The Influence of Scale on Distributed File System Design

Mahadev Satyanarayanan, *Member, IEEE*

Abstract— **Scale should be recognized as a primary factor influencing the architecture and implementation of distributed systems. This paper uses Andrew and Coda, distributed file systems built at Carnegie Mellon University, to validate this proposition. Performance, operability, and security are dominant considerations in the design of these systems. Availability is a further consideration in the design of Coda. Client caching, bulk data transfer, token-based mutual authentication, and hierarchical organization of the protection domain have emerged as mechanisms that enhance scalability. The separation of concerns made possible by functional specialization has also proved valuable in scaling. Heterogeneity is an important by-product of growth, but the mechanisms available to cope with it are rudimentary. Physical separation of clients and servers turns out to be a critical requirement for scalability.**

Index Terms— **Scalability, distributed file systems, Andrew, Coda, security, performance, availability, heterogeneity, replication, caching, large-scale distributed systems, design principles for scalability.**

I. Introduction

SOFTWARE engineering focuses on programming in the large, recognizing that *size* is a first-order influence on the structure of programs. This paper puts forth the analogous view that the *scale* of a distributed system is a fundamental influence on its design. Mechanisms which work well in a small distributed system fail to be adequate in the context of a larger system. *Scalability* should thus be a primary consideration in the design of distributed systems.

How does one measure "scale"? One can interpret the term in many ways. For example, it can refer to the number of nodes in a distributed system. Alternatively, it could be the number of users of the system. These definitions are equivalent in a distributed workstation environment, where each workstation is dedicated to a user or is shared by a small number of users. Another interpretation is that scale refers to the number of organizational boundaries encompassed by the system. Human limitations necessitate the grouping of users and equipment into autonomous or semi-autonomous organizations for management purposes. Hence a distributed system that has many users or nodes will also span many organizations. Regardless of the specific metric of scale, the designs of distributed systems that scale well are fundamentally different from less scalable designs.

In this paper we describe the lessons we have learned about scalability from the *Andrew File System* and the *Coda File System*. These systems are particularly appropriate for our discussion, because they have been designed for growth to thousands of nodes and users. We focus on distributed file systems, because they are the most widely used kind of distributed system in existence today, and because we have first-hand experience in their design, implementation, and use. But much of this paper also applies to other kinds of distributed services.

We begin our discussion by examining why growth occurs in distributed systems and what its consequences are. We then give a brief overview of Andrew and Coda. The bulk of the paper consists of a detailed examination of specific aspects of Andrew and Coda that were significantly influenced by the need to scale well. Wherever possible, we contrast our design choices with those of other related systems. We end the paper with a summary of what we have learned in the form of a set of design principles.

II. Origin and Consequences of Scale

There is relentless pressure on successful distributed systems to increase in scale. It is easy to see why. A distributed system simplifies communication between users and the sharing of information between them. As a distributed system grows, enabling more users to communicate and more information to be shared, it becomes an increasingly valuable resource. At some point in the evolution of the system, access to it is viewed as a necessity rather than luxury by the user community. There is then considerable incentive to allow users who were originally outside the scope of the system to be included in it. Growth thus acquires a momentum of its own.

The economic impact of growth is closely related to how cost is reflected. The optimal system design for an organization where all system costs are borne centrally will be different from one for an organization where some costs are borne centrally and others are passed on to the end users. The latter model is more common and favors designs in which the cost of incremental growth is almost entirely borne by the individuals benefiting from that growth. This in turn

Manuscript received January 27, 1988; revised September 10, 1991. Recommended by M. Evangelist. This work was supported by the Defense Advanced Research Projects Agency under Contract No. F33615-90-C-1465, ARPA Order No. 7597 (Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB), by the National Science Foundation under Contract No. CCR-8657907, by the IBM Corporation (Andrew Project, Faculty Development Award, Research Initiation Grant), by the Digital Equipment Corporation (External Research Project Grant), and by Bellcore (Information Networking Research Grant).

The author is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 9104757.

implies that incremental growth should minimally impact central resources.

In our experience, growth also has other serious consequences. For example, performance and operability become dominant concerns. Lax security is no longer acceptable. Precise emulation of single-site interface semantics becomes difficult. Heterogeneity of hardware and software is more likely. In general, algorithms and techniques that work well at small scale degenerate in nonobvious ways at large scale.

