On the Influence of Scale in a Distributed System

M. Satyanarayanan

Department of Computer Science
Carnegie Mellon University

Abstract

Scale should be recognized as a primary factor influencing the architecture and implementation of distributed systems. This paper uses Andrew, a distributed environment at Carnegie Mellon University, to validate this proposition. The design of Andrew is dominated by considerations of performance, operability and security. Caching of information and placing trust in as few machines as possible emerge as two general principles that enhance scalability. The separation of concerns made possible by specialized mechanisms is also valuable. Heterogeneity is a natural consequence of growth and anticipating it in the initial stages of system design is important. A location transparent shared file system considerably enhances the usability of a distributed environment.

1. Introduction

Software engineering focuses on programming in the large, recognizing that scale 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 are acceptable in small distributed systems fail to be adequate in the context of a larger system. Scale is thus a primary factor that guides the design and implementation of large distributed systems.

We substantiate this view using Andrew, a large distributed environment built at Carnegie Mellon University (CMU). When mature, it is expected to encompass over 5000 workstations spanning the CMU campus. At the time this paper was written, in mid-1987, there were about 400 workstations and 16 servers. There were over 3500 registered users of the system, of whom over 1000 used Andrew regularly. Approximately 6000 Mbytes of shared data were stored in the system. Although Andrew is not the sole computing facility at CMU, it is used as the primary computational environment of many courses and research projects.

Scale affects a distributed system in many ways:

- Considerations of performance and operability dominate the design.
- Security becomes a serious concern.
- Functionally specialized mechanisms rather than general-purpose solutions become more attractive.
- The system is likely to be composed of diverse elements, rather than a single homogeneous set of elements.

We discuss these and other related issues in this paper. Section 2 gives a brief overview of Andrew and mentions the systems we use to contrast its design. Sections 3, 4 and 5 discuss specific aspects of Andrew that have been significantly influenced by scale. Section 6 concludes the paper with a summary of its key points.

2. Distributed Systems

The phrase ''distributed system'' has been used in the literature to describe a variety of configurations, ranging from small multiprocessors to a nation-wide network such as the Arpanet. In this paper we use the term to mean a system composed of nodes that are capable of autonomous computation, but whose full potential is realized only in the presence of other nodes. A collection of workstations sharing file servers and print servers is a common instance of such a distributed system.

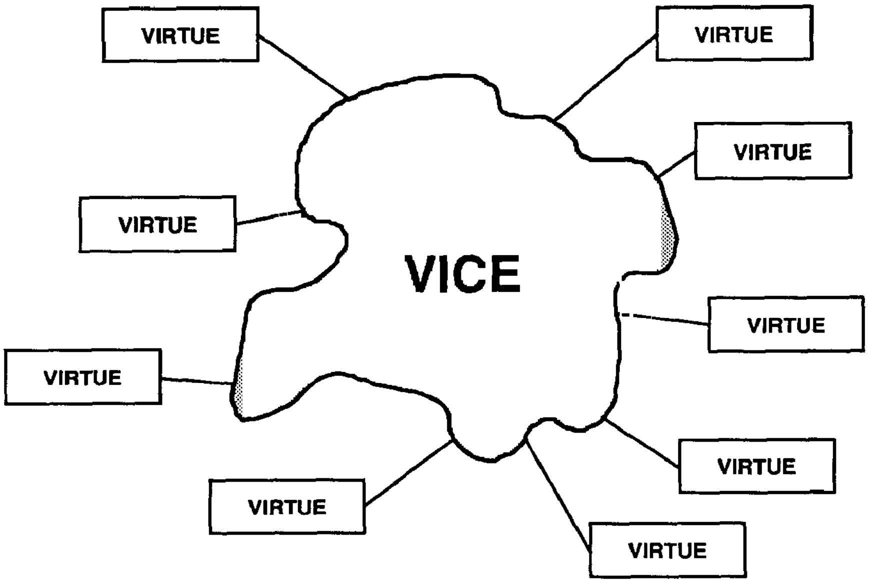
Andrew, the focus of this paper, combines a sophisticated user interface feasible only in personal computing with the data sharing simplicity of timesharing. This synthesis is achieved by close cooperation between two kinds of components, *Vice* and *Virtue*, shown in Figure 1. A Virtue workstation provides the power and capability of a dedicated personal computer, while Vice provides support for the timesharing abstraction.

