is the operation that transfers a file or directory from a workstation to a covolume. Having a covolume per replica allows us to perform migration immediately upon reintegration failure rather than waiting for connection to a particular site. The storage overhead of this approach is usually small since a covolume is almost always empty.

Mechanisms for building distributed file systems

Although there is considerable diversity in the manner in which distributed file systems are put together, all the current ones are built from a surprisingly small number of basic mechanisms. This sidebar presents the most important of these mechanisms and examples of how different systems have used them.

• **Mount points.** The mount mechanism in Unix enables the gluing together of filename spaces to provide applications with a single, seamless, hierarchically structured name space. In a distributed Unix file system, the mount mechanism provides a natural hook on which to hang a remote subtree.

There are two different ways to use the mechanism. The simpler approach is used by systems such as Sun NFS, in which each client individually mounts subtrees from servers. Although this approach is easier to implement, it has the disadvantage that the shared name space may not 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 Sun NFS) to automate and centralize mounts.

The alternative approach is to embed mount information in the data stored in the file servers. Andrew and Coda, for example, use mount points embedded in volumes. Sprite uses remote links for a similar purpose. This approach makes it relatively simple to ensure that all clients see the same shared filename space at all times.

• **Client caching.** 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. In most systems, clients maintain the cache in their main memory. Andrew and Coda, in contrast, cache on the local disk, with a further level of caching in main memory.

A key issue in caching is the size of the cached units of data. Most distributed file systems cache individual pages of files. Coda, Amoeba, AFS-1, and AFS-2 cache entire files. AFS-3 and most other file systems cache portions of a file.

Cache validation can be done in two ways. One approach is for the client to contact the server for validation. The alternative approach, used in AFS-2, AFS-3, Coda, AIX-DS, and Echo, 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. AIX-DS usually propagates changes to the server only when the file is explicitly flushed. Andrew and Coda propagate 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 it increases the possibility of server data being stale due to a client crash.

• **Hints.** In the context of distributed systems, a hint 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. 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 and Coda cache individual entries from the volume location database. Apollo Domain uses a more elaborate location scheme incorporating a hint manager.

• **Bulk data transfer.** Network communication overhead caused by protocol processing typically accounts for a major portion of the latency in a distributed file system. Transferring data in bulk reduces this overhead by amortizing fixed protocol overheads over many consecutive pages of a file. For effectiveness, bulk transfer protocols depend on spatial locality of reference within files.

The degree to which bulk transfer is exploited varies from system to system. Amoeba, Andrew, and Coda are critically dependent on it. Sun NFS and Sprite exploit bulk transfer by using very large packet sizes, typically 8 kilobytes. Bulk transfer protocols will increase in importance as distributed file systems spread across networks of wider geographic area and thus have greater inherent latency.

• **Encryption.** Encryption is an indispensable building block for enforcing security in a distributed system. It is used for remote authentication and for preventing unauthorized release and modification of data transmissions. The national standard DES (data encryption standard) is the most commonly used form of private-key encryption. The seminal work of Needham and Schroeder on the use of encryption for authentication is the basis of all current security mechanisms in distributed file systems.

Authentication can be performed with private or public keys. In the private-key schemes used by Kerberos and Andrew, a physically secure authentication server 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 encrypted with user passwords. The latter approach has the attractive characteristic that physical security of the authentication server is unnecessary. Its major drawback is that public-key encryption is computationally more expensive.

• **Replication.** Replication of data at multiple servers is the primary mechanism for providing high availability. The more recent file systems such as Coda and Echo provide read-write replication of data. Amoeba supports read-write replication at the directory level because files are immutable in that system. Although read-write replication is well understood theoretically, little experience of its use exists as yet.

More experience has been gathered with read-only data replication, which is supported by systems such as Sun NFS and Andrew. Though suitable only for files that change relatively rarely, it is valuable because many critical files (such as system binaries) possess this property.

We are currently in the midst of implementing disconnected operation. Although we are confident of our ability to support short-term disconnected operation (for a few minutes or hours), it remains to be seen whether long-term disconnected operation (for days or weeks) is feasible. Our concerns center on the overall size of the working set and on the predictive power of our caching strategies. Our own experience, and that of others, suggests that a cache of several tens of megabytes should be adequate for a typical disconnection of less than a day. Less obvious is whether any anticipatory caching strategy can, with a reasonable cache size, provide the near-perfect hit rates required for long-term disconnected operation.

Conflict resolution. When a conflict is detected, Coda first attempts to resolve it automatically. Since Unix files are untyped byte streams, there is no information to automate their resolution. A directory, on the other hand, is an object whose semantics is completely known and whose resolution can often be automated. For example, partitioned creation of uniquely named files in the same directory can be handled automatically by selectively replaying the missing creates. If automated resolution is not possible, Coda marks all accessible replicas of the object inconsistent and moves them to their respective covolumes. This ensures damage containment because normal operations on these replicas will subsequently fail.

Coda provides a repair tool to assist users in manually resolving conflicts. It uses a special interface to Venus so that requests from the tool are distinguishable from normal file system requests. This enables the tool to overwrite inconsistent files and to perform directory operations on inconsistent directories. The tool has evolved along with the rest of our system. Three generations of the tool are described here: the tool for the currently implemented system, the one we are working on at present, and a successor that will incorporate the current wish list.

In the first-generation tool, inconsistent files and directories are marked in conflict but are not moved to covolumes. Disconnected operation is not supported because there is nowhere to migrate objects to. When the tool is invoked on a given object, it mounts the accessible replicas of the object's volume in a scratch area of the name space. The user can then use normal Unix applications to inspect the replicas. The replicas are mounted in read-only mode so that the user cannot inadvertently alter anything. When the user has decided on a fix (such as selecting the version in one of the replicas to be the new permanent one), the tool performs the fix and cleans up the workspace.

