CMU-ITC-85-040

The ITC Distributed File System: Prototype and Experience

Michael West and David Nichols
John Howard
M. Satyanarayanan
Robert Sidebotham

Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

Draft: 30 March 85 14:58

Abstract

In this paper we describe a prototype of the ITC distributed file system, a design intended to span a network of thousands of workstations at Carnegie-Mellon University. Key features of the design are the use of whole file transfer and caching. The objective of the prototype is to evaluate the ability of the high-level design to meet the the goals of location transparency, user mobility, and application code compatibility.

A user community of about 75 software developers and support personnel has been using a prototype of this design for 9 months, on a system with 50 workstations and 2 servers. Both the servers and the workstations run Unix. Minor kernel modifications have been made on the workstations to intercept file system calls, but the cache management code is entirely outside the kernel. No kernel modifications have been made for the servers.

The goals of location transparency and mobility have been met. Compatibility with Unix has been met to a high degree: existing Unix application programs run without relinking or recompilation. We have experienced some minor incompatibilities in the areas of links and renaming directories.

Although the system is quite usable, it is noticeably slower than a stand-alone workstation: benchmarks indicate a degradation of about 80% in total elapsed time. The bottleneck appears to be CPU utilization on the servers.

Experience with this prototype has been used in redesigning certain aspects of the system. The changes mainly address the issue of performance, operability, and compatibility with Unix.

This project was funded by the IBM Corporation.

Table of Contents

1. Overview of Goals and Design

This paper presents our experience in building a distributed file system (Vice) and the workstation attachment (Venus) to interface with it. A separate paper [2] describes the design in detail, including justification for the decisions made and a survey of related work. Here we concentrate on implementation and experience to date, including only enough design information to make the paper self-contained.

Vice is intended to be shared by thousands of workstations on the Carnegie-Mellon University campus. It provides an integrated Unix-like file naming hierarchy, using a network of cooperating file servers; users' workstations (running Unix) attach to it by way of a high-performance local area network. At present Vice is used by several hundred individuals and about 100 workstations; over the next year we hope to expand it to handle several hundred workstations, with at least occasional use by anybody on campus.

The current version of Vice is a prototype, intended primarily to verify the following fundamental design concepts:

Whole File Transfer
: Workstations read and write entire files from file servers rather than pages or records. What effect does this have on performance? Are large files sufficiently rare?

File Caching
: Workstations cache files on their local disks. Are these disks large enough to cache a typical working set of files? How well do our cache management algorithms work?

Unix Semantics
: How well can we emulate Unix and retain the benefits of centralized timesharing systems in a distributed environment? Does whole file transfer conflict with our desire to run Unix application programs without modification?

Server Load
: Vice is implemented with multiple cooperating servers in order to grow with the user community. How many workstations can a single server support? Can the load be balanced across servers?

The rest of this paper presents key implementation decisions and our experience with the prototype system. Section 2 describes many of the important implementation details. Section 3 gives the current status of the system. In Section 4 we give measurements of the performance of the prototype. Section 5 describes the answers we discovered to our design questions and other experience we gained from the effort. Section 6 describes changes we are making to the next version based on our experience from the prototype. Finally, Section 7 contains a brief summary of the paper.

2. General Implementation

The major components of the implementation include the file server and related programs (Vice) and the workstation cache manager (Venus). They communicate using a remote procedure call package (RPC) that is linked into each program. These programs run on the Berkeley 4.2BSD release of Unix [1], which we have modified to allow Venus to intercept file system calls on the workstation.

Since our primary goal in the prototype was to test our design, we traded performance for function whenever it would affect development time. Often this meant using off-the-shelf software. For example, the file server itself is based on Unix and depends heavily on the Unix file system; the RPC package uses 4.2BSD IP sockets.

2.1. Vice/Venus Interface

Since Vice is based on whole file transfer, the interface between Vice and Venus is very simple. It has two basic types of objects: *files* are uninterpreted byte streams, and *directories* are lists of files and other directories. Objects are named in the interface with their full pathname.