Although Vice is shown as a single logical entity in Figure 1, it is actually composed of a collection of servers and a complex local area network. This network spans the entire CMU

The author was supported in the writing of this paper by the National Science Foundation (Contract CCR-8657907), by an IBM Faculty Development Award, and by the Defense Advanced Research Projects Agency (Contract F33615-87-C-1499). The views and conclusions in this document are those of the author and should not be interpreted as representing the official policies of the funding agencies or Carnegie Mellon University.

0270-5257/88/0000/0010$01.00 © 1988 IEEE

Recommended by: A. Wasserman and J. Mylopoulos



The amoeba-like structure in the center is a collection of networks and servers that constitute Vice. Virtue is typically a workstation, but can also be a mainframe.

Figure 1: Vice and Virtue

campus and is composed of Ethernet and IBM Token Ring segments interconnected by optic fiber links and active elements called *Routers*. Figure 2 shows the details of this network. Each Virtue workstation runs the Unix 4.2BSD operating system[1], and is thus an autonomous computational node. A distributed file system that spans all workstations is the primary data-sharing mechanism in Andrew. Section 3 describes the details of this file system.

To elucidate certain aspects of Andrew, this paper refers to a number of other distributed systems. One of these systems, *Grapevine* [17], is a distributed registry developed at Xerox PARC. Its primary function is to serve as a repository of naming information for the electronic mail system, and is similar to Andrew in that it pays attention to scaling issues. The following are the other distributed systems we consider:

Locus [10, 23] is a distributed operating system built at UCLA with the primary design goal of providing total network transparency.

Sun NFS [21] is similar to Andrew in that it is composed of a collection of Unix workstations sharing common files. It does not, however, make a fundamental distinction between clients and servers; every workstation can export its local file name space and thus become a server. Diskless operation is common in this environment.

Cedar [18] is a distributed environment developed at Xerox PARC as a research vehicle for ideas in programming environments and languages. Consistent with this goal, its design pays considerable attention to support for program development.

V [2] was built at Stanford University and uses a high performance interprocess communication mechanism to attain network transparency. It was one of the earliest systems to demonstrate the viability of diskless operation.

Unix United [1] is a mechanism developed at the University of Newcastle upon Tyne to allow a collection of autonomous Unix systems to share their file systems.

Vax/VMS [3] is the standard operating system for the Digital Equipment Corporation's series of Vax computers, and provides support for remote access of files.

Although these systems differ considerably from each other, they share the common property that they were not specifically designed for use in a large distributed environment.

3. Shared File System

The primary data sharing mechanism in Andrew is a distributed file system that spans all Virtue workstations. In this section we focus on a few aspects of its design that most clearly reveal the effects of large scale. More comprehensive descriptions can be found elsewhere [13, 4].

[1] Unix is a trademark of AT&T. In the rest of this paper the term "Unix" will refer to the 4.2BSD version.