III. Overview of Andrew and Coda

The Andrew File System was developed at Carnegie Mellon University (CMU) from 1983 to 1989. Today, it spans 40 sites on the Internet across the U.S. On the CMU campus alone it includes over 1000 workstations and is used by more than 3000 regular users. Further development and commercial distribution of this system is now being done by the Transarc Corporation.

Andrew is designed for an environment consisting of a large collection of untrusted clients with local disks. These clients are connected via a high bandwidth network to a small number of trusted servers, collectively called *Vice*. Users execute applications only at clients and see a location-transparent shared Unix[1] file system that appears as a single subtree of the local file system. To improve performance, clients cache files and directories from Vice. Cache management and emulation of Unix file system semantics is done at each client by a cache manager called *Venus*.

The design of Andrew has evolved over time, resulting in three distinct versions, called AFS-1, AFS-2, and AFS-3. In this paper the unqualified term "Andrew" applies to all three versions of the file system. The design of a fourth version, AFS-4, is proprietary at the present time and is therefore not discussed here.

As users become more dependent on distributed file systems, the *availability* of data in them becomes increasingly important. Today, a single failure in Andrew can seriously inconvenience many users for significant periods of time. Coda, whose development began in 1987 at CMU, strives to provide high availability while retaining the scalability of Andrew.

Coda provides resiliency to server and network failures through the use of two distinct but complementary mechanisms. One mechanism, *server replication*, stores copies of a file at multiple servers. The other mechanism, *disconnected operation*, is a mode of execution in which a caching site temporarily assumes the role of a replication site. Disconnected operation is particularly useful for supporting portable clients. Coda has been in serious daily use by a small user community for about a year, and substantial growth is anticipated in the near future.

Both Andrew and Coda have been described extensively in the literature [6], [7], [9], [18]–[21], [25]. In contrast to those papers, our discussion here is narrowly focused. In each of the following sections we highlight one aspect of Andrew or Coda that is a consequence of scale or contributes significantly to scalability. Wherever plausible alternative design choices exist, we evaluate our choices from the perspective of scalability.

IV. Scale-Related aspects of Andrew and Coda

A. Location Transparency

A fundamental issue is how files are named and located in a distributed environment. Andrew and Coda offer true *location transparency*: the name of a file contains no location information. Rather, this information is obtained dynamically by clients during normal operation. Consequently, administrative operations such as the addition or removal of servers and the redistribution of files on them are transparent to users. In contrast, some file systems require users to explicitly identify the site at which a file is located. For example, Unix United [2] uses pathname constructs of the form "*/../machine/localpath*" and Vax/VMS [4] uses "*machine::device:localpath.*"

The embedding of location information in a logical name space has a number of negative consequences. First, users have to remember machine names to access data. Although feasible in a small environment, this becomes increasingly difficult as the system grows in size. It is simpler and more convenient for users to remember a logical name devoid of location information. A second problem is that it is inconvenient to move files between servers. Since changing the location of a file also changes its name, file names embedded in application programs and in the minds of users become invalid. Unfortunately, data movement is inevitable when storage capacity on a server is exceeded or when a server is taken down for extended maintenance. The problem is more acute in large systems, because the likelihood of these events is greater.

Another alternative, used in Sun NFS [17], is to establish an association between a pathname on a remote machine and a local name. The association is performed using the *mount* mechanism in Unix when a machine is initialized, and remains in effect until the machine is reinitialized. Although this approach avoids pathnames with embedded server identification, it is not as flexible. If a file is moved from one server to another it will not be accessible under its original name unless the client is reinitialized.

Location transparency can be viewed as a binding issue. The binding of location to name is static and permanent when pathnames with embedded machine names are used. The binding is less permanent in a system like Sun NFS. It is most dynamic and flexible in Andrew and Coda. Usage experience has confirmed the benefits of a fully dynamic location mechanism in a large distributed environment.

B. Client Caching

The caching of data at clients is undoubtedly the architectural feature that contributes most to scalability in a distributed file system. Caching has been an integral part of the Andrew and Coda designs from the beginning. Today, every distributed file system in serious use uses some form of caching. Even AT&T's RFS [16], which initially avoided caching in the interests of strict Unix emulation, now uses it. In implementing

[1] Unix is a trademark of the AT&T Corporation.

caching one has to make three key decisions: where to locate the cache, how to maintain cache coherence, and when to propagate modifications.