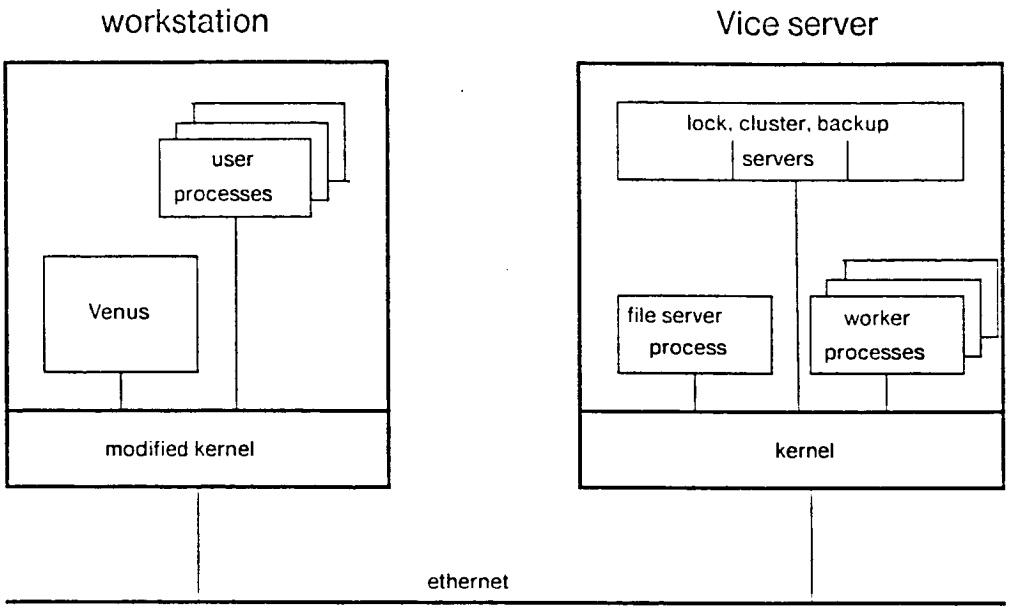

Figure 2-1: Vice/Venus Overview

The operations provided by the Vice interface allow Venus to:

- *Fetch, store and remove files.* The entire file is shipped on each fetch or store.
- *Make and remove directories.* Venus cannot directly store a directory; directories are updated as side effects of other operations.
- *Get and set status information.* For both files and directories.
- *Check access rights and file currency.* Access rights are checked for a particular user relative to a particular file or directory. The same operation also returns the last change date for the file or directory.
- *Acquire and release locks.* To implement Unix advisory locking.

2.2. Vice

Vice is distributed over several server machines. On each of these machines, there are a number of independent Unix processes that implement the server function by fielding requests and performing updates on the local Unix file system.

2.2.1. Data Storage

A Vice file server stores its data in the server machine's Unix file system. The Vice directory hierarchy is represented by an identical Unix hierarchy, plus additional files and directories used to store status information.

There are two types of file storage within Vice. *Normal files* are stored at one server only. *Replicated files* are stored at all servers. Venus can fetch replicated files from any server. Each file has a single server, its *custodian*, to which all stores must be directed. When replicated files are updated, the custodian is responsible for copying the changes to all other servers. The updating is done through an asynchronous process, and while the updating is going on the non-custodian servers have an old copy of the file. Replicated files are

primarily used for system binaries.

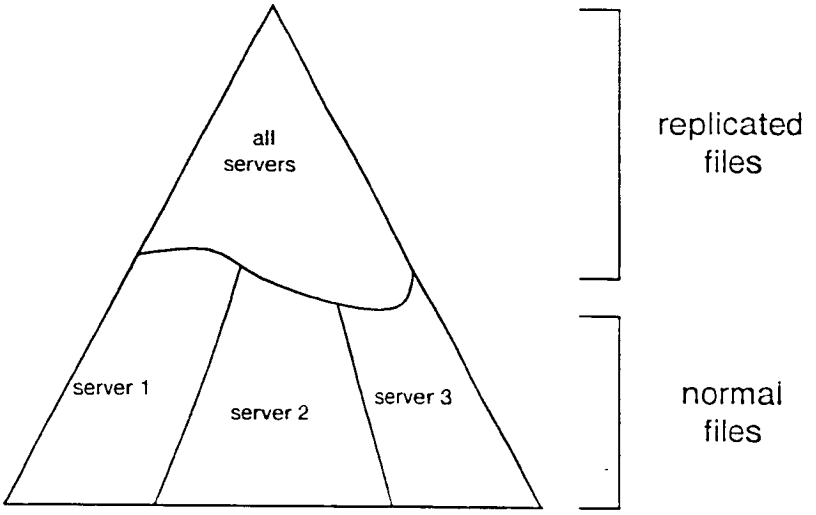

Figure 2-2: File Distribution

A replicated directory must have a replicated parent. This results in a tree that is replicated down to a frontier in the hierarchy, and is normal from there on down. Note that the subtree extending from a normal directory has a single custodian, and a server stores a file (or directory) only if it stores its parent.

