Scale and Performance in a Distributed File System

JOHN H. HOWARD, MICHAEL L. KAZAR, SHERRI G. MENEES, DAVID A. NICHOLS, M. SATYANARAYANAN, ROBERT N. SIDEBOTHAM, and MICHAEL J. WEST

Carnegie Mellon University

The Andrew File System is a location-transparent distributed file system that will eventually span more than 5000 workstations at Carnegie Mellon University. Large scale affects performance and complicates system operation. In this paper we present observations of a prototype implementation, motivate changes in the areas of cache validation, server process structure, name translation, and low-level storage representation, and quantitatively demonstrate Andrew's ability to scale gracefully. We establish the importance of whole-file transfer and caching in Andrew by comparing its performance with that of Sun Microsystem's NFS file system. We also show how the aggregation of files into volumes improves the operability of the system.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.8 [**Operating Systems**]: Performance—*measurements*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Andrew, caching, operability, scalability, Venus, Vice, Volumes, whole file transfer

1. INTRODUCTION

Andrew is a distributed computing environment that has been under development at Carnegie Mellon University since 1983. A comprehensive overview of the system has been presented by Morris et al. [3]. The characteristic of Andrew, which is most pertinent to this paper, is its expected final size. Each individual at CMU may eventually possess an Andrew workstation, thus implying a scale of 5000 to 10,000 nodes.

A fundamental component of Andrew is the distributed file system that constitutes the underlying information sharing mechanism. A detailed

This work was performed as a joint project of Carnegie Mellon University and the IBM Corporation. M. Satyanarayanan was supported in the writing of this paper by the National Science Foundation under contract CCR-8657907. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies of the IBM Corporation, the National Science Foundation, or Carnegie Mellon University.

Please direct correspondence to M. Satyanarayanan, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

description of this file system has been presented in an earlier paper [6]. Using a set of trusted servers, collectively called *Vice*, the Andrew File System presents a homogeneous, location-transparent file name space to all the client workstations. Clients and servers run the 4.2 Berkeley Software Distribution (*4.2BSD*) of the UNIX operating system.[1] The operating system on each workstation intercepts file system calls and forwards them to a user-level process on that workstation. This process, called *Venus*, caches files from Vice and stores modified copies of files back on the servers they came from. Venus contacts Vice only when a file is opened or closed; reading and writing of individual bytes of a file are performed directly on the cached copy and bypass Venus.

This file system architecture was motivated primarily by considerations of scale. 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. Venus on each workstation does the locating of a file on a specific server and initiates a dialogue with that server.

Our intent in this paper is to examine the design of the Andrew File System at the next level of detail. In particular, we concentrate on those features and design decisions that bear on the scalability of the system. Large scale affects a distributed system in two ways: it degrades performance, and it complicates administration and day-to-day operation. This paper addresses both of these consequences of scale on Andrew and shows that the mechanisms we have incorporated cope successfully with these concerns.

Section 2 of the paper describes an initial prototype implementation and our experience with it. That section also introduces a synthetic benchmark that is used as the basis of performance comparison in the rest of the paper. On the basis of this experience we made many design changes. The rationale for these changes is presented in Section 3. Section 4 discusses the effect of these design changes on performance. To place our design in perspective and quantify its relative merits, Section 5 presents the results of running the same benchmark on an alternative contemporary distributed file system, Sun Microsystem's NFS [9]. Section 6 shows how the operability of the system has been enhanced by our design changes. Finally, in Section 7 we discuss issues that are related peripherally to scale and examine the ways in which the present design can be enhanced.

2. THE PROTOTYPE

Our primary goal in building a prototype was to validate the basic file system architecture. In the implementation we had to carefully balance two opposing constraints: the desire to obtain feedback on our design as rapidly as possible, and the need to build a system that was usable enough to make that feedback meaningful. In retrospect, the prototype was successful in both these respects. The prototype was used by ourselves as well as by about 400 other users. At the

[1] UNIX is a trademark of AT&T Bell Laboratories. To avoid any possible ambiguity, we use the name "4.2 BSD" throughout this paper for the specific version of UNIX used in our system.

peak of its usage, there were about 100 workstations and 6 servers. The workstations were Sun2 with 65MB local disks, and the servers were Sun2s or Vax-750s, each with 2 or 3 400MB disks. As the rest of this paper illustrates, the experience we gained from the prototype was invaluable in developing a considerably improved implementation of the Andrew File System.

2.1 Description

In the prototype, Venus on a client workstation would rendezvous with a process listening at a well-known network address on a server. This process then created a dedicated process to deal with all future requests from the client. The dedicated process persisted until its client terminated the network connection. In steady state, a server typically operated with as many processes as there were clients who had ever contacted it. Since 4.2BSD does not allow sharing of address spaces between processes, all communication and manipulation of data structures between server processes took place via files in the underlying file system. User-level file locking was implemented by a dedicated lock-server process that serialized requests from the separate server processes and maintained a lock table in its address space.

Data and associated Vice status information were stored in separate files. Each server contained a directory hierarchy mirroring the structure of the Vice files stored on it. Vice file status information, such as an access list, was stored in shadow directories called *.admin* directories. The directory hierarchy contained *Stub* directories that represented portions of the Vice name space located on other servers. The location database that maps files to servers was thus embedded in the file tree. If a file were not on a server, the search for its name would end in a stub directory which identified the server containing that file. Below the top levels of the Vice name tree, files in the same subtree were likely to be located on the same server. Hence clients cached pathname prefix information and used this as the basis of a heuristic to direct file requests to appropriate servers.

The Vice–Venus interface named files by their full pathname. There was no notion of a low-level name, such as the *inode* in 4.2BSD. A rudimentary form of read-only replication, restricted to the topmost levels of the Vice name tree, was present. Each replicated directory had a single server site to which all updates were directed. An asynchronous slow-propagation mechanism reflected changes made at this site to the read-only replicas at all other sites.

All cached copies of files were considered suspect by Venus. Before using a cached file, Venus would verify its timestamp with that on the server responsible for the file. Each open of a file thus resulted in at least one interaction with a server, even if the file were already in the cache and up to date.

2.2 Qualitative Observations

Our preliminary experience with the prototype was quite positive. Almost every application program on workstations was able to use Vice files without recompilation or relinking. This put to rest one of our key concerns: namely, the successful emulation of 4.2BSD file system semantics using caching and whole-file transfer. There were some areas of incompatibility with standard 4.2BSD semantics, but they were never serious enough to discourage use of the prototype.

Command execution involving Vice files was noticeably slower than similar commands involving local files. However, the performance was so much better than that of the heavily loaded timesharing systems used by the general user community at CMU that our users willingly suffered!

As we had anticipated, the performance degradation was not uniform across all operations. CPU-bound operations, like the compilation of a large program, were almost as fast as on a stand-alone system. Other operations, such as the recursive directory listing of a large subtree of files, took much larger when the subtree was in Vice.

We were puzzled by certain application programs that ran much slower than we had expected, even when all relevant files were in the local cache. It turned out that such programs used the *stat* primitive in 4.2BSD to test for the presence of files or to obtain status information before opening them. In pathological cases, a file would be *stat*-ed twice or thrice before being actually opened. Since each *stat* call involved a cache validity check, the total number of client-server interactions was significantly higher than the number of file opens. This factor increased both the total running time of these programs and the load on the servers. We attempted to alleviate this problem by placing an upper bound on the frequency with which we checked the validity of a cache entry. Although performance did improve, it was still not satisfactory.

We found that performance was usually acceptable up to a limit of about 20 active users per server. However, there were occasions when even a few persons using the file system intensely caused performance to degrade intolerably.

The prototype turned out to be difficult to operate and maintain. The use of a dedicated process per client on each server caused critical resource limits to be exceeded on a number of occasions. It also resulted in excessive context switching overhead and in high virtual memory paging demands. However, it did have the virtue of simplicity and resulted in a relatively robust system because the failure of an individual server process affected only one client. The remote procedure call (RPC) package was built on top of a reliable byte-stream abstraction provided by the kernel. Although this simplified our implementation, it frequently caused network-related resources in the kernel to be exceeded. Our decision to embed the file location database in stub directories in the Vice name tree made it difficult to move users' directories between servers. When disk storage on a server was exhausted, it was easier to add another disk rather than move a few users to another server! Our inability to enforce disk storage quotas on individual users exacerbated this problem.

2.3 The Benchmark

To quantify the performance penalty due to remote access, we ran a series of controlled experiments with a synthetic benchmark. This benchmark consists of a command script that operates on a collection of files constituting an application program. The operations are intended to be a representative sample of the kinds of actions an average user might perform. Although we do not demonstrate any statistical similarity between these file references and those observed in real systems, it provides a convenient yardstick for comparing a variety of file system implementations.

Table I. Stand-alone Benchmark Performance

	Machine type		
Benchmark phase	Sun2	IBM RT/25	Sun3/50
Overall	1054 (5)	798 (20)	482 (8)
MakeDir	16 (1)	13 (1)	10 (0)
Copy	40 (1)	37 (2)	31 (2)
ScanDir	70 (4)	51 (9)	44 (5)
ReadAll	106 (2)	132 (8)	51 (0)
Make	822 (2)	566 (11)	346 (1)

Notes: This table shows the elapsed time in seconds of the benchmark when it was run on the local file systems of different machines. Since no remote file accesses were made, the differences in times are due solely to the hardware and operating system implementation. The amount of real memory used by each of the machine types was as follows: Sun2, 2 Mbytes; IBM RT, 2.8 Mbytes; Sun 3, 4 Mbytes. All the machines were configured as workstations rather than as servers and had relatively low performance disks. Each of these experiments was repeated 3 times. Figures in parentheses are standard deviations.

Throughout this paper the term *Load Unit* refers to the load placed on a server by a single client workstation running this benchmark. Server load is varied by initiating the benchmark simultaneously on multiple client workstations and waiting for all of them to complete. We refrain from using the term "client" in reporting benchmark results to avoid the possible misinterpretation that we are referring to a human user.[2] Our observations of network traffic indicate that a load unit corresponds to about five Andrew users.

The input to the benchmark is a read-only source subtree consisting of about 70 files. The files are the source code of an application program and total about 200 kilobytes in size. There are 5 distinct phases in the benchmark:

MakeDir Constructs a target subtree that is identical in structure to the source subtree.