Andrew and Coda cache on the local disk, with a further level of file caching by the Unix kernel in main memory. Most other distributed file systems maintain their caches only in main memory. Disk caches contribute to scalability by reducing network traffic and server load on client reboots, a surprisingly frequent event in workstation environments. They also contribute to scalability in a more indirect way by enabling disconnected operation in Coda. The latter feature is critically dependent upon a local disk or some other form of local nonvolatile storage. Since disconnected operation allows users to continue in the face of remote failures, and since the latter tend to be more numerous as a system grows, caching on local disks can be viewed as indirectly contributing to scalability.

Cache coherence can be maintained in two ways. One approach is for the client to validate a cached object upon use. This strategy, used in AFS-1 and Sprite [12], results in at least one interaction with a server for each open of a file. A more scalable approach is used in AFS-2, AFS-3, Coda, and Echo [5]. When a client caches an object, the server hands out a promise (called a *callback* or *token*) that it will notify the client before allowing any other client to modify that object. Although more complex to implement, this approach minimizes server load and network traffic, thus enhancing scalability. Callbacks further improve scalability by making it viable for clients to translate pathnames entirely locally.

Maintaining cache coherence is unnecessary if the data in question can be treated as a *hint* [28]. 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. Hints improve scalability by obviating the need for a cache coherence protocol. Of course, 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. Andrew and Coda, for instance, cache individual mappings of volumes to servers. Similarly, Sprite caches mappings of pathname prefixes to servers. A more elaborate location scheme, incorporating a hint manager, is used by Apollo Domain [8].

Existing systems use one of two approaches to propagating modifications from client to server. Write-back caching, used in Sprite and Echo, is the more scalable approach. When operating disconnected, a Coda client is effectively using deferred write-back caching. However, in the connected mode the current implementation of Coda uses write-through caching. We plan to change this to write-back caching in the near future. Because of implementation complexity and to reduce the chances of server data being stale due a client crash, Andrew uses a write-through caching scheme. This is a notable exception to scalability being the dominant design consideration in Andrew. Sun NFS is another example of a system that synchronously flushes dirty data to the server upon close of a file.

C. Bulk Data Transfer

An important issue related to caching is the granularity of data transfers between client and server. The approach used in AFS-1 and AFS-2 is to cache entire files. This enhances scalability by reducing server load, because clients need only contact servers on file `open` and `close` requests. The far more numerous `read` and `write` operations are invisible to servers and cause no network traffic. Whole-file caching also simplifies cache management, because clients only have to keep track of files, not individual pages, in their cache. Amoeba [10] and Cedar [23] are examples of other systems that employ whole-file caching.

Caching entire files has at least two drawbacks. First, files larger than the local disk cannot be accessed. Second, the latency of open requests is proportional to file size, and can be intolerable for extremely large files. To avoid these problems, AFS-3 uses partial-file caching. However, usage experience with AFS-3 at CMU has not demonstrated substantial improvement in usability or performance due to partial-file caching. On the contrary, it has exposed the following unexpected problem.

By default, Unix uses the file system rather than a separate swap area as backing store for the code segments of executing programs. With partial-file caching, parts of this backing store may not be present on the client's local disk; only the server is guaranteed to have the full image. This image is overwritten when a new version of a program is copied into AFS-3. If a client started executing the program prior to the copy, there is a possibility that page faults from that instance of the program may no longer be serviceable. This problem does not arise with whole-file transfer, because a local copy of the entire program is guaranteed to be present when execution of the program is begun. For partial-file caching to work correctly a server must prevent overwriting of a file as long as any client in the system is executing a copy of that file; or, a server must be able to defer deletion of older versions of executable files until it is sure that no client is executing any of those versions. In either case, the cache coherence protocol will be more complex and less scalable.

Coda also uses whole-file caching. However, scalability is not the only motivation in this case. From the perspective of disconnected operation, whole-file caching also offers another important advantage: remote failures are only visible on `open` and `close` operations. In our opinion, the simplicity and robustness of this failure model outweigh the merits of partial-file caching schemes such as those of AFS-3, Echo, and MFS [3].

When caching is done at large granularity, considerable performance improvement can be obtained by the use of a specialized bulk data-transfer protocol. 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 bulk transfer protocols to be effective there has to be substantial spatial locality of reference within files. The presence of such locality has been confirmed by empirical observations of Unix systems. For example, Ousterhout *et al.*

[13] note that most files are read in their entirety after being opened.