The location database, that tells which server is the custodian for each file, is embedded in the file tree. A server finds the custodian for a file by searching the replicated part of the tree until it finds either the file or a normal (non-replicated) directory whose custodian is not the server doing the search. Stub directories are maintained on all servers for the top level of non-replicated subtrees in order to point to the custodians of the subtrees. The server returns the name of the custodian and the name of the root of the normal subtree that contains the file. Venus keeps this information in a cache which it uses to guide subsequent requests to the proper server.

If a server is down, non-replicated files on that server can neither be read nor updated. If the custodian server for a replicated file is down, the file can be read from other servers but not updated.

Status information is stored in Vice in a set of shadow files. For each directory stored in Vice there is an associated *.admin directory*. Within that .admin directory is a *.admin file* that contains status information for the directory. Each file in the stored directory also has a corresponding file in the .admin directory that contains status information for the file.

Vice stores status information about files organized as a list of named properties. The information stored includes:

Type	Replicated (stored at all servers) or normal (stored only at one server). This is stored for directories only.
Location	The custodian for the file. This is stored for directories only.
Dates	The time at which the file last stored and the time that the status was last modified.
Access List	The access list which controls access to the files. This is stored for directories only.

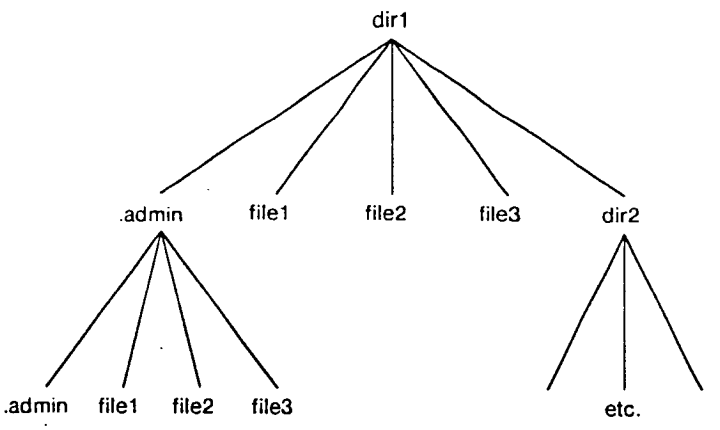

Figure 2-3: Status Information

Owner	The name of the owner of the directory. This is stored for directories only. The owner of a directory can always change the access list.
User Defined	Clients are allowed to define named properties and store them with the file. Venus uses this for the execute and setuid bits.

Since some of the properties are stored only with directories, all files in a directory have those properties in common. Specifically, all of the files in a directory have the same custodian, type, owner, and access list.

We chose this representation to avoid writing our own underlying file system. In addition to allowing us to use the storage and naming mechanism, it made available utility programs such as tape backup/restore and the file system consistency checker. Also, we have access to the files even if our file server fails. However, this choice imposes on us various restrictions in the way we can share files, and it leads to noticeable CPU overheads, for example, in looking up file names.

2.2.2. Process Structure

When a server machine is brought up several processes are started to handle the requests to the server. The primary process is the *file server process*, which listens for connecting workstations. When a workstation connects, the file server process forks a *worker process* which handles all requests for a particular user from that workstation. All requests from that user on that workstation are routed to this single worker process.

The *lock process* is used to handle any lock requests. Unix does not allow memory to be shared between processes, so we use a separate process in order to keep the locks in memory. Lock requests must be directed to the custodian of the file being locked.

The *backup* and *cluster process* are used to keep replicated files up-to-date on all servers. Worker processes write entries to a *backup queue* whenever a replicated file is modified. The backup process examines the queue and sends requests to update the file to the cluster processes on the other servers. The cluster process is like a worker process, but has a simpler interface and bypasses normal checking of requests.