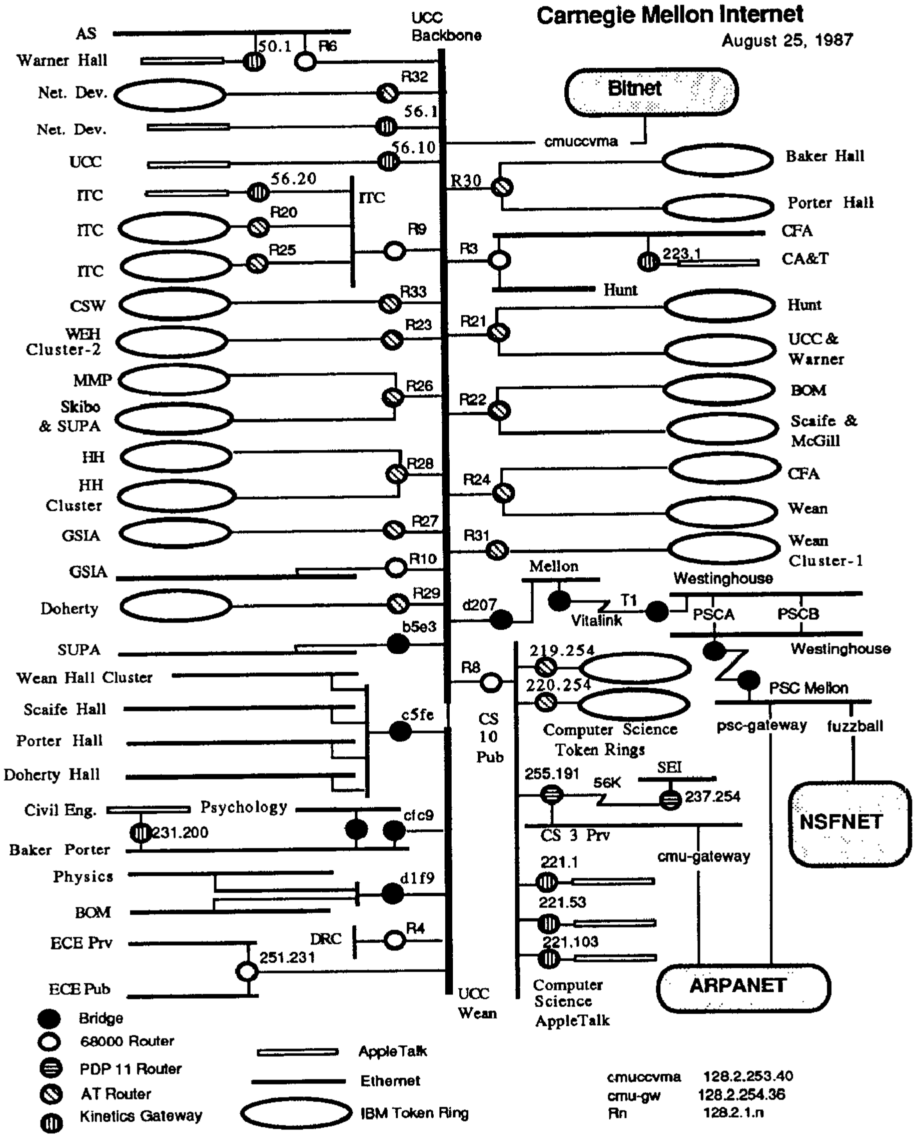


Figure 2: CMU Network

3.1. Naming and Location

A fundamental issue is how files are named and located in a distributed environment. In many file systems, users explicitly identify the site at which a file is located. Constructs such as "*/../machine/localpath*" (as in Unix United), "*/server/localpath*" (as in Cedar), and "*machine::device:localpath*" (as in Vax/VMS) contain location information in the file name. The embedding of physical 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.

Some systems, such as Sun NFS, provide a naming scheme that does not include explicit machine identification. Instead, an association is established between a pathname on a remote machine and a local name. This association is performed using the *Mount* mechanism in Unix when a machine is initialized and remains in effect until the machine is reinitialized. Although pathnames appear to be free of location information, this is merely an illusion. If a file is moved from one server to another, it will not be accessible under its original name unless the client is reinitialized.

Andrew, in contrast, offers true *Location Transparency*. The name of a file contains no location information. This information is dynamically obtained during the normal operation of a client. Moving a file from one server to another is an operation that is totally transparent to users. The addition or removal of servers, and the redistribution of files on them does not hinder normal usage of Andrew. Locus also provides location transparency, but does so primarily for ease of use rather than scalability.

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. Usage experience with Andrew demonstrates that a fully dynamic location mechanism is essential in a large distributed environment.

A user-level process, called *Venus*, on each Andrew workstation intercepts and processes remote file system calls. It is Venus that locates files and communicates with individual servers. To maximize the number of clients that can be supported by a server, as much of the work as possible is performed by Venus rather than by Vice. Only functions essential to the integrity, availability or security of the file system are retained in Vice. The servers are organized as a loose confederacy, with minimal communication among themselves. This file system architecture was motivated primarily by considerations of scale.

3.2. Caching and Whole-File Transfer

Although distributed file systems such as Locus, Sun NFS, and Unix United vary considerably in detail, they all share one fundamental property: *the data in a file is not fetched en masse; instead, the remote site potentially participates in each individual read and write operation.* Buffering and read-ahead are employed by some of the systems to improve performance, but the remote site is still conceptually involved in every I/O operation. We call this property *Remote Open*, since it is reminiscent of the situation where a file is actually opened on the remote site rather than the local site.

Andrew workstations, in contrast, cache entire files from Vice and store modified copies of files back on the servers they came from. Vice is contacted by Venus only when a file is opened or closed; reading and writing of individual bytes of a file are performed directly on the cached copy, bypassing Venus. The superior scaling properties of this design have been demonstrated by controlled experiments comparing it to a remote-open file system [4].