Systems that use whole-file caching are most naturally suited to using bulk transfer protocols. Other systems exploit bulk transfer in varying degrees. AFS-3 transfers data in large chunks, typically 64 kilobyte in size. Sun NFS and Sprite do not use bulk transfer protocols, but use large datagrams, typically 8 kilobyte in size. It is likely that bulk transfer protocols will increase in importance as distributed file systems spread across networks of wider geographic area.

D. Token-Based Mutual Authentication

A social consequence of large scale is that the casual attitude toward security typical of closely knit distributed environments is no longer viable. The relative anonymity of users in a large system requires security to be maintained by enforcement rather than by good will. This, in turn, raises the question of who can be trusted to enforce security. Security in Andrew and Coda is predicated on the integrity of a relatively small number of servers, rather than the much larger number of clients.

Many small-scale distributed systems present a facade of security by using simple extensions of the mechanisms used in a time-sharing environment. For example, authentication is often implemented by sending a password in the clear to the server, which then validates it. Besides the obvious danger of sending passwords in the clear, this also has the drawback that the client is not certain of the identity of the server. Andrew and Coda, in contrast, perform full mutual authentication using a variant of the Needham and Schroeder private key authentication algorithm [11].

In AFS-1, AFS-2, and Coda, this function is integrated with the RPC mechanism. To establish a secure and authenticated RPC connection, a 3-phase handshake takes place between client and server. The client supplies a variable-length identifier and an encryption key for the handshake. The server provides a key lookup procedure and a procedure to be invoked on authentication failure. The latter allows the server to record and possibly notify an administrator of suspicious authentication failures. At the end of a successful authentication handshake the server is assured that the client possesses the correct key, while the client is assured that the server is capable of looking up his key. The use of randomized information in the handshake guards against replays by adversaries.

A naive use of the RPC handshake would require the user to supply his password every time a new connection had to be established. The obvious improvement of having the user type in his password once and storing it in the clear at the client is risky. The approach used in Andrew and Coda is to provide a level of indirection using *authentication tokens.* When a user logs in to a client, the password he types in is used as the key to establish a secure RPC connection to an authentication server. A pair of authentication tokens are then obtained for the user on this secure connection. These tokens are saved by the client and are used by it to establish secure RPC connections on behalf of the user to file servers. To bound the period during which lost tokens can cause damage, tokens expire after a finite time (typically 24 h).

Like a file server, an authentication server runs on physically secure hardware. To improve availability and to balance load, there are multiple instances of the authentication server. Only one instance accepts updates; the others are slaves and respond only to queries. To improve accountability, the master maintains an audit trail of changes to the authentication database.

For reasons of standardization, AFS-3 uses the Kerberos authentication system [26]. Kerberos provides functionality equivalent to the authentication mechanism described above, and resembles it in design.

E. Hierarchical Groups and Access Lists

Controlling access to data is substantially more complex in large-scale systems than it is in smaller systems. There is more data to protect and more users to make access control decisions about. This is an area in which the Unix file system model is seriously deficient. The Unix protection model was obtained by simplifying the Multics protection model to meet the needs of small time-sharing systems. Not surprisingly, the Unix model becomes inadequate when a system is scaled up. To enhance scalability Andrew and Coda organize their protection domains hierarchically and support a full-fledged access-list mechanism.

The protection domain is composed of *users* and *groups.* Membership in a group is inherited, and a user's privileges are the cumulative privileges of all the groups he or she belongs to, either directly or indirectly. New additions to a group G, automatically acquire all privileges granted to the groups to which G belongs. Conversely, when a user is deleted, it is only necessary to remove him from those groups in which he is explicitly named as a member. Inheritance of membership conceptually simplifies the maintenance and administration of the protection domain, a particularly attractive trait at large scale. At least two other systems, CMU-CFS [1] and Grapevine [22], have also used a hierarchical protection domain.

Andrew and Coda use an *access-list* mechanism for file protection. The total rights specified for a user are the union of the rights specified for him and 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. Although the real enforcement of protection is done on the basis of access lists, Venus superimposes an emulation of Unix protection semantics by honoring the owner component of the Unix *mode bits* on a file. The combination of access lists on directories and mode bits on files has proved to be an excellent compromise between protection at fine granularity, scalability, and Unix compatibility.