2.3. Venus

Venus is implemented as a user-level process running on a workstation with a modified Unix kernel. In response to file system calls made by client programs, Venus caches files from the servers and makes the files in its cache available to the client programs. In order to do this, Venus must locate the files among the

servers, connecting to servers as needed on behalf of the client.

2.3.1. Kernel interface

From the point of view of an application running on the workstation, the Vice file system appears as a subtree in the name space. The workstation kernel intercepts file system calls directed to files in that subtree and routes them to Venus. When it has performed the system call, Venus sends a reply via the kernel to the original calling process.

Unix uses a special kind of file system object known as a *device file* to implement device drivers. This mechanism allows the kernel implementor to provide special routines to handle reads and writes to the device file. Venus uses such a file to communicate with the kernel. Whenever Venus reads from the device file, the kernel delivers the parameters for an intercepted file system call. Venus replies to the call by writing to the device file.

When a program opens a remote file, Venus first verifies that the program making the request is allowed to open the file. It then checks that the file is in the cache and up-to-date, fetching a new copy if necessary. Venus sends the name of the local copy of the file to the kernel. The kernel opens the file in the cache and gives the requesting program a file descriptor for the local copy. Read, writes, and seeks applied to this descriptor are not trapped by the kernel, so they run at full speed.

Close requests are trapped and sent to Venus. If the file was open for writing, Venus stores it back to the file server.

Other calls are simpler: Venus receives the request, does some work to perform the operation, and sends a reply back to the kernel. For example, a *utimes* system call, which sets the modified date for the file, is translated into the Vice call that sets the file's status information.

2.3.2. The cache

The cache is maintained with an LRU replacement algorithm. The current implementation limits the cache size by the number of files in the cache; a later version will use the disk space available on the local disk to control the cache size.

Each entry is marked with the version number of the file that was fetched to create it. When a program opens a file that is in the cache, Venus must check that the cache entry is still valid. In addition, Venus checks that the user making the request is authorized to open the file, for it may have been fetched into the cache by another user. These checks are combined into a single file server operation (*TestAuth*).

If a file is open for writing, Venus satisfies requests concerning the file from the version being modified in the cache. For example, if a program is writing a file and requests its length, this information is taken from the version being written instead of from the version on the server.

Unix supports fine-grain sharing of file data between processes on a single workstation. If a process writes ten bytes of data to a file, then the new data is immediately visible to any other process that attempts to read it. Because our server only supports the transfer of entire files, we cannot support this degree of sharing between processes on different machines.

Fine-grain sharing between processes on the same machine is supported by having the processes share cache entries. If a program is writing a file and another opens it, they will wind up with file descriptors onto the same file in the cache. Processes that are related via forks are guaranteed to get fine-grain sharing because they must be on the same machine. Also, a user can watch the progress of a program that is writing a large output file.

Applications that attempt to synchronize unrelated processes (e.g. two users running a database program from different workstations) cannot use the fine-grain sharing. The applications *can* share data on an open/close

granularity instead of read/write granularity.

We have not yet encountered any applications that are affected by this difference, but it may become a problem later as more database applications are developed for our system.

2.3.3. Locating files

Whenever Venus sends a request to the wrong server, it receives in the reply the name of a subtree that contains the file it needs and the custodian for that subtree. Venus uses the custodian name to continue its search for the file.

Venus remembers the names of the subtrees and uses this information to make first guesses as to where to find files in subsequent requests. When Venus needs to direct a file operation to some server, it looks in the location table for the longest prefix of the pathname named in the operation. This prefix gives the deepest spot in the file hierarchy along the path to this file for which a server is known. This server is used as the first guess for the real location of the file.

2.3.4. Connection management

Since authentication applies to RPC connections instead of to individual remote procedure calls, Venus must maintain separate connections to the file servers for each user on the workstation. The current implementation of RPC restricts the number of active connections, so Venus maintains a table of active connections, making and breaking connections to file servers as needed, using an LRU replacement algorithm. Venus closes connections that are idle for 30 minutes to conserve server resources.

2.4. Other

2.4.1. RPC

The RPC package provides a high level of communication between Vice and Virtue based on a client-server model using remote procedure calls for transfer of data and control. The RPC subroutine package has been implemented on top of the Internet protocols. The distinctive features of the RPC package are:

- *The transfer of files as side effects of remote procedure calls.* This capability is used extensively in the file system for transferring files to and from the workstations. A separate channel is used for file transfer to allow for optimization by the system for the differences between file and control transfers.
- *Built-in authentication facilities* which allow two mutually suspicious parties to exchange credentials via a three way encrypted handshake.
- *Optional use of encryption* for secure communication, using session keys generated during the authentication handshake.