Copy Copies every file from the source subtree to the target subtree.

ScanDir Recursively traverses the target subtree and examines the status of every file in it. It does not actually read the contents of any file.

ReadAll Scans every byte of every file in the target subtree once.

Make Compiles and links all the files in the target subtree.

On a Sun2 workstation with a local disk, this benchmark takes about 1000 seconds to complete when all files are obtained locally. The corresponding times for other machines are shown in Table I.

2.4 Performance Observations

A fundamental quantity of interest in a caching file system is the hit ratio observed during actual use. Venus used two caches: one for files and the other for status information about files. A snapshot of the caches of 12 machines

[2] We are indebted to Jerry Saltzer for alerting us to this danger.

Table II. Distribution of Vice Calls in Prototype

		Call distribution						
Server	Total calls	TestAuth (%)	GetFileStat (%)	Fetch (%)	Store (%)	SetFileStat (%)	ListDir (%)	All others (%)
cluster0	1,625,954	64.2	28.7	3.4	1.4	0.8	0.6	0.9
cluster1	564,981	64.5	22.7	3.1	3.5	2.8	1.3	2.1
cmu-0	281,482	50.7	33.5	6.6	1.9	1.5	3.6	2.2
cmu-1	1,527,960	61.1	29.6	3.8	1.1	1.4	1.8	1.2
cmu-2	318,610	68.2	19.7·	3.3	2.7	2.3	1.6	2.2
Mean		61.7 (6.7)	26.8 (5.6)	4.0 (1.5)	2.1 (1.0)	1.8 (0.8)	1.8 (1.1)	1.7 (0.6)

Notes: Data shown here was gathered over a one-month period. The figures in parentheses are standard deviations.

Table III. Prototype Benchmark Performance

	Overall benchmark time		Time per TestAuth call	
Load units	Absolute (s)	Relative (%)	Absolute (ms)	Relative (%)
1	1789 (3)	100	87 (0)	100
2	1894 (4)	106	118 (1)	136
5	2747 (48)	154	259 (16)	298
8	5129 (177)	287	670 (23)	770
10	7326 (69)	410	1050 (13)	1207

Notes: Each data point is the mean of 3 trials. Clients and servers were Sun2s. Each client had a 300-entry cache. Figures in parentheses are standard deviations. In each row, the value in a column marked "Relative" is the ratio of the absolute value at that load to its value at load 1. Part of the data presented here is reproduced in Figure 1.

showed an average file-cache hit ratio of 81 percent, with a standard deviation of 9.8 percent and an average status-cache hit ratio of 82 percent, with a standard deviation of 12.9 percent.

Also of interest is the relative distribution of client/server interactions. Such a profile is valuable in improving server performance, since attention can be focused on the most frequent calls. Table II shows the observed distribution of those Vice calls that accounted for more than 1 percent of the total. These data were gathered over a 1-month period on 5 servers. The distribution is dramatically skewed, with 2 calls accounting for nearly 90 percent of the total. The *TestAuth* call validated cache entries, whereas *GetFileStat* obtained status information about files absent from the cache. The table also shows that only 6 percent of the calls to Vice (*Fetch* and *Store*) actually involved file transfer and that the ratio of *Fetch* calls to *Store* calls was approximately 2:1.

We also performed a series of controlled experiments using the benchmark. Table III presents the total running time for the benchmark as a function of server load. The table also shows the average response time for the most frequent Vice operation, *TestAuth*, during each of the experiments. One important observation from this table is that the benchmark took about 70 percent longer

Table IV. Prototype Server Usage

		CPU utilization			Disk 1			Disk 2		
Server	Samples	Total	User	System	Utilization	KBytes	xfers	Utilization	KBytes	xfers
cluster0	13	37.8%	9.6%	28.2%	12.0%	380,058	132,804	6.8%	186,017	75,212
		(12.5)	(4.4)	(8.4)	(3.3)	(84,330)	(35,796)	(4.2)	(104,682)	(42,972)
cluster1	14	12.6%	2.5%	10.1%	4.1%	159,336	45,127	4.4%	168,137	49,034
		(4.0)	(1.1)	(3.4)	(1.3)	(41,503)	(21,262)	(2.1)	(63,927)	(32,168)
cmu-0	15	7.0%	1.8%	5.1%	2.5%	106,820	28,177			
		(2.5)	(0.7)	(1.8)	(0.9)	(41,048)	(10,289)			
cmu-1	14	43.2%	7.2%	36.0%	13.9%	478,059	126,257	15.1%	373,526	140,516
		(10.0)	(1.8)	(8.7)	(4.5)	(151,755)	(42,409)	(5.4)	(105,846)	(40,464)

Notes: Data shown here was gathered from servers over two weeks, from 9 am to 5 pm on weekdays. Figures in parentheses are standard deviations.

at a load of 1 than in the stand-alone case. A second observation is that the time for *TestAuth* rose rapidly beyond a load of 5, indicating server saturation. For this benchmark, therefore, a server load between 5 and 10 was the maximum feasible.

For measuring server usage, we installed software on servers to maintain statistics about CPU and disk utilization and about data transfers to and from the disks. Table IV presents these data for four servers over a 2-week period. The data are restricted to observations made during 9:00 A.M. to 5 P.M. on weekdays, since this was the period of greatest system use. As the CPU utilizations in the table show, the servers loads were not evenly balanced. This fact is confirmed by Table II, which shows a spread of about 5 : 1 in the total number of Vice calls presented to each server. Under these circumstances, moving users to less heavily loaded servers would have improved the quality of service considerably.

Table IV also reveals that the 2 most heavily used servers showed an average CPU utilization of about 40 percent. This is a very high figure, considering that it was an average over an 8-hour period. Closer examination of the raw data showed much higher short-term CPU utilization: Figures in the neighborhood of 75 percent over a 5-minute averaging period were common. Disk utilizations, however, were much lower. The 8-hour average was less than 15 percent, and the short-term peaks were rarely above 20 percent. We concluded from these figures and from server utilization data obtained during the benchmarks, that the performance bottleneck in our prototype was the server CPU. On the basis of profiling the servers, we deduced that the 2 factors chiefly responsible for this high CPU utilization were the frequency of context switches between the many server processes and the time spent by the servers in traversing full pathnames presented by workstations.

To summarize, the measurements reported in this section indicate that significant performance improvement is possible if we reduce the frequency of cache validity checks, reduce the number of server processes, require workstations rather than the servers to do pathname traversals, and balance server usage by reassigning users.

3. CHANGES FOR PERFORMANCE

On the basis of our experience with the prototype, we set out to build a revised version of the Andrew File System. Although we were under no constraint to reuse code or ideas, the resulting design uses the same fundamental architectural principle as the prototype: *Workstations cache entire files from a collection of dedicated autonomous servers.* Our analysis convinced us that the shortcomings of the prototype were due to inadequacies in its realization rather than in its basic architecture. We were also convinced that this was the most promising path to our goal of supporting at least 50 clients per server.

Some aspects of the prototype implementation have remained unchanged. Both Venus and server code run as user-level processes. Communication between servers and clients is based on the RPC paradigm and uses an independently optimized protocol for the transfer of bulk data. The mechanism in the workstation kernels to intercept and forward file requests to Venus is the same as in the prototype.

Although we retained these aspects of the prototype, we have changed many details. The changes fall into two categories: those made to enhance performance and those made to improve the operability of the system. In this section we describe the changes made for performance and defer discussion of changes for operability until Section 6. The changes for performance are in four distinct areas:

—Cache management
—Name resolution
—Communication and server process structure
—Low-level storage representation

These are orthogonal changes, although a small degree of interdependency is inevitable. We discuss the individual changes in Sections 3.1 to 3.4 and then describe their synthesis in Section 3.5.

3.1 Cache Management

Caching, the key to Andrew's ability to scale well, is further exploited in our redesign. Venus now caches the contents of directories and symbolic links in addition to files. There are still two separate caches, one for status and the other for data. Venus uses a simple least-recently-used (*LRU*) algorithm to keep each of them bounded in size. The status cache is kept in virtual memory to allow rapid servicing of *stat* system calls. Each entry contains information such as the size of a file and its modification timestamp. The data cache is resident on the local disk, but the 4.2BSD I/O buffering mechanism does some caching of disk blocks in memory that is transparent to Venus.

Modifications to a cached file are done locally and are reflected back to Vice when the file is closed. As mentioned earlier, Venus intercepts only the opening and closing of files and does not participate in the reading or writing of individual bytes on a cached copy. For reasons of integrity, modifications to a directory are made directly on the server responsible for that directory. However, Venus reflects the change in its cached copy to avoid refetching the directory.

A significant point of departure from the prototype is the manner in which cache entries are kept consistent. Rather than checking with a server on each open, Venus now 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*, dramatically reduces the number of cache validation requests received by servers. A small amount of cache validation traffic is still present, usually to replace callbacks lost because of machine or network failures. When a workstation is rebooted, Venus considers all cached files and directories suspect and generates a cache validation request for the first use of each such entry.

Callback complicates the system because each server and Venus now maintain callback state information. Before modifying a file or directory, a server has to notify every workstation that has a callback on that file. If the amount of callback state maintained by a server is excessive, its performance may degrade. Under such circumstances, it may be appropriate for servers to break callbacks and reclaim storage. Finally, there is a potential for inconsistency if the callback state maintained by a Venus gets out of sync with the corresponding state maintained by the servers.

In spite of these complications, we are convinced of the importance of callback. By reducing cache validation traffic, callback reduces the load on servers considerably. Callback also makes it feasible to resolve pathnames on workstations, as described in the next section. In the absence of callback, the lookup for every component of a pathname would generate a cache validation request.

3.2 Name Resolution

In a conventional 4.2BSD system, a file has a unique, fixed-length name, its *inode*, and one or more variable-length *Pathnames* that map to this inode. The routine that performs this mapping, *namei*, is usually one of the most heavily used and time consuming parts of the kernel. In our prototype, Venus was aware only of pathnames; there was no notion of an inode for a Vice file. However, because of the data representation on our servers, each Vice pathname presented by a Venus involved an implicit *namei* operation on the server to locate the file. This resulted in considerable CPU overhead on the servers and was an obstacle to scaling. It also made full emulation of 4.2BSD semantics difficult.