The ability to *rapidly revoke* access privileges is important in a large distributed system. Revocation is usually done by removing an individual from an access list. But that individual may be a direct or indirect member of one or more groups that give him or her rights on the object. The process of discovering all groups that the user should be removed from, performing the removal at the site of the master authentication server, and propagating it to all slaves may take a significant amount of time in a large distributed system. Andrew and Coda simplify rapid and selective revocation by allowing access lists to 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 thus decouple the problems of rapid revocation and propagation of group membership information.

The loss of *accountability* caused by the shared use of a pseudo-user id (such as "root" in Unix systems) by system administrators is a serious problem at large scale. Consequently, administrative privileges in Andrew and Coda are obtained by membership in a distinguished group named "System:Administrators." This improves accountability, since system administrators have to reveal their true identity during authentication.

F. First Versus Second-Class Replication

The use of two distinct mechanisms for high availability in Coda, server replication and disconnected operation, is an indirect consequence of Coda's desire to scale well. Systems such as Locus [29] that rely solely on server replication have poor scaling characteristics. Since disconnected operation is almost free, while server replication incurs additional hardware costs and protocol overhead, it is natural to ask why the latter mechanism is needed at all. The answer to this question depends critically on the very different assumptions made about clients and servers in Coda. These assumptions, in turn, reflect the usage and administrative characteristics of a large distributed system.

Clients are like appliances: they can be turned off at will and may be unattended for long periods of time. They have limited disk storage capacity, their software and hardware may be tampered with, and their owners may not be diligent about backing up the local disks. Servers, in contrast, have much greater disk capacity, are physically secure, and are carefully monitored and administered by a professional staff. It is therefore appropriate to distinguish between *first-class replicas* on servers and *second-class replicas* on clients (i.e., cached copies). First-class replicas are of higher quality—they are more persistent, widely known, secure, available, complete, and accurate. Second-class replicas, in contrast, are inferior along all these dimensions. Only by periodic revalidation with respect to a first-class replica can a second-class replica be useful.

The function of a cache coherence protocol is to combine the performance and scalability advantages of a second-class replica with the quality of a first-class replica. When disconnected the quality of the second-class replica may be degraded, because the first-class replica upon which it is contingent is inaccessible. The longer the duration of disconnection, the greater the potential for degradation. Whereas server replication preserves the quality of data in the face of failures, disconnected operation forsakes quality for availability. Hence server replication is important, because it reduces the frequency and duration of disconnected operation, which is properly viewed as a measure of last resort.

G. Data Aggregation

In a large system, considerations of *operability* and *system administration* assume major significance. To facilitate these functions, Andrew and Coda organize file system data into *volumes* [24]. A volume is a collection of files located on one server and forming a partial subtree of the Vice name space. Volumes are invisible to application programs and are only manipulated by system administrators. The aggregation of data provided by volumes reduces the apparent size of the system as perceived by operators and system administrators. Our operational experience in Andrew and Coda confirms the value of the volume abstraction in a large distributed file system.

Virtually all administrative functions in Andrew and Coda are done at the granularity of volumes. For example, volumes are the unit of read-only replication in Andrew, and read-write replication in Coda. Balancing of the available disk space and utilization on servers is accomplished by redistributing volumes across one or more servers. These modifications can be made during normal operation without disrupting service to users. Disk storage quotas are specified and enforced on individual volumes

Volumes also form the basis of the backup and restoration mechanism. To backup a volume, a read-only *clone* is first made, thus creating a frozen snapshot of the constituent files. Since cloning is an efficient operation, users rarely notice any loss of access to that volume. An asynchronous mechanism then transfers this clone to a staging machine from where it is dumped to tape. The clone is also made available on-line. This substantially reduces the number of restoration requests received by operators, since users can themselves undo recent deletions by copying data from the clone.

H. Decentralized Administration

A large distributed system is unwieldy to manage as a monolithic entity. For smooth and efficient operation, it is essential to delegate administrative responsibility along lines that parallel institutional boundaries. Such a system decomposition has to balance site autonomy with the desirable but conflicting goal of system-wide uniformity in human and programming interfaces. The *cell* mechanism of AFS-3 [30] is an example of a mechanism that provides this balance.