The server half of the RPC package supports the file server's process structure by automatically forking a worker process for each new connection. The fork can be suppressed for new connections, but the server process must then disconnect from the client before serving a new client.

2.4.2. Protection

We felt that standard Unix protection would not be sufficient to handle a group of users as large as the CMU campus. To allow for this, a more elaborate protection mechanism was devised. This allows users or groups to be given access to directories. *Users* are accountable entities within the file system and *groups* are collections of users or other groups. An *access list* is stored with each directory that maps users or groups to *access rights* to that directory. The access rights that are supported are:

Read	Allows a file to be fetched.
Write	Allows an existing file to be stored.
Insert	Allows a new file or subdirectory to be created.
Delete	Allows an existing file or subdirectory to be removed.
Lookup	Allows the names of the members of a directory to be retrieved.
Lock	Allows a Lock call to be made.
Administer	Allows the access list to be changed.

These access lists allow better control of who can do what to files than the Unix protection system. However, since they are associated only with directories and not with files, all files in a given directory must have the same level of control.

2.4.3. Authentication

RPC implements a three way handshake authentication protocol and the ability to encrypt traffic on the ethernet. Initially, however, we have just used the userid on the workstation to establish the connection to Vice. There is currently no check in Vice to ensure that the user is who he claims to be; we intend to capture the login password and use it to authenticate connections.

3. Status

Our servers run on SUN 170s and VAX 11/750s with four megabytes of memory and two 400 megabyte disk drives each. The workstations are SUN 120s and SUN 100s with two megabytes of memory and 70 megabyte local disks. SUNs are based on the Motorola 68010 processor chip.

By using standard facilities and by using machines that store data in different byte orders, we have tried to keep our code machine independent so that we can port it to other machines easily.

As of March 1985 we have two file systems running. One is our internal system and consists of two servers with about 50 workstations and 75 users and has been in operation for nine months. The other is used outside the ITC and consists of four servers, with 50 workstations attached exclusively to it in addition to the internal workstations has been in operation about two months. About 250 users are authorized to use the external system. We use Ethernets with fiber optic links between gateways to connect the workstations and servers. The internal system has about a gigabyte of data stored in it, and the external system has about 1.5 gigabytes of data.

4. Performance

In this section, we discuss a number of performance-related issues pertinent to our implementation. One of the most important of these issues is the ratio of remote to local file access times. Also important are the effects of whole-file transfer and caching on performance. We are also interested in knowing how many users can be reasonably assigned to a server, whether we are balancing the load on our servers evenly, and on the factors which currently limit performance.

A user who is accustomed to a stand-alone workstation perceives some qualitative performance differences when he uses a Virtue workstation. Section 4.1 describe these differences. We then present quantitative results in Section 4.2, based on actual measurements of the system.

4.1. Qualitative Observations

Although command execution is noticeably slower in Virtue than on a stand-alone workstation, the performance is often better than on the timesharing systems used by the general campus user community at CMU. Performance degradation is not uniform across all operations. Some operations, like the compilation of a large program, proceed at almost stand-alone speed. Other operations, such as a recursive search of a subtree of files, take much longer when the subtree is in Vice.

Certain programs run much slower than we had originally expected, even when all relevant files are in the local cache. This is because such programs obtain status information about files (using the *stat* primitive in Unix), before actually opening them. Since each *stat* call either involves a cache miss or a cache validity check, the total number of client-server interactions is significantly higher than the number of file opens. This increases both the total running time of these programs and the load on the servers.

The whole-file transfer approach contributes significantly to good performance during many frequent user operations such as program compilation. Execution proceeds at the same speed as on a stand-alone system except for an initial delay to fetch the compiler binary and the file being compiled (or to validate their cache entries), and to store the generated code back into Vice. Another common user operation is the editing of files. The editors in use in our environment read the entire file being edited into virtual memory prior to using them. Whole-file transfer neither improves nor hurts performance in this case.

We find that performance is usually acceptable up to a limit of about 25 active users per server. However, there have been occasions when even a few users intensely using the file system have caused performance to degrade intolerably.