To alleviate these problems we reintroduced the notion of two-level names. Each Vice file or directory is now identified by a unique fixed-length *Fid*. Each entry in a directory maps a component of a pathname to a fid. Venus now performs the logical equivalent of a *namei* operation, and maps Vice pathnames to fids. Servers are presented with fids and are, in fact, unaware of pathnames. As discussed in Section 3.4, we have performed further optimizations to ensure that no implicit *namei* operations are performed on a server when accessing data.

A fid is 96 bits long and has 3 components: a 32-bit *Volume number*, a 32-bit *Vnode number*, and a 32-bit *Uniquifier*. The volume number identifies a collection of files, called a *Volume*, located on one server. Volumes are discussed in Section 6. The vnode number is used as an index into an array containing the file storage information for the files in a single volume. The actual accessing of file data, given a fid, is thus an efficient operation. The uniquifier guarantees that no fid is ever used twice in the history of the file system. This allows reuse of vnode numbers, thereby keeping certain critical server data structures compact.

It is important to note that a fid contains no explicit location information. Moving files from one server to another does not, therefore, invalidate the contents of directories cached on workstations. Location information is contained in a *Volume Location Database* replicated on each server. This is a slowly changing database that allows every server to identify the location of every volume in the system. It is the aggregation of files into volumes that make it possible to keep the location database at a manageable size.

3.3 Communication and Server Process Structure

As the context switching and paging overheads in our prototype indicated, the use of a server process per client did not scale well. A related problem was that server processes could not cache critical shared information in their address spaces because 4.2BSD does not permit processes to share virtual memory. The redesign solves these problems by using a single process to service all clients of a server.

Since multiple threads of control provide a convenient programming abstraction, we have built a user-level mechanism to support multiple nonpreemptive *Lightweight Processes* (*LWPs*) within one process. Context switching between LWPs is only on the order of a few procedure-call times. The number of LWPs (typically five) is determined when a server is initialized and remains fixed thereafter. An LWP is bound to a particular client only for the duration of a single server operation. A client thus has long-term state on a server but not a corresponding thread of control associated with it. Since Venus also uses the LWP mechanism, it can act concurrently on remote file access requests from multiple user processes on its workstation.

As in the prototype, clients and servers communicate via an RPC mechanism. Unlike the prototype, however, this implementation is entirely outside the kernel and is capable of supporting many hundreds or thousands of clients per server. It is integrated with the LWP mechanism and thus allows the server to continue servicing client requests unless all its LWPs are blocked on network events. The RPC mechanism runs on a variety of workstations, provides exactly-once semantics in the absence of failures, supports whole-file transfer using an optimized bulk transfer protocol, and provides secure, authenticated communication between workstations and servers.

3.4 Low-Level Storage Representation

Our decision to retain 4.2BSD on the servers implied that files would hold Vice data, as in the prototype. As mentioned in Section 3.2, we were wary of the cost of the *namei* operations involved in accessing data via pathnames. Therefore, we decided to access files by their inodes rather than by pathnames. Since the internal inode interface is not visible to user-level processes, we had to add an appropriate set of system calls. The vnode information for a Vice file identifies the inode of the file storing its data. Data access on a server is thus quite rapid; it consists of an index of a fid into a table to look up vnode information, followed by an *iopen* call to read or write the data.

For efficiency, Venus also uses this mechanism. A local directory on the workstation is used as the cache. Within the directory are files whose names are placeholders for cache entries. Venus accesses these files directly by their inodes. We have thus eliminated nearly all pathname lookups on workstations and servers, except explicit ones performed on cached directories by Venus. Such explicit lookups are, in fact, faster than kernel lookups because of the improved internal organization of Vice directories.

3.5 Overall Design

The result of our redesign can be best understood by examining a remote file access in detail. Suppose a user process opens a file with pathname *P* on a workstation. The kernel, in resolving *P*, detects that it is a Vice file and passes it to Venus on that workstation. One of the LWPs comprising Venus now uses the cache to examine each directory component *D* of *P* in succession:

—If *D* is in the cache and has a callback on it, it is used without any network communication.

—If *D* is in the cache but has no callback on it, the appropriate server is contacted, a new copy of *D* is fetched if it has been updated, and a callback is established on it.

—If *D* is not in the cache, it is fetched from the appropriate server, and a callback is established on it.

When the target file *F* is identified, a current cache copy is created in the same manner. Venus then returns to the kernel, which opens the cached copy of *F* and returns its handle to the user process. Thus, at the end of the pathname traversal, all the intermediate directories and the target file are in the cache with callbacks on them. Future references to this file will involve no network communication at all, unless a callback is broken on a component of *P*. Venus regains control when the file is closed and, if it has been modified locally, updates it on the appropriate server. An LRU replacement algorithm is periodically run to reclaim cache space.

When processing a pathname component, Venus identifies the server to be contacted by examining the volume field of the fid of that component. If an entry for this volume is not present in a mapping cache, Venus contacts any server that it already has a connection to, requests the location information, and enters it into the mapping cache. Unless Venus already has a connection to the server responsible for that volume, it establishes a new connection. It then uses this connection to fetch the file or directory. Connection establishment and future requests from the workstation are serviced by any of the LWPs comprising the server process.

The above description is a simplified view of the actual sequence of events [2]. In particular, authentication, protection checking, and network failures complicate matters considerably. Also, since the other LWPs in Venus may be concurrently servicing file access requests from other processes, accesses to cache data structures must be synchronized. However, although the initial access of a file may be complex and rather expensive, further accesses to it are much simpler and cheaper. It is the locality inherent in actual file access patterns that makes this strategy viable.

Table V. Vice Interface

Fetch	Returns the status and (optionally) data of the specified file or directory and places a callback on it.
Store	Stores the status and (optionally) data of the specified file.
Remove	Deletes the specified file.
Create	Creates a new file and places a callback on it.
Rename	Changes the name of a file or a directory. Cross-volume renames are illegal.
Symlink	Creates a symbolic link to a file or directory.
Link	Creates a hard link to a file. Cross-directory links are illegal.
Makedir	Creates a new directory.
Removedir	Deletes the specified directory which must be empty.
SetLock	Locks the specified file or directory in shared or exclusive mode. Locks expire after 30 minutes.
Releaselock	Unlocks the specified file or directory.
GetRootVolume	Returns the name of the volume containing the root of Vice.
GetVolumeInfo	Returns the name(s) of servers that store the specified volume.
GetVolumeStatus	Returns status information about the specified volume.
SetVolumeStatus	Modifies status information on the specified volume.
ConnectFS	Initiates dialogue with a server.
DisconnectFS	Terminates dialogue with a server.
RemoveCallBack	Specifies a file that Venus has flushed from its cache.
GetTime	Synchronizes the workstation clock.
GetStatistics	Returns server CPU, memory, and I/O utilization.
CheckToken	Determines whether the specified authentication token for a user is valid.
DisableGroup	Temporarily disables membership in a protection group.
EnableGroup	Enables membership in a temporarily disabled protection group.
BreakCallback	Revokes the callback on a file or directory. Made by a server to Venus.

Some of the complexity of our implementation arises from our desire to provide a useful yet efficient notion of file consistency across multiple machines. We examined a variety of choices ranging from the strict serializability of operations typically provided by database systems to the laissez-faire attitude exemplified by the SUN NFS file system in which a file created on a workstation may not be visible on another workstation for 30 seconds. Our design converged on the following consistency semantics:

—Writes to an open file by a process on a workstation are visible to all other processes on the workstation immediately but are invisible elsewhere in the network.

—Once a file is closed, the changes made to it are visible to new opens anywhere on the network. Already open instances of the file do not reflect these changes.

—All other file operations (such as protection changes) are visible everywhere on the network immediately after the operation completes.

—Multiple workstations can perform the same operation on a file concurrently. In conformance with 4.2BSD semantics, no implicit locking is performed. Application programs have to cooperate in performing the necessary synchronization if they care about the serialization of these operations.

Actual usage has convinced us that this is a useful and easily understood model of consistency in a distributed file system. It is also one that we have successfully implemented without serious performance penalty.

Table VI. Andrew Benchmark Times

	Overall time (in seconds)		Time for each phase (in seconds)				
Load units	Absolute	Relative (%)	MakeDir	Copy	ScanDir	ReadAll	Make
1	949 (33)	100	14 (1)	85 (28)	64 (3)	179 (14)	608 (16)
2	948 (35)	100	14 (1)	82 (16)	65 (9)	176 (13)	611 (14)
5	1050 (19)	111	17 (1)	125 (30)	86 (0)	186 (17)	637 (1)
8	1107 (5)	117	22 (1)	159 (1)	78 (2)	206 (4)	641 (6)
10	1293 (70)	136	34 (9)	209 (13)	76 (5)	200 (7)	775 (81)
15	1518 (28)	160	45 (3)	304 (5)	81 (4)	192 (7)	896 (12)
20	1823 (42)	192	58 (1)	433 (45)	77 (4)	192 (6)	1063 (64)

Notes: The clients were IBM RT/25s on a token ring, and the server was a Sun2 on an Ethernet. Most of the clients were 1 router hop away from the server, but a few were 2 hops away. Each of the experiments was repeated 3 times. Figures in parentheses are standard deviations. In each row, the value in a column marked "Relative" is the ratio of the absolute value at that load to its value at load 1.

Finally, it is important to note that the changes we describe in this paper are only those relevant to scale. Other changes, typically for better 4.2BSD emulation or security, are not discussed here. The current interface between Venus and Vice is summarized in Table V.

4. EFFECT OF CHANGES FOR PERFORMANCE

The revised implementation of the Andrew File System has been operational for over a year. The evaluation of this system focuses on two questions. First, how effective were our changes? In particular, has the anticipated improvement in scalability been realized? Second, what are the characteristics of the system in normal operation? The first question is addressed in Section 4.1, and information pertinent to the second question is presented in Section 4.2.

4.1 Scalability

To investigate the behavior of the system, we repeated the experiments that we had performed on the prototype. The server was a Sun2, as in the experiments on the prototype, but the clients were IBM-RTs. Table VI shows the absolute and relative times of the benchmark as a function of server load. The times for the individual phases of the benchmark are also shown in this table. Figure 1 presents some of this data graphically and compares it with prototype data from Table III.