A cell corresponds to a completely autonomous Andrew system, with its own protection domain, authentication and file servers, and system administrators. A federation of cells can cooperate in presenting users with a uniform, seamless file name space. Although the presence of multiple protection domains complicates the security mechanisms in Andrew, Venus hides much of the complexity from users. For example, authentication tokens issued in a cell are only valid within that cell. To preserve transparency when accessing files from different cells, Venus maintains a collection of tokens for the cells of interest. A user is aware of the existence of cells only at the beginning of a session, when he or she authenticates himself to individual cells to obtain authentication tokens. After this initial step, Venus uses the appropriate tokens when establishing a secure connection to a file server.

I. Functional Specialization

When implementing a distributed realization of an interface originally defined for a single site, one often finds that scalability and exact interface emulation make conflicting demands. One way to improve scalability is to relax the accuracy with which the interface is emulated. This strategy is particularly attractive if there is a natural partitioning of applications based on their use of the interface. Each equivalence class of applications can then use a distributed realization of the interface tuned to its critical requirements.

In Andrew and Coda, propagating modifications only upon `close` operations violates strict Unix semantics, but is irrelevant to most Unix applications. The use of caching and bulk data transfer presume substantial temporal and spatial locality of file accesses. The use of an optimistic replication strategy in Coda is based on the assumption that sequential write sharing is relatively rare. But the assumptions on which these techniques are based usually fail to hold for databases. Poor locality, fine granularity of update and query, and frequent concurrent and sequential write-sharing are the norm rather than the exception in databases.

Rather than compromise scalability in an attempt to support databases, Andrew and Coda partition the problem into two orthogonal components—file access and database access—and only address the former. Support for database access has to be provided by a separate mechanism. This two-pronged strategy is in contrast to the unified strategies of time-sharing Unix file systems, where all accesses (from databases or otherwise) are supported on the same interface.

Functional specialization also characterizes the mechanism in Andrew for supporting personal computers (PC's) such as the IBM PC and Apple Macintosh. Such machines differ from full-fledged Andrew clients in that they do not run Unix, typically possess limited amounts of memory, and often do not possess a local disk. Caching of whole files, or large chunks of files, is not a viable design strategy for such machines. However, since a significant number of Andrew users also use PC's, we felt it essential to allow PC users to access Vice files. This functionality is provided by a mechanism called *PCServer* [15] that is orthogonal to the Andrew file system.

PCServer runs on an Andrew client and makes its file system appear to be a transparent extension of the file systems of a number of PC's. Since Vice files are transparently accessible from the client, they are also transparently accessible from the PC. The client thus acts as a surrogate for Vice. The protocol between PCServer and its clients is tuned to the capabilities of a PC. From the point of view of Venus, it appears as if the PC user had actually logged in at the client running PCServer. The decoupling provided by PCServer allows the Andrew file system to exploit techniques essential to good performance at large scale, without distorting its design to accommodate machines with limited hardware and software capability.

J. Heterogeneity

As a distributed system evolves it tends to grow more diverse. One factor contributing to diversity is the improvement in performance and decrease in cost of hardware over time. This makes it likely that the most economical hardware configurations will change over the period of growth of the system. Another source of heterogeneity is the use of different computer platforms for different applications. For example, the same individual may use a supercomputer for simulations, a Macintosh for document processing, a Unix workstation for program development, and a laptop IBM PC while traveling. Easy access to shared data across these diverse platforms would substantially improve usability.

Andrew did not set out to be a heterogeneous computing environment. Initial plans for it envisioned a single type of client, running one operating system, with the network constructed of a single type of physical media. Yet heterogeneity appeared early in its history and proliferated with time. Some of this heterogeneity is attributable to the decentralized administration typical of universities, but we are convinced that much of it is intrinsic to the growth and evolution of any distributed system.

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. The distributed file system community has gained some experience with heterogeneity. For example, Pinkerton *et al.* describe an experimental file system at Washington [14] that focuses on heterogeneity. TOPS [27] is a product offered by Sun Microsystems which allows shared-file access across the MS-DOS and Macintosh operating systems. PC-NFS, also from Sun, allows MS-DOS applications to access files on an NFS server. PCServer, described in the previous section, performs a similar function in the Andrew environment.

V. Design Principles for Scalability

The essence of the Andrew and Coda strategy is to decompose a large distributed system into a small nucleus that changes relatively slowly, and a much larger and less static periphery. From the perspectives of security and operability, the scale of the system appears to be that of the nucleus. But from the perspectives of performance and availability, a user at the periphery receives almost stand-alone service. It is the thesis of this paper that such a strategy is feasible and effective.