4.2. Measurements

An obvious quantity of interest in a caching file system is the hit ratio observed during actual use. Venus uses two caches: one for files and the other for status information about files. A snapshot of the caches of 12 machines in our environment shows an average file cache hit ratio of 81%, with a standard deviation of 9.8%, and an average status cache hit ratio of 82%, with a stand deviation of 12.9%.

Also of interest is the relative distribution of individual Vice calls. Such a profile is valuable in improving server performance, since attention can be focussed on the most frequent calls. Table 4-1 shows the observed distribution of each Vice call which accounts for more than one percent of the total. This data was gathered over a one-month period on five cluster servers. The distribution is dramatically skewed, with two calls accounting for nearly 90% of the total. The *TestAuth* call is used to validate check entries, while *GetFileStat* is used to obtain status information about files absent from the cache. The table also shows that only 6% of the calls to Vice (*Fetch* and *Store*) actually involve file transfer, and that the ratio of *Fetch* calls to *Store* calls is approximately 2:1.

In order to investigate the performance penalty caused by Vice and Venus, we performed a series of controlled experiments using a benchmark. This benchmark stresses the file system far more intensely than a typical user and involves a series of file copy operations, directory and file scans, and a large compilation. Table 4-2 presents the total running time for the benchmark as a function of the number of clients simultaneously executing that benchmark. 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 takes about 80% longer in the 1 client/server case than in the standalone case. A second observation is that the time for *TestAuth* rises rapidly beyond a load of 5 clients/server, indicating server saturation. For this benchmark, therefore, a client-server ratio between 5 and 10 is the maximum feasible.

For measuring server usage, we have installed software on servers to maintain statistics about CPU and disk utilization, and about data transfers to and from the disks. Table 4-3 presents this data for four servers over a

Server	Total Calls	Call Distribution						
		TestAuth	GetFileStat	Fetch	Store	SetFileStat	GetMembers	All Others
cluster0	1625954	64.2%	28.7%	3.4%	1.4%	0.8%	0.6%	0.9%
cluster1	564981	64.5%	22.7%	3.1%	3.5%	2.8%	1.3%	2.1%
cmu-0	281482	50.7%	33.5%	6.6%	1.9%	1.5%	3.6%	2.2%
cmu-1	1527960	61.1%	29.6%	3.8%	1.1%	1.4%	1.8%	1.2%
cmu-2	318610	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)

NOTE:

Figures in parentheses are standard deviations.
The data shown here was gathered over a one-month period.

Table 4-1: Observed Distribution of Vice Calls

two-week period. The data is restricted to observations made during 9am to 5pm on weekdays, since this is the period of most intense system use. As the CPU utilizations in the table show, the servers are not evenly balanced. This fact is independently confirmed by Table 4-1, which shows a spread of about 5:1 in the total number of Vice calls presented to each server. Moving users to less heavily loaded servers is possible, but relatively cumbersome at the present time.

Table 4-3 also reveals that the two most heavily used servers show an average CPU utilization of about 40%. This is a very high figure, considering that it is an average over an 8-hour period. Closer examination of the raw data shows much higher short-term CPU utilization: figures in the neighborhood of 75% over a 5-minute averaging period are common. Disk utilizations, however, are much lower. The 8-hour average is less than 15%, and the short-term peaks are rarely above 20%. We conclude from these figures, and from server utilization data obtained during the benchmarks, that the current performance bottleneck is the server CPU. Based on profiling of the servers, we are led to believe that the two factors chiefly responsible for this high CPU utilization are 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 presented in this section indicate that significant performance improvements are 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. Section 6 discusses the specific ways in which we intend to incorporate these changes.

Configuration	Overall Benchmark Time		Time per TestAuth Call	
	Absolute (s)	Relative	Absolute (ms)	Relative
Stand-Alone	998 (9)	100%	NA	NA
1 client/server	1789 (3)	179%	87 (0)	100%
2 clients/server	1894 (4)	190%	118 (1)	136%
5 clients/server	2747 (48)	275%	259 (16)	298%
8 clients/server	5129 (177)	514%	670 (23)	770%
10 clients/server	7326 (69)	734%	1050 (13)	1207%

NOTE:

Figures in parentheses are standard deviations.
Each client had a 300-entry cache.
Each data point is the mean of 3 independent replications.