The caching of entire files on local disks in the Andrew was motivated primarily by considerations of scale:

- Locality of file references by typical users makes caching attractive: server load and network traffic are reduced.
- A whole-file transfer approach contacts servers only on opens and closes. Read and write operations, which are far more numerous, are transparent to servers and cause no network traffic.
- The study by Ousterhout et al [9] has shown that most files in a 4.2BSD environment are read in their entirety. Whole-file transfer exploits this property by allowing the use of efficient bulk data transfer protocols.
- Disk caches retain their entries across reboots, a surprisingly frequent event in workstation environments. Since few of the files accessed by a typical user are likely to be modified elsewhere in the system, the amount of data fetched after a reboot is usually small.
- Finally, caching of entire files simplifies cache management. Venus only has to keep track of the files in its cache, not of their individual pages.

Whole-file transfer is thus a concession to scale, motivated by the desire to provide good performance. This approach does have disadvantages, such as the need for local disks and the inability to handle files larger than the available cache space. Fortunately these have not proved to be serious drawbacks in our environment.

Cedar also employs caching of entire files as its remote access mechanism. However, it does this not for reasons of scale, but because its file system semantics are oriented specifically toward a program development environment. Files are treated as immutable objects, with new versions of files being created rather than the original versions being overwritten. Modifications to a file on a workstation are not automatically reflected on the servers. An explicit user action has to be performed for the changes to be propagated.

3.3. Data Aggregation

As the scale of a system grows operability assumes major significance. A typical distributed file system, such as Sun NFS, maps an entire subtree on a server on to a leaf node in the local name space of a client. Andrew, on the other hand, uses a data structuring primitive called a *Volume* [19] to enhance operability. A *Volume* is a collection of files located on one server and forming a partial subtree of the Vice name space. The file system hierarchy is constructed out of volumes, but is orthogonal to it.

Balancing of the available disk space and utilization on servers is accomplished by redistributing volumes among the available disk partitions on one or more servers. A volume may be used, even for update, while it is being moved. Storage quotas are implemented on a per volume basis. Each user of the system is assigned a volume and each volume is assigned a quota.

Read-only replication is supported at the granularity of an entire volume. A read-only *Clone* of a volume can be created and propagated to replication sites to improve availability and balance server load. Read-only volumes are also valuable in system administration since they form the basis of an orderly release process for system software. It is easy to back out a new release in the event of an unanticipated problem with it.

Volumes also form the basis of the backup and restoration mechanism in Andrew. To backup a volume, a read-only clone is first made, thus creating a frozen snapshot of those 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.

Positive operational experience with volumes lead us to conclude that the volume abstraction, or something similar to it, is indispensible in a large distributed file system.

3.4. Implementation

Besides affecting the high-level architecture of the Andrew file system, scale has also affected many lower-level aspects of its design and implementation. In the worst case, each server has to be able to service hundreds or thousands of users concurrently. Experience with an initial prototype of Andrew provided the motivation for many of the decisions described in this section.

A common strategy in Unix implementations of the client-server model is to dedicate a single Unix server process to each client. Unfortunately such an implementation does not scale well due to excessive context switching and paging overheads. In addition, the server processes cannot cache critical shared information since Unix processes do not share virtual memory. Therefore, Andrew uses a single Unix process to service all clients of a server. Since multiple threads of control provide a convenient programming abstraction, a user-level mechanism is used to support multiple non-preemptive *Lightweight Processes* (LWPs) within one Unix process.

Both Venus and server code run as user-level Unix processes. Communication between them is based on the RPC paradigm and uses an independently-optimized protocol for the transfer of bulk data. This RPC implementation is entirely outside the kernel and is fully integrated with the LWP mechanism. A call can be made by an LWP even when one or more of its siblings is blocked on an outstanding RPC.

Caching, the key to Andrew's ability to scale well, requires a mechanism to validate data in the system. Rather than checking with a server on each open, Venus assumes that cache entries are valid unless otherwise notified. When a workstation caches a file or directory, the server promises to notify it before allowing a modification by any other workstation. This promise, called a *Callback*, reduces the number of cache validation requests received by servers although it does complicate the implementation. In spite of the complications, callback has proved critical to the scalability of Andrew. Systems like Sun NFS, which validate cache entries when they are used, exhibit considerably poorer performance under high load [4].

Callback also makes it feasible for workstations to translate variable-length pathnames to unique fixed length *File Identifiers* (*Fid*), by which they are identified on servers. In the absence of callback, the lookup of every component of a pathname would generate a cache validation request. The ability to perform name translations on workstations rather than servers contributes to scalability, since such translations are among the most heavily used and time consuming system functions in Unix. Physical data storage on the servers is also organized in a manner that avoids name translation.