A consequence of this strategy is that clients and servers need to be *physically distinct* machines. This seemingly minor detail turns out to be critical. Without this dichotomy, one cannot make different security and administrative decisions about clients and servers, nor can one optimize their hardware and software configurations independently. Although the need to have physically distinct clients and servers is not a problem at large scale, it is an expensive proposition at small scale. It is therefore tempting to make the client-versus-server distinction only a *logical* one, so that the start-up cost of a small installation is low. Unfortunately, systems such as NFS and Locus that have chosen this approach have foundered on the rock of scalability. Growth in these systems is unwieldy, and none of them appears capable of growth to thousands of sites. One is therefore forced to conclude that the client–server distinction is a fundamental one from the perspective of scalability, and that a higher initial cost is the price one pays for a system that can grow gracefully.

Besides this high-level principle, we have also acquired more detailed insights about scalability in the course of building Andrew and Coda. We present these insights here as a collection of design principles:

- *Clients have the cycles to burn*

  Whenever there is a choice between performing an operation on a client and performing it on a server, it is preferable to pick the client. This will enhance the scalability of the design, since it lessens the need to increase central resources as clients are added.

  The only functions performed by servers in Andrew and Coda are those critical to the 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 accessible servers, 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 currently 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 usage properties*

  Knowledge about the use of real systems allows better design choices to be made. For example, 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, in a distributed file system design.

  Almost one-third of 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 observation that sequential write-sharing of user files is rare.
- *Minimize system-wide 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 in a consistent manner. The scalability of a design is enhanced if it rarely requires global information to be monitored or atomically updated.

  Clients in Andrew and Coda only monitor 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 heavy-weight mechanism) with caching to improve availability without losing scalability. Knowledge of a caching site is confined to those servers with callbacks for the caching site. Coda does not depend on knowledge of system-wide topology, nor does it incorporate any algorithms requiring system-wide election or commitment.

  Another instance of the application of this principle is the use of negative rights. More rapid revocation is possible by modifications to an access list at a single site rather than by a system-wide change to 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 clients, 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 client integrity is delegated to the owner of each client. Andrew and Coda rely on end-to-end encryption rather than physical link security.
- *Batch if possible*

  Grouping operations together can improve throughput (and hence scalability), although it is 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.

VI. Conclusion

The central message of this paper is that growth is an inevitable characteristic of successful and long-lived distributed systems. Designers should therefore prepare for growth *a priori*, rather than treating it as an afterthought. Our experience with Andrew and Coda has taught us much about building scalable distributed systems. We now have a collection of mechanisms that have been shown to enhance scalability, and a set of general principles to guide future design choices. But there is always the danger that system designers, like old generals, are fighting the last war. Each quantum increase in scale is likely to expose new ways in which the old tricks fail to work. It is with some trepidation, therefore, that we await the challenges posed by the next generation of large-scale distributed systems.

Acknowledgment

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 T. Anderson, S. Chutani, J. Howard, M. Kazar, S. Menees Nichols, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West, and E. Zayas. Contributions to the early design of Andrew were also made by D. Gifford and A. Spector.

Coda is being built in the School of Computer Science at Carnegie Mellon University. Contributors to Coda include J. Kistler, P. Kumar, H. Mashburn, B. Noble, M. Okasaki, M. Satyanarayanan, D. Steere, E. Siegel, and W. Smith.

The views and conclusion expressed in this paper are those of the author, and should not be interpreted as being those of the funding agencies or Carnegie Mellon University.

REFERENCES

[1] M. Accetta, G. Robertson, M. Satyanarayanan, and M. Thompson, "The design of a network-based central file system," Dept. Computer Sci., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-80-134, 1980.

[2] D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle connection," *Software Pract. Exper.*, vol. 12, pp. 1147–1162, 1982.

[3] M. Burrows, "Efficient data sharing," Ph.D. diss., Computer Lab., Univ. Cambridge, Dec. 1988.

[4] "VMS system software handbook," Digital Equipment Corp., Maynard, MA, 1985.

[5] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart, "Availability and consistency trade-offs in the Echo distributed file system," in *Proc. 2nd IEEE Workshop on Workstation Operating Syst.*, Sept. 1989.

[6] J. H. Howard *et al.*, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, Feb. 1988.

[7] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, Feb. 1992.

[8] P. H. Levine, "The Apollo DOMAIN distributed file system," in *NATO ASI Series: Theory and Practice of Distributed Operating Systems*, Y. Paker, J.-P. Banatre, and M. Bozyigit, Eds. New York: Springer-Verlag, 1987.