The performance penalty for remote access has been reduced considerably. Data from Tables I and VI show that an Andrew workstation is 19 percent slower than a stand-alone workstation. The prototype was 70 percent slower. The improvement in scalability is remarkable. In the prototype, the benchmark took more than 4 times as long at a load of 10 as at a load of 1. In the current system, it takes less than twice as long at a load of 20 as at a load of 1. At a load of 10, it takes only 36 percent longer.

Table VI shows that the Copy and Make phases are most susceptible to server load. Since files are written in both these phases, interactions with the server for

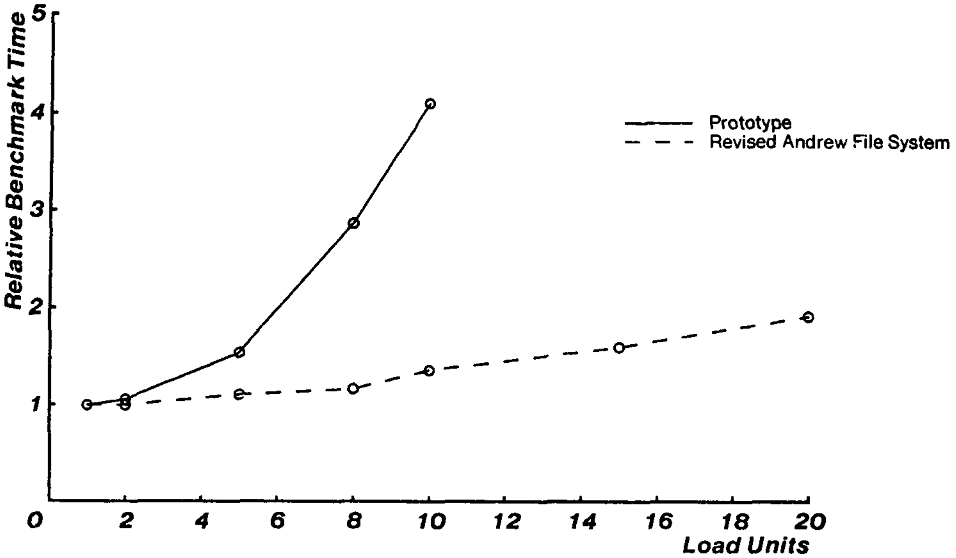

Fig. 1. Relative running time of benchmark. This figure compares the degradation in performance of the prototype and the current Andrew file system as a function of load. The clients were Sun2s in the prototype and IBM RTs in the current file system. The server was a Sun2 in both cases. Tables III and VI present this information in greater detail.

Table VII. Andrew Server Utilization During Benchmark

	Utilization (percent)	
Load units	CPU	Disk
1	8.1 (0.7)	2.7 (0.1)
2	15.0 (1.3)	4.7 (0.4)
5	29.4 (1.5)	9.2 (0.3)
8	41.8 (0.8)	12.8 (0.6)
10	54.6 (6.6)	17.8 (3.6)
15	64.7 (1.2)	20.9 (0.1)
20	70.9 (2.2)	23.6 (0.6)

Notes: This table shows the Sun2 server CPU and disk utilization as a function of load. The utilizations are averaged over the entire duration of the benchmark. These data were obtained from the same experiment as Table VI. Each of the experiments was repeated 3 times. Figures in parentheses are standard deviations. A part of the data is reproduced in Figure 2.

file stores are necessary. Further, it is during the Copy phase that files are fetched and callbacks established. In contrast, the ScanDir and ReadAll phases are barely affected by load. Callback eliminates almost all interactions with the server during these phases.

Table VII and Figure 2 present CPU and disk utilization on the server during the benchmark. CPU utilization rises from about 8 percent at a load of 1 to over

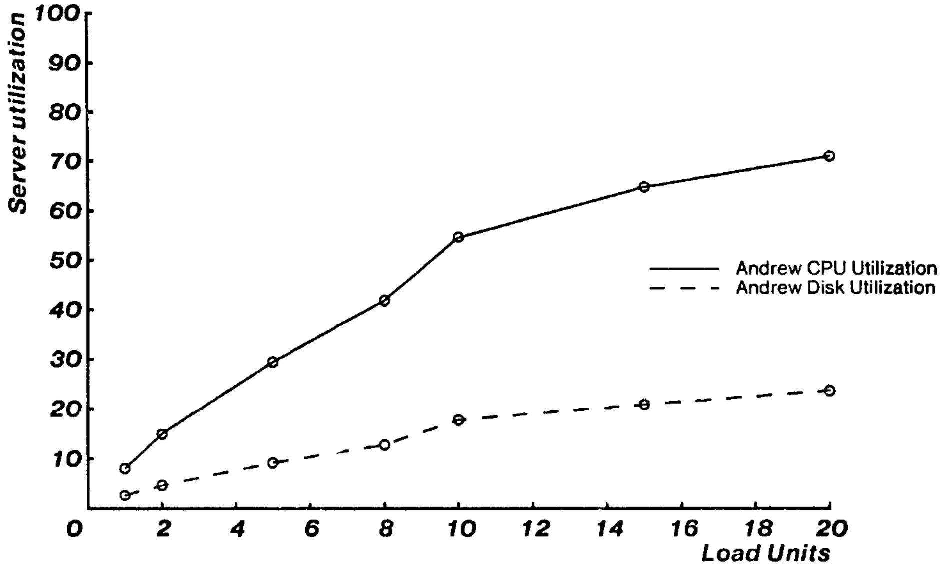

Fig. 2. Andrew server utilization during benchmark. Server CPU and disk utilization are presented in this figure as a function of load. All clients are IBM RTs and use a single Sun2 server. Table VII presents this information in greater detail.

70 percent at a load of 20. But disk utilization is below 25 percent even at a load of 20. This indicates that the server CPU still limits performance in our system, though it is less of a bottleneck than in the prototype. Better performance under load will require more efficient server software or a faster server CPU. Figure 2 shows an anomaly at a load of 10. Since the corresponding data in Table VII show a high standard deviation, we suspect that server activity unrelated to our experiments occurred during one of these trials.

In summary, the results of this section demonstrate that our design changes have improved scalability considerably. At a load of 20, the system is still not saturated. Since a load unit corresponds to about 5 typical Andrew users, we believe our scale goal of 50 users per server has been met.

4.2 General Observations

Table VIII presents server CPU and disk utilizations in Andrew. The figures shown are averages over the 8-hour period from 9:00 A.M. to 5:00 P.M. on weekdays. Most of the servers show CPU utilizations between 15 and 25 percent. One of the servers, vice4, shows a utilization of 35.8 percent, but the disk utilization is not correspondingly high. The high standard deviation for the CPU utilization leads us to believe that this anomaly was caused by system maintenance activities that were unexpectedly performed during the day rather than at night. Server vice9, on the other hand, shows a CPU utilization of 37.6 percent with a small standard deviation. The disk utilization is 12.1 percent, the highest of any server. The high utilization is explained by the fact that this

Table VIII. Andrew Server Usage

		CPU utilization			Disk 1		Disk 2		Disk 3	
Server	Samples	Total	User	System	Utilization	KBytes	Utilization	KBytes	Utilization	KBytes
vice2	4	16.7	3.5	13.2	2.9%	149,525	1.9%	109,058	0.5%	20,410
		(4.5)	(1.2)	(3.6)	(0.3)	(15,461)	(1.0)	(64,453)	(0.1)	(2,794)
vice3	5	19.2	5.1	14.1	3.0%	126,951	2.7%	98,441	2.3%	96,445
		(2.8)	(2.1)	(1.6)	(0.6)	(24,957)	(0.7)	(26,975)	(0.3)	(13,758)
vice4	4	35.8	14.1	21.7	4.8%	195,618	3.6%	140,147	5.2%	217,331
		(24.5)	(16.9)	(8.9)	(3.0)	(132,128)	(1.1)	(47,836)	(3.5)	(151,199)
vice5	5	19.9	3.5	16.4	3.2%	152,764	3.2%	174,229	0.9%	37,851
		(2.9)	(0.7)	(2.4)	(0.4)	(19,247)	(0.5)	(26,849)	(0.2)	(9,167)
vice6	5	14.3	2.4	12.0	2.4%	117,050	2.3%	131,305	0.4%	14,923
		(1.4)	(0.3)	(1.2)	(0.3)	(17,244)	(0.3)	(15,950)	(0.1)	(3,985)
vice7	5	26.0	6.4	19.5	6.9%	349,075	0.2%	4,143	1.4%	59,567
		(3.2)	(1.0)	(2.3)	(0.2)	(9,515)	(0.1)	(4,217)	(1.2)	(60,504)
vice8	5	7.5	1.5	6.0	1.4%	62,728	0.3%	6,079	0.6%	28,704
		(1.2)	(0.4)	(0.8)	(0.3)	(12,226)	(0.1)	(9,025)	(0.4)	(24,248)
vice9	5	37.6	7.2	30.4	12.1%	558,839	2.5%	103,109	2.2%	103,517
		(2.7)	(0.4)	(2.3)	(1.4)	(63,592)	(0.8)	(38,392)	(0.5)	(21,171)
vice10	5	23.3	6.1	17.2	5.8%	262,846	2.1%	82,502	1.7%	74,043
		(8.6)	(5.0)	(4.8)	(1.8)	(82,210)	(0.7)	(44,606)	(0.8)	(35,699)
vice11	5	18.0	5.6	12.5	3.0%	124,783	3.0%	129,667	0.8%	24,321
		(6.0)	(4.4)	(2.0)	(0.5)	(23,398)	(0.9)	(39,455)	(0.2)	(7,715)
vice12	5	13.2	5.8	7.5	2.7%	118,960	1.5%	49,022	0.2%	71
		(10.1)	(7.9)	(2.7)	(1.0)	(49,760)	(0.5)	(20,452)	(0.0)	(58)
vice13	5	12.5	3.1	9.4	1.6%	70,632	0.9%	26,687	1.9%	84,874
		(2.4)	(1.2)	(1.3)	(0.2)	(8,476)	(0.3)	(10,974)	(0.5)	(26,433)
vice14	5	15.3	7.0	8.3	2.3%	104,861	1.1%	34,442	1.0%	36,587
		(12.0)	(8.6)	(3.5)	(1.1)	(57,648)	(0.3)	(12,202)	(0.5)	(25,190)

Notes: These data were gathered over a week from 9 February 1987 to 15 February 1987. Omitted from this table are servers used for experimental versions of the system. Three of the servers in the table above (vice11, vice13, vice14) had a fourth disk. In all cases the utilization of that disk was less than 0.5 percent. All the servers listed above were Sun2s. Figures in parentheses are standard deviations.

server stores the bulletin boards, a collection of directories that are frequently accessed and modified by many different users.