4. Security

A consequence of large scale is that the casual attitude towards 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 the good will of the user community. Most distributed systems present a facade of security by using simple extensions of the mechanisms used in a timesharing environment. For example, network traffic between clients and servers is sent in the clear, thereby allowing a malicious individual to eavesdrop. 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. The design of Andrew pays serious attention to these and related security issues, while ensuring that the mechanisms for security do not inhibit legitimate use of the system [15].

A fundamental assumption pertains to the question of who enforces security in Andrew. Rather than trusting thousands of workstations, security in Andrew is predicated on the integrity of the much smaller number of Vice servers. These servers are located in physically secure rooms, are accessible only to trusted operators, and run software that is above suspicion. No user software is ever run on servers. Workstations may be owned privately or located in public areas. Andrew assumes that owners may modify both the hardware and software on their workstations in arbitrary ways.

4.1. Protection Domain

In Andrew, the protection domain 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 group is a set of other groups and users, associated with a user called its *Owner*.

A group named "System:Administrators" is distinguished. Membership in this group endows special administrative privileges, including unrestricted access to any file in the system. The use of a group "System:Administrators" 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. This is particularly important in view of the scale of Andrew. Another advantage is that revocation of special privileges can be done by modifying group membership rather than by changing a password and communicating it securely to the users who are administrators.

Membership in a group is inherited. The privileges that a user has at any time are the cumulative privileges of all the groups he 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. The scale of Andrew makes this an important advantage. Hierarchical group structures have also been shown to be useful in Grapevine.

A common practice in many timesharing and distributed systems is to create a single entry in the protection domain to stand for a collection of users. Such a collective entry, often referred to as a "group account" or a "project account," may be used for a variety of reasons. It may be the case that excessive administrative overheads are involved in adding new users. Sometimes the identities of all collaborating users may not be known a priori. A rigid protection mechanism may make it simpler to allow appropriate access to a single pseudo-user than to a number of users.

The use of collective entries is detrimental to security in a large environment like Andrew. Such entries exacerbate the already-difficult problem of accountability in a large distributed system. In Andrew, the hierarchical organization of the protection domain and the use of an access list mechanism for file protection makes specification of protection policies particularly simple. This reduces the motivation for collective entries.

4.2. Authentication

Authentication is the indisputable establishment of identities between two mutually suspicious parties in the face of adversaries with malicious intent. In Andrew, the two parties are a user at a Virtue workstation and a Vice server, while the adversaries are eavesdroppers on the network or modified network hardware that alters the data being transmitted.

The RPC mechanism used in Andrew provides support for secure, authenticated communication between mutually suspicious parties. To establish a secure and authentication 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.

Andrew uses a two-step authentication scheme that is closely integrated into the RPC mechanism. When a user logs in to a workstation, 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 passed to Venus and are saved by it to establish secure RPC connections on behalf of the user to file servers.

The level of indirection provided by tokens yields transparency and robustness. These criteria are important from the point of view of usability as well as scalability. Venus is able to establish secure connections to file servers without users having to supply their password each time a server is contacted. Passwords do not have to be stored in the clear on workstations. Since tokens expire after a finite time (typically 24 hours) there is a bound on the period during which lost tokens can cause damage. Finally, system programs other than Venus can use the tokens to authenticate themselves to Vice without user intervention.

The authentication server runs on a trusted Vice machine. For robustness, there are multiple instances of the authentication server. These are slaves and respond only to queries. Only one server, the master, accepts updates. Changes are propagated to slaves by the master. Server performance is considerably improved by maintaining a write-through cache copy of the entire database in the server's virtual memory. A modification to the database immediately overwrites cached information. The copy on disk is not, however, overwritten. Rather, the change is appended to an audit trail maintained in the authentication database. This improves accountability in a system administered by multiple individuals.

4.3. File System Protection

As the custodian of shared information in Andrew, Vice enforces the protection policies specified by users. The scale, character and periodic change in the composition of the user community in a university necessitate a protection mechanism that is simple to use yet allows complex policies to be expressed. A further consequence of these factors is that revocation of access privileges is an important and common operation.

On account of these considerations Andrew uses an *Access List* mechanism. The total rights specified for a user are the union of all the rights collectively specified for him and for all the groups of which he is a direct or indirect member. In conjunction with the ability to specify groups hierarchically, this accumulation of rights allows the large and diverse user community served by Andrew to specify flexible protection policies.