[9] J. H. Morris *et al.*, "Andrew: a distributed personal computing environment," *Commun. ACM*, vol. 29, no. 3, Mar. 1986.

[10] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: a distributed operating system for the 1990s," *IEEE Trans. Computer*, vol. 23, pp. 44–53, May 1990.

[11] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, Dec. 1978.

[12] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *ACM Trans. Comput. Syst.* vol. 6, no. 1, Feb. 1988.

[13] J. Ousterhout *et al.*, "Trace-driven analysis of the Unix 4.2 BSD file system," in *Proc. 10th ACM Symp. on Operating System Principles*, Dec. 1985.

[14] C. B. Pinkerton, E. D. Lazowska, D. Notkin, and J. Zahorjan, "A heterogeneous remote file system," Dept. Computer Sci., Univ. Washington, Seattle, Tech. Rep. 88-08-08, Aug. 1988.

[15] L. K. Raper, "The CMU PC Server project," Inform. Techn. Ctr., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-ITC-051, Feb. 1986.

[16] A. P. Rifkin *et al.*, "RFS architectural overview," in *Proc. Usenix Conference* (Atlanta, GA), 1986.

[17] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network filesystem," in *Proc. Usenix Conf.* (Portland), 1985.

[18] M. Satyanarayanan *et al.*, "The ITC distributed file system: principles and design," in *Proc. 10th ACM Symp. on Operating System Principles*, Dec. 1985.

[19] M. Satyanarayanan, "Integrating security in a large distributed system," *ACM Trans. Comput. Syst.*, vol. 7, no. 3, Aug. 1989.

[20] M. Satyanarayanan *et al.*, "Coda: a highly available file system for a distributed workstation environment," *IEEE Trans. Computers*, vol. 39, no. 4, Apr. 1990.

[21] M. Satyanarayanan, "Scalable, secure, and highly available distributed file access," *IEEE Computers*, vol. 23, no. 5, May 1990.

[22] M. D. Schroeder, A. D. Birrell, and R. M. Needham, "Experience with Grapevine: the growth of a distributed system," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 3–23, Feb. 1984.

[23] M. D. Schroeder, D. K. Gifford, and R. M. Needham, "A caching file system for a programmer's workstation," in *Proc. 10th Symp. on Operating System Principles*, Dec. 1985.

[24] R. N. Sidebotham, "Volumes: the Andrew file system data structuring primitive," in *Proc. Eur. Unix User Group Conf.*, Aug. 1986 (also available as Information Techn. Ctr., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-ITC-053).

[25] D. C. Steere, J. J. Kistler, and M. Satyanarayanan, "Efficient user-level file cache management on the Sun Vnode interface," in *Proc. Usenix Conf.* (Anaheim, CA), June 1990.

[26] J. G. Steiner, C. Neumann, and J. I. Schiller, "Kerberos: an authentication service for open network systems," in *Proc. Usenix Conf.* (Dallas, TX), Feb. 1988.

[27] G. Stroud, "Introduction to TOPS," *Sun Techn.*, vol. 1, no. 2, Spring 1988.

[28] D. B. Terry, "Caching hints in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-13, Jan. 1987.

[29] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS distributed operating system," in *Proc. 9th Symp. on Operating System Principles*, Oct. 1983.

[30] E. R. Zayas and C. F. Everhart, "Design and specification of the cellular Andrew environment," Inform. Techn. Ctr., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-ITC-070, June 1988.

Mahadev Satyanarayanan (S'80–M'82) received the Bachelor's degree in electrical engineering and the Master's degree in computer science from the Indian Institute of Technology, Madras. He received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1983.

He is an Associate Professor of Computer Science at Carnegie Mellon University, where his research addresses the general problem of sharing access to information in large-scale distributed systems. In his current work on the Coda File System, he is investigating techniques to provide high availability without significant loss of performance or usability. An important aspect of this work is providing distributed file access to portable computers that may operate disconnected for significant periods of time. Earlier, he was one of the principal architects and implementors of the Andrew File System, a location-transparent distributed file system that addressed issues of scale and security. 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. He has been a Consultant to industry and government.

Dr. Satyanarayanan is a member of the ACM and Sigma Xi. He was made a Presidential Young Investigator by the National Science Foundation in 1987.