The distribution of Vice calls over a three-day period is shown in Table IX. The servers with the most calls are vice7, which stores common system files used by all workstations, and vice9, the server that stores bulletin boards. The most frequent call is *GetTime*, which is used by workstations to synchronize their clocks and as an implicit keepalive. The next most frequent call is *FetchStatus*. We conjecture that many of these calls are generated by users listing directories in parts of the file name space that they do not have cached. It is interesting that despite caching, fetches dominate stores. The call *RemoveCB* is made by Venus when it flushes a cache entry. Server vice9 shows one of the highest occurrences of *RemoveCB* which indicates that the files it stores exhibit poor locality. This is precisely the behavior one would expect of bulletin boards, since users tend not to read bulletin board entries more than once. Only vice8, which is a special server used by the operations staff, shows a higher occurrence of *RemoveCB*. On

Table IX. Distribution of Calls to Andrew Servers

		Call distribution								
Server	Total calls	FetchData (%)	FetchStatus (%)	StoreData (%)	StoreStatus (%)	GetStat (%)	RemoveCB (%)	GetTime (%)	VolStats (%)	Other (%)
vice2	274,432	10.9	29.6	0	0	1.7	0	35.2	0	22.6
vice3	307,200	7.2	19.3	4.0	2.6	1.5	12.7	19.1	25.4	8.2
vice4	405,504	10.3	21.7	7.2	3.2	1.2	14.5	18.2	13.6	10.1
vice5	348,160	13.1	36.3	0.2	0.3	1.3	11.6	29.7	0	7.5
vice6	212,992	11.8	33.2	0	0	2.2	0.3	41.9	0	10.6
vice7	708,608	12.8	26.2	3.3	3.9	0.7	7.9	14.6	1.8	28.8
vice8	40,960	8.6	29.2	1.9	0.5	9.7	26.7	12.2	0	11.2
vice9	692,224	19.3	37.9	1.6	0.1	0.6	21.8	10.5	1.1	7.1
vice10	208,896	11.3	21.7	5.0	3.6	1.2	13.7	24.7	11.7	7.1
vice11	368,640	7.5	35.8	3.6	1.9	1.3	10.3	18.8	12.9	7.9
vice12	122,880	8.8	18.2	4.4	3.1	3.8	12.9	31.0	9.7	8.1
vice13	180,224	9.8	19.0	5.5	4.0	2.6	12.1	25.1	13.6	8.3
vice14	114,688	8.2	12.9	3.9	2.2	2.5	9.6	19.3	34.4	7.0

Notes: The data in this table were gathered from the main Andrew servers over 78 hours in the period from 3 February 1987 to 6 February 1987. Data for three servers running an experimental version of the system are not shown here. The user information in Table X is derived from the same set of observations. Note that the call *FetchData* fetches both data and status, whereas *FetchStatus* fetches only status. *StoreData* and *StoreStatus* are similarly related.

Table X. Active Users on Andrew Servers

Server	Type of volumes	Average active users	
		Overall	Peak period
vice2	Read-only, System	23	64
vice3	Read-write, User	23	56
vice4	Read-write, User	27	68
vice5	Read-only and Read-write	28	76
vice6	Read-only, System	18	52
vice7	Read-write, System	59	128
vice8	Special	2	3
vice9	Read-write, BBoard	25	63
vice10	Read-write, User	29	77
vice11	Read-write, User	23	58
vice12	Read-write, User	8	24
vice13	Read-write, User	13	31
vice14	Read-write, User	11	29

Notes: The second column describes the kind of volumes stored on each server. Server vice9 stores the bulletin boards, which are the most frequently updated set of directories shared by many users. Server vice5 has the read-write volumes whose read-only clones are on vice2, vice5 and vice6. Vice7 has the common system volumes that cannot be read-only and are therefore used by all of the workstations in the system. Data for 3 servers running an experimental version of the system are not shown here. An active user on a server is one on whose behalf some workstation has interacted with that server during the past 15 minutes. Peak period is defined to be 9:00 A.M. to 5:00 P.M. on weekdays.

the basis of these measurements, we have modified Venus to remove callback on groups of files rather than one file at a time, when possible. This modification has reduced the observed frequency of *RemoveCB* considerably.

Table X, derived from the same set of observations as Table IX, shows the type of data stored on each server and the average number of persons actively using that server. During the peak period of use, most of the servers have between 50 and 70 active users, that is, those on whose behalf a request other than *GetTime* has been received in the last 15 minutes. In interpreting this data it should be kept in mind that a user often uses files from many different servers.

Although we do not present detailed data here, network utilization is quite low, typically in the neighborhood of 5 percent for the 10-megabit Ethernet and 12 percent for the 4-megabit token ring. The routers that interconnect segments of the local area network have occasionally shown signs of overload. This problem does not yet cause us concern but may require attention in the future.

5. COMPARISON WITH A REMOTE-OPEN FILE SYSTEM

The caching of entire files on local disks in the Andrew File System was motivated primarily by the following 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. [4] 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.

Our approach does have its drawbacks. Although diskless operation is possible, workstations require local disks for acceptable performance. Files that are larger than the local disk cache cannot be accessed at all. Strict emulation of 4.2BSD concurrent read and write semantics across workstations is impossible, since reads and writes are not intercepted. Building a distributed database using such a file system is difficult, if not impossible.

Despite these disadvantages, we persisted in our approach because we believed it would provide superior performance in a large scale system. The drawbacks listed in the previous paragraph have not proved to be significant in actual usage in our environment. And, as the discussions of Section 4 have established, the Andrew File System does scale well. But could an alternative design have produced equivalent or better results? How critical to scaling are caching and whole-file transfer? The rest of this section examines these questions in detail.

5.1 Remote Open

A number of distributed file systems such as Locus [12], IBIS [11], and the Newcastle Connection [1], have been described in the research literature and surveyed by Svobodova [10]. The design of such systems has matured to the point where vendor-supported implementations like Sun Microsystem's NFS [9], AT&T's RFS [5], and Locus are available.

Although the details of these systems vary considerably, all of them share one fundamental property: *the data in a file are 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 in which a file is actually opened on the remote site rather than the local site. Only the Andrew File System and the Cedar File System [7] employ caching of entire files as their remote access mechanism.

To explore how vital our approach is to scaling, we compared Andrew under controlled conditions to a representative of the set of remote open file systems. We selected Sun Microsystem's NFS as the candidate for comparison for the following reasons:

(1) NFS is a mature product from a successful vendor of distributed computing hardware and software. It is not a research prototype.

(2) Sun has spent a considerable amount of time and effort to tune and refine NFS. Deficiencies in its performance are therefore likely to be due to its

basic architecture, rather than inadequacies in implementation. A comparison of Andrew and NFS is thus most likely to yield significant insights into the scaling characteristics of caching and remote-open file systems.

(3) NFS and Andrew can run on precisely the same hardware and operating system. They can, in fact, coexist on the same machine and be used simultaneously. Using NFS allowed us to conduct controlled experiments in which the only significant variable was the file system component. The performance differences we observed were due to the design and implementation of the distributed file systems and were not artifacts of hardware, network, or operating system variation.

(4) There is a perception in the 4.2BSD user community that NFS is a de facto standard. We were curious to see how well Andrew measured up to it.

To be fair, it must be pointed out that NFS was not designed for operation in a large environment. It was designed as a distributed file system for use by a small collection of trusted workstations. It must also be emphasized that our comparison is based on a single benchmark. Other benchmarks may yield different results.

We also wish to emphasize that the focus of this comparison is scalability. The question of interest is "How does the performance perceived by a workstation degrade as the load on its server increases?" This justifies our comparison of NFS and Andrew on identical hardware configurations. A different question would be to compare the cost of NFS and Andrew configurations for a given level of performance at a given load. Since the price of hardware is subject to a variety of factors beyond the scope of this paper, we do not address this issue here.

5.2 The Sun Network File System

In this section we present a minimal overview of NFS. Only those details relevant to this paper are discussed here. Further information can be obtained from the documentation [9].

NFS does not distinguish between client and server machines. Any workstation can export a subtree of its file system and thus become a server. Servers must be identified and mounted individually; there is no transparent file location facility as in Andrew. Both the client and server components of NFS are implemented within the kernel and are thus more efficient than their counterparts in Andrew.

NFS caches inodes and individual pages of a file in memory. On a file open, the kernel checks with the remote server to fetch or revalidate the cached inode. The cached file pages are used only if the cached inode is up-to-date. The validity check on directory inodes is suppressed if a check was made within the last 30 seconds. Once a file is open, the remote site is treated like a local disk with read-ahead and write-behind of pages.

It is difficult to characterize the consistency semantics of NFS. New files created on a workstation may not be visible elsewhere for 30 seconds. It is indeterminate whether writes to a file at one site are visible to other sites that have file open for reading. New opens of that file will see the changes that have been flushed to the server. Because of the caching of file pages, processes on different workstations that perform interleaved write on a file will produce a result that is different from the same sequence of writes by processes on one workstation. Thus NFS neither provides strict emulation of 4.2BSD semantics, nor the open/close action consistency of Andrew.

5.3 Results of Comparison

The benchmark described in Section 2.3 was used as the basis of comparison between NFS and Andrew. Eighteen Sun3 workstations with local disks were available to us for our experiments. We added the Andrew kernel intercepts to these workstations so that Venus could be run on them. These modifications were orthogonal to NFS. A Sun3 was used as the server for both the Andrew and NFS trials. Clients and servers communicated on a 10-megabit Ethernet.

A set of experiments operating on files in NFS and another set operating on files in Andrew were run. The Andrew experiments consisted of two subsets: a *Cold Cache* set, in which workstation caches were cleared before each trial, and a *Warm Cache* set, in which caches were left unaltered. Since the target subtree is entirely re-created in each trial of a benchmark, the only benefit of a warm cache is that it avoids fetching of files from the source subtree. In all cases, at least three trials were performed for each experiment.