An access list can specify *Negative Rights*. An entry in a negative rights list indicates *denial* of the specified rights, with denial overriding possession in case of conflict. Negative rights are primarily a means of rapidly and selectively revoking access to sensitive objects. Although such revocation is more properly done by changes to the protection domain, the changes may take time to propagate in a large distributed system. Negative rights can reduce the window of vulnerability, since changes to access lists are effective immediately. They effectively decouple the problems of rapid revocation and propagation of information in a large distributed system.

Vice associates an access list with each directory. The access list applies to all files in the directory, thus giving them uniform protection status. The primary reason for this design decision is conceptual simplicity. Users have, at all times, a rough mental picture of the protection state of the files they access. In a large system, the reduction in state obtained by associating protection with directories rather than files is considerable.

In Unix, a file has 9 *Mode* bits associated with it. These mode bits are, in effect, a 3-entry access list specifying whether or not the owner of the file, a single specific group of users, and everyone else can read, write or execute the file. Venus provides a close emulation of Unix protection semantics. The Vice access list check described in the previous paragraph performs the real enforcement of protection. In addition, the three owner bits of the file mode indicate readability, writability or executability. These bits, which now 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 providing protection at fine granularity, retaining conceptual simplicity and retaining Unix compatibility.

5. Other Aspects

Certain insights about large distributed systems have emerged from experience with Andrew. First, in designing such a system it is beneficial to simplify the overall design by the use of specialized mechanisms to handle important special cases. Second, the design will have to contend with heterogeneity at many levels of abstraction. Third, the usability of the system is considerably enhanced by the presence of a location-transparent shared file system. We discuss these issues in the rest of this section.

5.1. Functional Specialization

The design of the Andrew file system uses whole-file transfer and caching as key techniques to deal with scale. The use of these techniques was motivated by observations of real systems that indicate that files tend to be small, that fine-grain write sharing is relatively rare, and that most files are read in their entirety. However, databases are an important class of applications that violate these assumptions. Large aggregation of data, fine granularity of update and query, and selective examination of data are the norm rather than the exception in databases.

The Andrew file system is thus inadequate to support databases. This is in contrast to timesharing systems where a database is often implemented on top of the underlying file system [20]. Although it would have been conceptually attractive to have a data access mechanism that supported normal file access as well as database access, we felt that the scale of Andrew would render such a mechanism inefficient. Instead we chose to partition the problem into two orthogonal components, file access and database access. File access is provided by the distributed file system described in Section 3, while database access is provided by *Scylla* [16].

Scylla integrates a remote procedure call mechanism, RPC2 [14], with a vendor-supported timesharing database system, Informix [6, 5]. A user at a workstation interacts with a client process that communicates via RPC with a Scylla server process on a standalone Unix machine. The local file system on the server contains the shared database. Concurrency control, data access and error recovery are performed on the server, while all user-interface processing is done on the client. Scylla thus occupies a point in the design space between fully distributed databases and single-site databases. Scylla has just become operational and our initial experience with it is favorable. Future versions of it may use caching of data at clients to enhance scalability.

Functional specialization also characterizes the mechanism in Andrew for supporting personal computers (*PCs*) such as the IBM PC and the Apple Macintosh. Such machines differ from full-fledged Andrew workstations 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 is not a viable design strategy for such machines. However, since a significant number of Andrew users also use PCs, we felt it essential to allow PC users to access Vice files. This functionality is provided by a mechanism called *PCServer* [11] that is orthogonal to the Andrew file system.

PCServer runs on an Andrew workstation and makes its file system appear to be a transparent extension of the file systems of a number of PCs. Since Vice files are transparently accessible from the workstation, they are also transparently accessible from the PC. The workstation 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 workstation 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 capability.

5.2. Heterogeneity

Andrew did not set out to be a heterogeneous computing environment. Initial plans for it envisioned a single type of workstation, running one operating system, with the network constructed of a single type of physical media. In its present form, however, Andrew is far from being completely homogeneous. This is partially due to the fact that Andrew was built in a university environment, where administrative control is decentralized. But we also believe that certain aspects of its evolution are typical of any distributed system that grows significantly over time

One reason for heterogeneity is that a distributed system becomes an increasingly valuable resource as it grows in size and stores larger amounts of shared data. There is then considerable incentive and pressure to allow users who are currently outside the scope of the system to participate in the use of its resources. Such users are likely to be using hardware and software that are different from the standard in the distributed system. The PCServer solution, described in Section 5.1, and the *Common User Interface (CUI)* component of the mail and bulletin board system in Andrew [12] were motivated by precisely these considerations.