The general problem of sharing information effectively in large distributed systems is far from being solved. The next decade poses many challenges and promises to be a fertile and exciting period for researchers in this area.

The second-generation tool supports disconnected operation because it knows about covolumes. Inconsistent objects are immediately moved to the associated covolume when the inconsistency is detected. When the tool is invoked, it constructs a temporary workspace and mounts, read-only, the covolumes as well as the replicas of the object. As before, the user can navigate through the replicas. However, names of inconsistent objects now correspond to objects in the associated covolumes. The tool applies the fix and cleans up the workspace as the first-generation tool does.

The primary refinement provided by the third-generation tool will be a considerably simplified user interface. Venus, in conjunction with the tool, will present the illusion of an in-place "explosion" of inconsistent objects into their distinct versions. Invocation of the tool will put Venus in a mode whereby inconsistent objects can be viewed within the existing name space. In this mode, Venus will map an inconsistent file or directory into a read-only directory with the same name as the original. This directory will be populated with entries translated by Venus to the versions in the various covolumes. The tool will handle the fix phase of the repair in the same way as the second-generation tool.

Throughout the evolution of the Andrew and Coda file systems, the underlying model of computation has remained unchanged. A small collection of trusted servers jointly provides a shared data repository for a much larger number of untrusted workstations. The system design facilitates incremental growth by the addition of users and workstations. The security of the system is not contingent upon the integrity of the workstations or of the network.

The problems of scalability, security, and availability will continue to be important as distributed file systems grow in size. In addition, three other problems will be of fundamental importance to the broader goal of effective data sharing in large distributed systems. These are the problems of heterogeneity, access to diverse types of data, and rapid search.

As a distributed system grows, it tends to become more heterogeneous. 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 accommodated by ad hoc mechanisms. Unfortunately, heterogeneity cannot be ignored since it is likely to be a chronic problem.

Alternative models of data are likely to become more important in the future. Relational databases are already in widespread use in certain application domains. Speech, music, images, and video are examples of other forms of data that the repositories of the future will have to store and retrieve. We presently have little knowledge of how to share such diverse types of data in large-scale distributed systems.

Finding data in a large distributed file system is already difficult. As distributed storage repositories grow larger and store more diverse types of data, the problem of searching for relevant information will become acute. This is another area where we have barely scratched the surface.

In conclusion, we have made much progress in the design and implementation of distributed file systems over the last decade. Andrew and Coda embody many of the key advances made during this period. But the general problem of sharing information effectively in large distributed systems is far from being solved. The next decade poses many challenges and promises to be a fertile and exciting period for researchers in this area. ■

Acknowledgments

The Andrew file system was built by the File System Group of the Information Technology Center at Carnegie Mellon University. The membership of this group over time has included Ted Anderson, Sailesh Chutani, John Howard, Michael Kazar, Sherri Menees Nichols, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, Michael West, and Edward Zayas. Contributions to the early design of Andrew were also made by David Gifford and Alfred Spector.

Coda is being built in the School of Computer Science at Carnegie Mellon University. Contributors to Coda include James Kistler, Puneet Kumar, Maria Okasaki, Mahadev Satyanarayanan, Ellen Siegel, Walter Smith, and David Steere.

James Kistler assisted in writing this article.

This research was supported by the National Science Foundation (contract No. CCR-8657907), Defense Advanced Research Projects Agency (order No. 4976, contract No. F33615-87-C-1499), IBM Corporation (faculty development award, graduate fellowship, and the Andrew project), and Digital Equipment Corporation (equipment grant). The views and conclusions in this article are those of the author and do not represent the official policies of the funding agencies or of Carnegie Mellon University.

References

1. J.H. Howard et al., "Scale and Performance in a Distributed File System," *ACM Trans. Computer Systems*, Vol. 6, No. 1, Feb. 1988, pp. 51-81.

2. M.N. Nelson, B.B. Welch, and J.K. Ousterhout, "Caching in the Sprite Network File System," *ACM Trans. Computer Systems*, Vol. 6, No. 1, Feb. 1988, pp. 134-154.

3. M. Satyanarayanan, "Integrating Security in a Large Distributed System," *ACM Trans. Computer Systems*, Vol. 7, No. 3, Aug. 1989, pp. 247-280.

4. R.M Needham and M.D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Comm. ACM*, Vol. 21, No. 12, Dec. 1978, pp. 993-998.

5. J.G. Steiner, C. Neumann, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *Proc. Usenix Conf.*, Dallas, Texas, Feb. 1988, pp. 191-202.

6. M. Satyanarayanan et al., "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. Computers*, Vol. 39, No. 4, Apr. 1990, pp. 447-459.

Mahadev Satyanarayanan is an associate professor of computer science at Carnegie Mellon University. His research addresses the general problem of sharing access to information in large-scale distributed systems. He was one of the principal architects and implementers of the Andrew file system and currently leads the Coda project. His work on Scylla explored access to relational databases in a distributed workstation environment. His previous research included the design of the CMU-CFS file system, measurement and analysis of file usage data, and the modeling of storage systems.

Satyanarayanan received the PhD in computer science from Carnegie Mellon in 1983, after receiving a bachelor's degree in electrical engineering and a master's degree in computer science from the Indian Institute of Technology, Madras. He is a member of the IEEE, the IEEE Computer Society, ACM, and Sigma Xi, and has been a consultant to industry and government. He was named a Presidential Young Investigator by the National Science Foundation in 1987.

Readers can write to Satyanarayanan at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

May 1990

Copyright © 1990 The Institute of Electrical and Electronics Engineers, Inc.
Reprinted with permission from **COMPUTER**,
10662 Los Vaqueros Circle, Los Alamitos, CA 90720