We ran into serious functional problems with NFS at high loads. At loads of ten or greater we consistently observed that some of the workstations terminated the final phase of the benchmark prematurely because of file system errors. Examination of the NFS source code revealed that the problem was probably being caused by lost RPC reply packets from servers during periods of high network activity. The RPC protocol used in NFS is based on unreliable datagrams and depends on retries at the operation level rather than at the RPC level. Nonidempotent file system calls that were retried by NFS sometimes failed, and these were reflected as file system errors in the running of the benchmark. Since the effective server load was lower than the nominal load in the last phase of these experiments, the results presented here are biased in favor of NFS at high loads. We did not encounter any functional problems of this nature with Andrew.

Table XI and Figure 3 present the overall running time in seconds of the benchmark as a function of server load. NFS performs slightly better than Andrew at low loads, but its performance degrades rapidly with increasing load. The crossover point is at a load of about 3 in the warm cache case and about 4 in the cold cache case. Close examination of Table XI reveals that the ScanDir, ReadAll, and Make phases contribute most to the difference in NFS and Andrew performance. Caching and callback in Andrew result in having the time for these phases only slightly affected by load. In NFS, the lack of a disk cache and the need to check with the server on each file open cause the time for these phases to be considerably more load dependent. The use of a warm cache in Andrew improves the time only for the Copy phase.

Figure 4 and Table XII present data on server CPU utilization during these experiments. At a load of 1, server CPU utilization is about 22 percent in NFS; in Andrew it is approximately 3 percent in both the cold and warm cache cases. At a load of 18, server CPU utilization saturates at 100 percent in NFS; in Andrew it is about 38 percent in the cold cache case and about 42 percent in the warm cache case.

Table XI. Benchwork Times of NFS and Andrew

		Time (seconds)			
Phase	Load units	NFS	Andrew cold	Andrew warm	Stand-alone
Overall	1	511 (1)	588 (2)	564 (23)	482 (8)
	2	535 (2)	582 (4)	567 (29)	
	5	647 (5)	605 (2)	564 (7)	
	7	736 (5)	636 (4)	573 (19)	
	10	888 (13)	688 (4)	621 (30)	
	15	1226 (12)	801 (2)	659 (8)	
	18	1279 (84)	874 (2)	697 (14)	
MakeDir	1	5 (1)	5 (1)	5 (1)	10 (0)
	2	8 (0)	5 (1)	5 (1)	
	5	18 (1)	7 (1)	7 (0)	
	7	33 (2)	9 (1)	8 (0)	
	10	59 (2)	12 (1)	12 (1)	
	15	82 (5)	18 (1)	19 (1)	
	18	81 (7)	24 (1)	18 (1)	
Copy	1	44 (0)	71 (4)	56 (8)	31 (2)
	2	51 (1)	72 (3)	57 (3)	
	5	84 (2)	85 (1)	58 (1)	
	7	95 (2)	104 (2)	62 (3)	
	10	107 (9)	137 (5)	88 (18)	
	15	164 (2)	200 (3)	116 (5)	
	18	245 (10)	241 (4)	133 (4)	
ScanDir	1	67 (1)	100 (2)	98 (14)	44 (5)
	2	67 (0)	98 (0)	96 (10)	
	5	72 (1)	97 (0)	95 (5)	
	7	78 (2)	97 (1)	96 (13)	
	10	85 (1)	94 (0)	99 (4)	
	15	107 (1)	91 (0)	82 (0)	
	18	111 (5)	90 (1)	96 (4)	
ReadAll	1	68 (1)	50 (3)	48 (1)	51 (60)
	2	76 (2)	50 (2)	57 (17)	
	5	93 (0)	47 (0)	49 (3)	
	7	117 (3)	48 (0)	48 (0)	
	10	152 (8)	48 (0)	49 (1)	
	15	215 (14)	48 (1)	48 (0)	
	18	211 (17)	48 (0)	48 (0)	
Make	1	327 (1)	363 (3)	357 (5)	346 (1)
	2	334 (3)	356 (2)	352 (1)	
	5	380 (4)	368 (2)	355 (1)	
	7	414 (3)	377 (2)	359 (3)	
	10	485 (8)	395 (2)	373 (10)	
	15	658 (15)	442 (2)	394 (6)	
	18	638 (73)	469 (3)	410 (11)	

Notes: These data correspond to the same set of experiments as Table XII, which describes the hardware configuration as well as problems encountered with NFS at loads of 10, 15, and 18. The stand-alone numbers are reproduced from Table I. A part of these data is reproduced in Figure 3. Figures in parentheses are standard deviations.

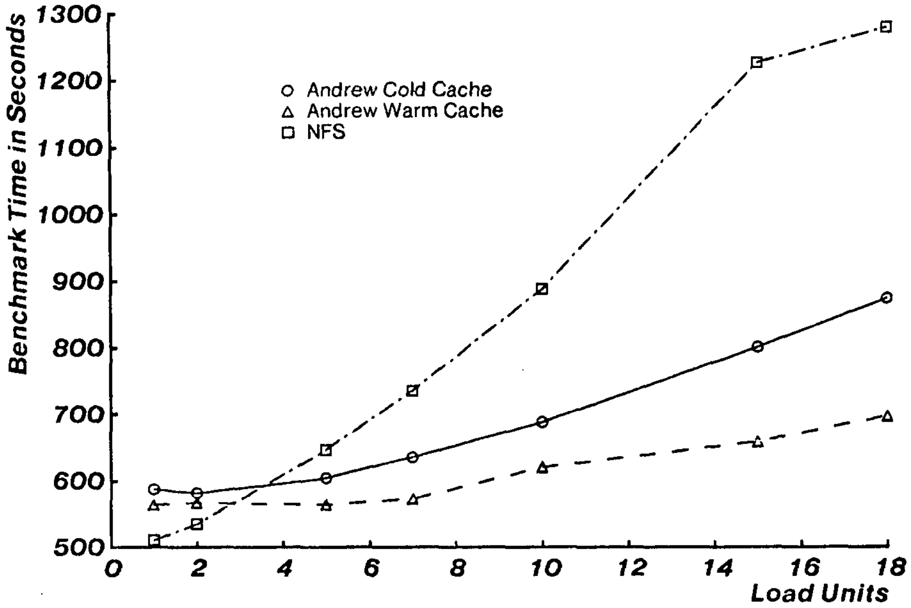

Fig. 3. NFS and Andrew overall benchmark times. This figures compares the benchmark times of NFS and the Andrew file system as a function of load. Table XI presents these data in greater detail. Table XII describes the conditions under which the data were obtained.

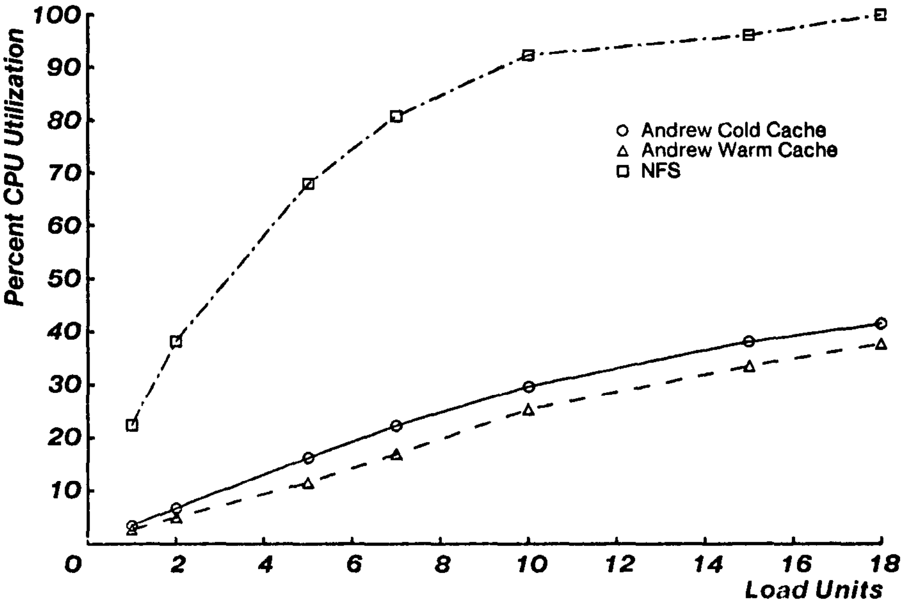

Fig. 4. NFS and Andrew server CPU utilization. This figure compares the server CPU utilizations of NFS and Andrew as a function of load. Table XII presents these data in greater detail and describes the conditions under which they were obtained.