A second source of heterogeneity is the improvement in performance and decrease in cost of hardware over time. This makes it likely that the most effective hardware configurations will change over the period of growth of the system. Diversity is inevitable, unless one is willing to forego *a priori* all such improvements, or is willing to bear the cost of total replacement of existing equipment at each stage of enhancement. In Andrew, workstations span a variety of types such as Sun2s, Sun3s, DEC MicroVaxes and IBM RTs.

A third reason for heterogeneity is the need to use existing hardware that is not identical. This is almost always for cost considerations and is likely to be particularly true for network hardware. A substantial fraction of the cost of a network is the labor involved in laying the cables. There is considerable cost benefit in growing a network by using an existing network rather than by laying a new network to enforce homogeneity. The complexity of the campus network at CMU, as shown in Figure 1, is partly due to this reason.

Andrew addresses heterogeneity in a number of different ways. Customized mechanisms, such as PCServer, are one solution. By using a widely-supported version of the Unix operating system, and by implementing most of Andrew outside the kernel, the process of porting Andrew to new workstations is simplified. By using the Darpa IP/UDP network protocols as standard, and by building a highly portable RPC package on top of them, it has been possible for Andrew software to ignore diversity in network media. The extensive use of symbolic links to structure the local name space of workstations addresses the issue of heterogeneity in workstations. For instance, the local pathname "/usr/andrew/bin" is a symbolic link to "/cmu/unix/vax/usr/andrew/bin" on a Vax and to "/cmu/unix/sun/usr/andrew/bin" on a Sun workstation. This simplifies the embedding of pathnames in scripts and application programs.

Designs that ignore heterogeneity often have hidden assumptions pertaining to scalability. For example, Sun NFS uses large logical packet sizes and depends on low-level packet fragmentation on Ethernet to achieve performance in its network protocols. This works well on a single Ethernet cable, but performs poorly in a complex network because of limitations of the routing elements. As another example, the V system depends on multicast support in the network media and interfaces to support parallel communication. While this functionality is available on Ethernet, it is not necessarily available on all segments of a large heterogeneous network. Both these examples describe situations where the technical decisions are defensible in the context in which they were made, but which render growth difficult.

Although the details of the evolution of distributed systems may vary, the underlying causes of heterogeneity that we have discussed here are pervasive. Recognizing this at the outset is likely to produce a design that is better able to cope with growth.

5.3. Usability

Experience with Andrew confirms that a location-transparent file system is a key element in making a large distributed system usable. The model of a giant file system that appears identical from all Andrew workstations is one that even naive users comprehend and use effectively. Most users, in the course of a single day, use more than one workstation without any concern about the accessibility or consistency of their files. As discussed in Section 3.1, Andrew also insulates users from operational changes in the file system.

The shared file system also simplifies the implementation of a number of higher-level mechanisms. In most cases, an application that works in a timesharing Unix environment can be used in Andrew with little or no change. Source code control, for instance, is done using the standard *Revision Control System* (RCS) software [22]. Distribution of new system software is trivial. Installing a new version of a program in the shared file system invalidates all cached copies; each workstation will automatically fetch the new version when it is next used. Electronic mail is implemented by each user having a mailbox directory into which any user can insert new files, but which only the owner can manipulate in any other way. Bulletin boards are implemented as directories into which any user can insert new files and read existing files. *Rem*, a widely-used program that makes idle workstations available for parallel computation [8], also relies heavily on the fact that all the workstations share a file system.

Unfortunately the timesharing file system abstraction breaks down when a server or network failure occurs. The underlying complexity of the distributed environment then becomes visible. Each application program that needs to be robust, such as electronic mail, has to include mechanisms to recover from file system failures. The change that would most substantially improve the usability of Andrew would be a distributed file system that completely masked failures from users and application programs. It is still an open question whether this goal is achievable in conjunction with good performance and scalability.

6. Conclusion

In this paper we have examined the design of Andrew and contrasted specific aspects of it with a variety of other distributed systems. Andrew differs in many unique ways from these systems. The differences are a logical consequence of the scale of Andrew, and are not merely due to the subjective preferences of its designers.

Caching of information and placing trust in as few machines as possible are two general principles that enhance scalability. The separation of concerns made possible by specialized mechanisms is also valuable. Heterogeneity is a natural consequence of growth and anticipating this in the initial stages of system design is important. A location transparent shared file system considerably enhances the usability of a distributed environment.

It is not the intent of our discussion to prove that the specific design details of Andrew are optimal. Further growth may, in fact, reveal inadequacies in its mechanisms. Rather, Andrew is being used here as a case study to reveal the extent to which considerations of scale influence many levels of design and implementation. The thesis of this paper is that scale has to be recognized as a fundamental factor in the design of a distributed system. A system in which scale is treated as an afterthought is not likely to be usable, maintainable or efficient when the number of nodes increases beyond a relatively low threshold.

Acknowledgments

Andrew is a joint project of Carnegie Mellon University and the IBM Corporation, and was built at the Information Technology Center. The entire staff of the ITC has been instrumental in the success of Andrew. The file system and security mechanisms described in this paper were designed and implemented by a team whose membership over time has included John Howard, Michael Kazar, David King, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, Michael West, and Edward Zayas. PCServer was built by Larry Raper. Jonathan Rosenberg, Craig Everhart and Nathaniel Borenstein designed and implemented the mail system referred to in the paper. Bradley White and Stephen Hecht built Scylla. Although the Virtue user-interface software has not been discussed in this paper, it is a substantial and important component of Andrew. A survey paper [7] presents an overview of this aspect of the system.

James Kistler provided helpful comments that improved the quality of this paper. Figure 2 was provided by Thomas Holodnik.

References

1. Brownbridge, D.R., Marshall, L.F. and Randell, B. "The Newcastle Connection". *Software Practice and Experience 12* (1982), 1147-1162.

2. Cheriton, D.R. and Zwaenepoel, W. The Distributed V Kernel and its Performance for Diskless Workstations. Proceedings of the Ninth Symposium on Operating System Principles, October, 1983.

3. *VMS System Software Handbook.* Digital Equipment Corporation, Maynard, Mass, 1985.

4. Howard, J.H., Kazar, M.J., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N. and West, M.J. Scale and Performance in a Distributed File System. Proceedings of the 11th ACM Symposium on Operating System Principles, November, 1987.

5. *Informix-SQL Reference Manual.* Informix Software, Inc., 1986.

6. *Informix-SQL User Guide.* Informix Software, Inc., 1986.

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

8. Nichols, D.A. Using Idle Workstations in a Shared Computing Environment. Proceedings of the 11th ACM Symposium on Operating System Principles, November, 1987.

9. Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M. and Thompson, J. A Trace-Driven Analysis of the Unix 4.2 BSD File System. Proceedings of the 10th ACM Symposium on Operating System Principles, December, 1985.

10. Popek, G.J. and Walker, B.J.. *The LOCUS Distributed System Architecture.* The MIT Press, 1985.

11. Raper, L.K. The CMU PC Server Project. Tech. Rept. CMU-ITC-051, Information Technology Center, Carnegie Mellon University, February, 1986.

12. Rosenberg, J., Everhart, C.F. and Borenstein, N.S. An Overview of the Andrew Message System. Proceedings of the ACM Sigcomm '87 Workshop, Stowe, Vermont, August, 1987.

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

14. Satyanarayanan, M. RPC2 User Manual. Tech. Rept. CMU-ITC-84-038, Information Technology Center, Carnegie Mellon University, 1986 (revised).

15. Satyanarayanan, M. Integrating Security in a Large Distributed Environment. Department of Computer Science, Carnegie Mellon University, 1987.

16. Satyanarayanan, M., White, B.W. and Hecht, S.A. Workstation Access to Databases (in preparation). Department of Computer Science, Carnegie Mellon University, 1987.

17. Schroeder, M.D:, Birrell, A.D. and Needham, R.M. "Experience with Grapevine: The Growth of a Distributed System". *ACM Transactions on Computer Systems 2*, 1 (February 1984), 3-23.

18. Schroeder, M.D., Gifford, D.K. and Needham, R.M. A Caching File System for a Programmer's Workstation. Proceedings of the Tenth Symposium on Operating System Principles, December, 1985.

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

20. Stonebraker, M, Wong, E., Kreps, P. and Held, G. "The Design and Implementation of INGRES". *ACM Transactions on Database Systems 1*, 3 (September 1976), 189-222.

21. *Networking on the SUN Workstation.* Sun Microsystems, Inc., 1986.

22. Tichy, W.F. "RCS -- A System for Version Control". *Software Practice and Experience 15*, 7 (July 1985), 637-654.

23. Walker, B., Popek, G., English, R., Kline, C. and Thiel, G. The LOCUS Distributed Operating System. Proceedings of the Ninth Symposium on Operating System Principles, October, 1983.