Table XII. Server Utilization by NFS and Andrew

Load		CPU utilization			Disk 1		Disk 2	
units	File system	Total (%)	User (%)	System (%)	Utilization (%)	KBytes	Utilization (%)	KBytes
1	NFS	22.3 (0.1)	0.0 (0.0)	22.3 (0.1)	2.7 (0.1)	4,821 (103)	8.7 (0.2)	11,474 (277)
	Andrew cold	3.4 (0.1)	0.6 (0.1)	2.9 (0.1)	3.5 (0.2)	5,149 (449)	0.2 (0.1)	0 (0)
	Andrew warm	2.7 (0.1)	0.5 (0.1)	2.1 (0.1)	2.9 (0.2)	3,904 (355)	0.3 (0.0)	0 (0)
2	NFS	38.2 (0.2)	0.0 (0.0)	38.2 (0.2)	4.3 (0.1)	8,487 (15)	18.0 (0.8)	23,604 (429)
	Andrew cold	6.7 (0.1)	1.3 (0.1)	5.7 (0.1)	5.3 (0.2)	8,350 (754)	0.3 (0.1)	2 (0)
	Andrew warm	5.0 (0.1)	1.0 (0.0)	4.1 (0.1)	4.4 (0.2)	6,706 (783)	0.2 (0.1)	2 (0)
5	NFS	68.0 (0.7)	0.0 (0.0)	68.0 (0.7)	10.1 (0.3)	19,169 (890)	46.2 (0.4)	58,279 (2,698)
	Andrew cold	16.2 (0.3)	3.0 (0.2)	13.2 (0.1)	10.3 (0.2)	16,911 (743)	0.2 (0.0)	2 (0)
	Andrew warm	11.5 (0.1)	2.4 (0.1)	9.2 (0.1)	9.2 (0.4)	14,470 (512)	0.2 (0.1)	2 (0)
7	NFS	80.8 (0.4)	0.0 (0.1)	80.8 (0.3)	12.8 (0.2)	26,313 (420)	55.9 (0.3)	86,397 (2,755)
	Andrew cold	22.2 (0.3)	4.1 (0.1)	18.1 (0.3)	13.7 (0.9)	23,073 (1932)	0.3 (0.0)	4 (1)
	Andrew warm	16.9 (0.5)	3.4 (0.2)	13.5 (0.4)	10.8 (0.4)	17,665 (746)	0.2 (0.1)	2 (0)
10	NFS	92.3 (0.8)	0.0 (0.0)	92.3 (0.8)	16.8 (0.5)	35,989 (753)	71.1 (0.7)	124,982 (2,092)
	Andrew cold	29.6 (0.9)	5.8 (0.1)	24.0 (0.8)	18.7 (0.4)	34,568 (1,108)	0.3 (0.1)	3 (1)
	Andrew warm	25.3 (2.0)	5.1 (0.4)	20.1 (1.6)	16.5 (0.9)	27,886 (2,655)	0.2 (0.1)	4 (2)
15	NFS	96.2 (0.7)	0.0 (0.0)	96.2 (0.7)	19.4 (0.5)	52,198 (3,446)	86.3 (0.8)	213,181 (6,942)
	Andrew cold	38.1 (0.5)	7.1 (0.2)	31.0 (0.5)	24.8 (1.3)	51,830 (1,771)	0.3 (0.0)	5 (1)
	Andrew warm	33.5 (0.6)	6.9 (0.1)	26.6 (0.5)	22.5 (1.3)	38,953 (2,819)	0.2 (0.1)	6 (0)
18	NFS	100 (1.1)	0.0 (0.0)	100 (1.1)	19.3 (0.7)	53,858 (4,148)	95.0 (1.4)	243,547 (15,683)
	Andrew cold	41.5 (1.9)	8.0 (0.2)	33.4 (1.8)	27.6 (1.5)	58,901 (10,410)	0.3 (0.0)	6 (0)
	Andrew warm	37.7 (0.9)	7.6 (0.1)	30.1 (0.7)	24.6 (0.6)	46,628 (1,536)	0.3 (0.1)	5 (1)

Notes: This table shows the utilization of the server as a function of load. The clients were Sun3/50s with 4 Mbytes of real memory and a 70 Mbyte local disk. The server was a Sun3/160 with 8Mbytes of real memory and 2 450-Mbyte disks. In the Andrew experiments, system libraries and user files were both located on the same server disk (Disk1). In the NFS experiments, system libraries were located on one server disk (Disk1) and user files on the other (Disk2). In all the experiments, the system binaries were located on the local disks of each client. The clients and the server were on the same physical Ethernet cable with no intervening routers. Each of these experiments was repeated at least 3 times. Figures in parentheses are standard deviations. In the Andrew cold-cache experiments, the local disk cache was completely cleared before each trial. The warm cache experiments were run with cache state unchanged from the previous trial. A subset of the data is graphically displayed in Figues 3 and 4. Table XI corresponds to the same set of experiments.

In the NFS experiments, at loads of 10 or more, some of the clients failed to complete the final phase of the benchmark. The number of such premature terminations increased as the load increased. At a load of 18, at least 3 clients failed in each of the 6 trials. See Section 5.3 for a complete discussion of this problem. In both Andrew and NFS experiments, the ScanDir phase of the benchmark was run with an incorrect binary, which caused additional local file references and computation but no remote references. This added a fixed, load-independent overhead to that phase and lengthened the overall running time of the benchmark. The data presented above and in Table XI have been corrected to exclude this overhead.

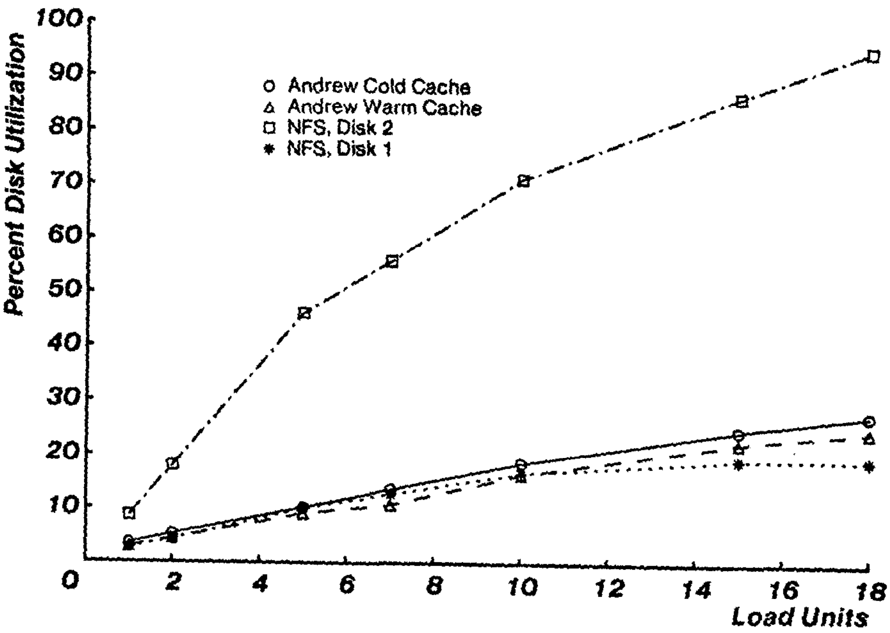

Fig. 5. NFS and Andrew server disk utilization. This figure compares the server disk utilizations of NFS and Andrew as a function of load. Table XII presents these data in greater detail and describes the conditions under which they were obtained.

Table XIII. Network Traffic for Andrew and NFS

	Andrew	NFS
Total packets	3,824 (463)	10,225 (106)
Packets from Server to Client	2,003 (279)	6,490 (86)
Packets from Client to Server	1,818 (189)	3,735 (23)

Notes: This table presents the observed network traffic generated by the benchmark when a single client was using a server. In the Andrew case, the server was a Sun2 on an Ethernet connected via a router to an IBM RT client on a token ring. In the NFS case, the server was a Sun2 on the same Ethernet cable as its Sun 3 client. Each of the experiments was repeated 3 times. Figures in parentheses are standard deviations.

Data on server disk utilization are presented in Figure 5 and Table XII. NFS used both disks on the server, with utilizations rising from about 9 and 3 percent at a load of 1 to nearly 95 and 19 percent at a load of 18. Andrew used only one of the server disks, with utilization rising from about 4 percent at a load of 1 to about 33 percent at a load of 18 in the cold cache case. Disk utilization is slightly, but not substantially, lower in the warm cache case.

Another quantity of interest is the relative amount of network traffic generated by NFS and Andrew during the execution of the benchmark. Table XIII presents this information. As the table indicates, NFS generates nearly three times as many packets as Andrew at a load of one.

Table XIV. Latency of NFS and Andrew

File size	Time (milliseconds)			
(bytes)	Andrew cold	Andrew warm	NFS	Stand-alone
3	160.0 (34.6)	16.1 (0.5)	15.7 (0.1)	5.1 (0.1)
1,113	148.0 (17.9)			
4,334	202.9 (29.3)			
10,278	310.0 (53.5)			
24,576	515.0 (142.0)		15.9 (0.9)	

Notes: This table shows the latency in milliseconds as a function of file size. Latency is defined here as the total time to open a file, read one byte, and then close the file. A Sun3 server and a single Sun3 client were used in all cases. In the Andrew warm cache case, the file being accessed was already in the cache. The cold cache numbers correspond to cases in which the file had to be fetched from the server. The figures in parentheses are standard deviations.

Low latency is an obvious advantage of remote-open file systems. To quantify this fact, we ran a series of experiments that opened a file, read the first byte, and then closed it. Table XIV illustrates the effect of file size on latency in NFS and Andrew. Latency is independent of file size in NFS and is about thrice that of a local file. In Andrew, when the file is in the cache, latency is close to that of NFS. When the file is not in the cache, latency increases with file size. In interpreting Andrew data it is important to note that the close-system call completes before Venus transfers the file to the server.

What can we conclude from these observations? First, it is clear that Andrew's scaling characteristics are superior to those of NFS. Second, the improved scaling of Andrew is not achieved at the price of substantially poorer small-scale performance. Andrew is implemented almost entirely in user space, whereas NFS is entirely in the kernel. We anticipate a significant reduction in overhead if we move Andrew code into the kernel. There is thus untapped potential for improved performance in Andrew, whereas we see no similar potential in NFS. Finally, Andrew provides a well-defined consistency semantics, as well as support for security and operability. We are pleased to observe that such additional functionality has been incorporated without detriment to our primary goal of scalability.

6. CHANGES FOR OPERABILITY

As the scale of a system grows, its users become increasingly dependent on it and operability assumes major significance. Since the prototpye paid scant attention to operability, it was imperative that we address this aspect of the system seriously in the redesign. Our goal was to build a system that would be easy for a small operational staff to run and monitor with minimal inconvenience to users.

At the heart of the operability problems in the prototype was an inflexible mapping of Vice files to server disk storage. This mapping, described in Section 2.1, was deficient in a number of ways:

(1) Vice was constructed out of collections of files glued together by the 4.2BSD *Mount* mechanism. Only entire disk partitions could be mounted.

Consequently, only sets of files on different disk partitions could be independently located in Vice. To minimize internal fragmentation on the disks, such partitions had to be quite large, typically consisting of the files of ten or more users. The fact that repartitioning of a disk had to be done off-line further reduced flexibility.

(2) The embedding of file location information in the file storage structure made movement of files across servers difficult. It required structural modifications to storage on the servers, and modifications to the files while the move was in progress were sometimes lost.

(3) It was not possible to implement a quota system, which we believe to be important in a system with a large number of users.

(4) The mechanisms for file location and for file replication were cumbersome because of the lack of well-defined consistency guarantees. The embedded location database was often wrong, and failures during the propagation of replicated files sometimes left it inconsistent.

(5) Standard utilities were used to create backup copies of the files in the system. Although these utilities are adequate for a single-site system, they are not convenient for use in a distributed environment, where files may have been moved since they were last backed up. The wiring-in of location information made restoration of files particularly difficult.

(6) Backup was further complicated by the fact that a consistent snapshot of a user's files could not be made unless the entire disk partition containing those files was taken off-line. We felt that this factor was an unacceptable imposition on users.

To address these problems, our redesign uses a data structuring primitive called a *Volume* [8]. In the rest of this section we describe volumes and show how they have improved the operability of the system.

6.1 Volumes

A *Volume* is a collection of files forming a partial subtree of the Vice name space. Volumes are glued together at *Mount Points* to form the complete name space. A mount point is a leaf node of a volume that specifies the name of another volume whose root directory is attached at that node. Mount points are not visible in pathnames; Venus transparently recognizes and crosses mount points during name resolution. The mount mechanism in Vice is thus conceptually similar to the standard 4.2BSD mount mechanism.

A volume resides within a single disk partition on a server and may grow or shrink in size. Volume sizes are usually small enough to allow many volumes per partition. We have found it convenient to associate a separate volume with each user. As mentioned in Section 3.2, volume-to-server mapping information is maintained in a volume location database replicated at all servers.

6.2 Volume Movement

Balancing of the available disk space and utilization on servers is accomplished by redistributing volumes among the available partitions on one or more servers. When a volume is moved, the volume location database is updated. The update does not have to be synchronous at all servers since temporary forwarding information is left with the original server after a move. It is thus always possible for a workstation to identify the server responsible for a volume. A volume may be used, even for update, while it is being moved.

The actual movement is accomplished by creating a frozen copy-on-write snapshot of the volume called a *Clone*, constructing a machine-independent representation of the clone, shipping it to the new site, and regenerating the volume at the remote site. During this process the volume may be updated at the original site. If the volume does change, the procedure is repeated with an incremental clone by shipping only those files that have changed. Finally the volume is briefly disabled, the last incremental changes shipped, the volume made available at the new site, and requests directed there. The volume move operation is atomic; if either server crashes the operation is aborted.

6.3 Quotas

Quotas are implemented in this system on a per volume basis. Each user of the system is assigned a volume, and each volume is assigned a quota. The responsibility for managing the allocated space within a volume is left to the user. Access lists permitting, a user may store files in a volume belonging to another user. However, it is always the owner of a volume who is charged for its usage. System administrators can change quotas easily on volumes after they are created.

6.4 Read-Only Replication

Executable files corresponding to system programs and files in the upper levels of the Vice name space are frequently read but seldom updated. Read-only replication of these files at multiple servers improves availability and balances load. No callbacks are needed on such files, thereby making access more efficient. Read-only replication is supported at the granularity of an entire volume. The volume location database specifies the server containing the read-write copy of a volume and a list of read-only replication sites.

As described in Section 6.2, a read-only clone of a volume can be created and propagated efficiently to the replication sites. Since volume propagation is an atomic operation, mutual consistency of files within a read-only volume is guaranteed at all replication sites. However, there may be some period of time during which certain replication sites have an old copy of the volume while others have the new copy.

Read-only volumes are 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. Any one of a collection of servers with identical sets of read-only volumes (and no read-write volumes) can be introduced or withdrawn from service with virtually no impact on users. This provides an additional measure of availability and serviceability.

6.5 Backup

Volumes form the basis of the backup and restoration mechanism in our redesign. 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. The staging software is not aware of the internal structure of volumes but merely dumps and restores them in their entirety. Volumes can be restored to any server, since there is no server-specific information embedded in a volume.

Experience has shown that a large fraction of file restore requests arise from accidental deletion by users. To handle this common special case, the cloned read-only backup volume of each user's files is made available as a read-only subtree in that user's home directory. Restoration of files within a 24-hour period can thus be performed by users themselves using normal file operations. Since cloning uses copy-on-write to conserve disk storage, this convenient backup strategy is achieved at modest expense.

6.6 Summary

Our experience with volumes as a data structuring mechanism has been entirely positive. Volumes provide a level of *Operational Transparency* that is not supported by any other file system we are aware of. From an operational standpoint, the system is a flat space of named volumes. The file system hierarchy is constructed out of volumes but is orthogonal to it.

The ability to associate disk usage quotas with volumes and the ease with which volumes may be moved between servers have proved to be of considerable value in actual operation of the system. The backup mechanism is simple and efficient and seldom disrupts normal user activities. These observations lead us to conclude that the volume abstraction, or something similar to it, is indispensible in a large distributed file system.

7. CONCLUSION

Scale impacts Andrew in areas besides performance and operability. The large number of users and workstations in the system has resulted in sizable authentication and network databases. As the system grows, the existing mechanisms to update and query these databases will become inadequate. Fault-tolerance is another area in which scaling stresses Andrew. The access of a Vice file can, in the worst case, involve multiple servers and network elements. Every one of these components has to be up for the file access to succeed. Read-only replication of system files alleviates this problem to a certain extent but does not entirely solve it. Although a uniform, location-transparent file name space is a major conceptual simplification, the failure modes that arise can be quite difficult for a naive user to comprehend. The issue of software version control and orderly release of critical software to workstations will also increase in importance as the system grows in size.

In choosing to focus on scale, we have omitted discussion of many other important aspects of the evolution of the Andrew File System. Security and emulation of UNIX semantics, for example, are two areas fundamental to the file system. Network topology, hardware, and software are other such examples. We have had to pay close attention to these and other similar areas in the course of our design and implementation.

At the time this paper was written, in early 1987, there were about 400 workstations and 16 servers. About a fifth of the workstations were in public terminal rooms. There were over 3500 registered users of the system, of whom over 1000 used Andrew regularly. The data stored on the servers was approximately 6000 megabytes and was distributed over about 4000 volumes. Although Andrew is not the sole computing facility at CMU, it is used as the primary computational environment of many courses and research projects.

What do we see for the future? Usage experience gives us confidence that this system will scale with minimal changes to about 500 to 700 workstations. From there to our eventual goal of 5000 workstations is, of course, a large gap. Although the performance data presented in this paper confirms that our high-level architecture is appropriate for scaling, it is inevitable that significant changes will have to be made with each quantum increase in the size of the system.

The changes we have thought of address a variety of issues. Moving Venus and the server code into the kernel would improve performance considerably. Changing the kernel intercept mechanism to an industry standard would simplify the maintenance and portability of the system. The ability to allow users to define their own protection groups would simplify administration. As users become more dependent on the system, availability becomes increasingly important. Some form of replication of writable files will be necessary eventually. The distributed nature of the system and its inherent complexity make it a difficult system to troubleshoot. Monitoring, fault isolation, and diagnostic tools that span all levels of the hardware and software will become increasingly important. Finally, as the system grows, decentralized administration and physical dispersal of servers will be necessary.

In conclusion, we look upon the present state of the Andrew File System with satisfaction. We are pleased with its current performance and with the fact that it compares favorably with the most prominent alternative distributed file system. At the same time, we are certain that further growth will stress our skill, patience, and ingenuity.

ACKNOWLEDGMENTS

The Andrew File System was developed at the Information Technology Center of Carnegie-Mellon University. We wish to thank our colleagues for their many contributions to this project and our users for their patience in dealing with a system under development. We wish to express our special appreciation to Vasilis Apostolides who assisted us in running the benchmarks that compared Andrew and NFS. The suggestions of James Kistler, Richard Snodgrass, James Peterson, and the SOSP referees improved this paper in many ways.

REFERENCES

1. BROWNBRIDGE, D. R., MARSHALL, L. F., AND RANDELL, B. The Newcastle connection. *Softw. Pract. Exper. 12* (1982), 1147-1162.
2. KAZAR, M. L. Synchronization and caching issues in the Andrew file system. Tech. Rep. CMU-ITC-058, Information Technology Center, Carnegie Mellon Univ., Pittsburgh, Pa., June, 1987.
3. MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S., AND SMITH, F. D. Andrew: A distributed personal computing environment. *Commun. ACM 29*, 3 (Mar. 1986), 184-201.
4. OUSTERHOUT, J., DA COSTA, H., HARRISON, D., KUNZE, J., KUPFER, M., AND THOMPSON, J. A trace-driven analysis of the UNIX 4.2BSD file system. In *Proceedings of the 10th ACM Symposium*

on Operating System Principles (Orcas Island, Wash., Dec. 1-4, 1985). ACM, New York, 1985, pp. 15–24.

5. RIFKIN, A. P., FORBES, M. P., HAMILTON, R. L., SABRIO, M., SHAH, S., AND YUEH, K. RFS architectural overview. In *Usenix Conference Proceedings* (Atlanta, Ga., Summer 1986). USENIX Association, Berkeley, Calif., pp 248–259.
6. SATYANARAYANAN, M., HOWARD, J. H., NICHOLS, D. A., SIDEBOTHAM, R. N., SPECTOR, A. Z., AND WEST, M. J. The ITC distributed file system: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating System Principles* (Orcas Island, Wash., Dec. 1-4, 1985). ACM, New York, 1985, pp. 35–50.
7. SCHROEDER, M. D., GIFFORD, D. K., AND NEEDHAM, R. M. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating System Principles* (Orcas Island, Wash., Dec. 1-4, 1985). ACM, New York, 1985, pp. 25–34.
8. SIDEBOTHAM, R. N. Volumes: The Andrew file system data structuring primitive. *European Unix User Group Conference Proceedings* (August 1986). Also available as Tech. Rep. CMU-ITC-053, Information Technology Center, Carnegie Mellon Univ., Pittsburgh, Pa.
9. SUN MICROSYSTEMS, INC. *Networking on the SUN Workstation.* Sun Microsystems, Mountain View, Calif., 1986.
10. SVOBODOVA, L. File servers for network-based distributed systems. *Comput. Surv. 16*, 4 (Dec. 1984), 353–398.
11. TICHY, W. F., AND RUAN, Z. Towards a distributed file system. Tech. Rep. CSD-TR-480, Computer Science Dept., Purdue Univ., West Lafayette, In. 1984.
12. WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating System Principles* (Bretton Woods, N.H., Oct. 11–13, 1983). ACM, New York, 1983, pp. 49–70.

Received May 1987; Revised August 1987; accepted September 1987.