Table 4-2: Stand-Alone versus Remote Access

5. Evaluation

This section presents our observations of the prototype along dimensions other than performance. We first discuss the extent to which our high-level design decisions were validated by the prototype. We then present additional observations that bear on the detailed design decisions. The material presented this section motivates the changes proposed in Section 6.

5.1. High-Level Design

5.1.1. Whole file transfer

Transferring entire files to and from the workstation contributes greatly to the success of our prototype. Despite the relatively slow performance of our servers, the overall system is quite usable because reads and writes are performed locally.

Server	Samples	CPU Utilization			Disk 1			Disk 2		
		total	user	system	util	KBytes	xfers	util	KBytes	xfers
cluster0	13	37.8% (12.5)	9.6% (4.4)	28.2% (8.4)	12.0% (3.3)	380058 (84330)	132804 (35796)	6.8% (4.2)	186017 (104682)	75212 (42972)
cluster1	14	12.6% (4.0)	2.5% (1.1)	10.1% (3.4)	4.1% (1.3)	159336 (41503)	45127 (21262)	4.4% (2.1)	168137 (63927)	49034 (32168)
cmu-0	15	7.0% (2.5)	1.8% (0.7)	5.1% (1.8)	2.5% (0.9)	106820 (41048)	28177 (10289)			
cmu-1	14	43.2% (10.0)	7.2% (1.8)	36.0% (8.7)	13.9% (4.5)	478059 (151755)	126257 (42409)	15.1 (5.4)	373526 (105846)	140516 (40464)

NOTE:

Figures in parentheses are standard deviations
Peak period is defined as 9am to 5pm on weekdays.
The data shown here was gathered over a two-week period.

Table 4-3: Server Usage During Peak Period

With the current system's performance, we feel quite comfortable handing files up to about a megabyte. We have rarely encountered larger files in day-to-day usage.

5.1.2. File caching

The default file cache for Venus is 300 files. With this cache size, we achieve an average cache hit rate of over 80%. The cache normally fits into about ten megabytes of the 30 megabytes of disk our workstations allocate for the cache. This disk allocation has been generous enough that we have not yet devoted the effort to converting Venus to use a disk space limited cache.

5.1.3. Unix semantics

The cache manager provides an interface to the file system that is highly compatible with Unix. We run standard Unix applications without modification. The high-level design decisions do not interfere with our ability to emulate Unix. The difference in file-sharing granularity has not proved to be a problem in practice. However, as described in Sections 5.2.3 and 5.2.4, certain decisions made in the prototype implementation precluded the renaming of directories and the use of symbolic links.

5.1.4. Server load

The current prototype can handle 15 to 25 workstations per server while providing acceptable performance. Our measurements indicate, and experience bears out, that the servers can become substantially slower if a few workstations are doing file system intensive work.

Replication of system files will allow us to balance server load. The current imbalance reported in Section 4.2 is due both to a bug in the implementation of the file location algorithm and to the difficulty of reassigning users to servers in the current system.

5.2. Detailed Design

In this section we discuss certains effects of decisions made in the prototype implementation. These observations do not reflect adversely on the high-level design, but indicate changes that need to be made in building a more usable system.

5.2.1. Operation

The current system is difficult to operate and maintain. The two main problems are an inability to reassign users to servers easily and the lack of convenient facilities for backup and restoration of files.

The fact that the file system does not tie the file name to the server that it is on is a big advantage. It allows us to move users from one server to another without changing the names of their files. However, actually moving a user to a different server is difficult because the location database is embedded in the files. To move a user's subtree of files, we must save the files somewhere, delete the old subtree, create a new directory for the user on the new custodian, and then restore his files.

We back up the current system by doing tape dumps of the server file structure. Because the dump programs do not understand the invariants of the Vice data structures, special care must be taken when restoring files from the backup tapes.

5.2.2. Protection

We have found the access list mechanism to be quite useful. It is superior to Unix group protection because the user is easily able to create lists of users for protection purposes without interacting with the system administrator. While it has many advantages, it makes emulation of the Unix protection scheme difficult. We are occasionally inconvenienced by not being able to set protections separately on individual files. Also, since the *chmod* system call does not change the access lists, Unix programs cannot change the protection of files. This has been a minor inconvenience.

5.2.3. Symbolic Links

Vice does not implement symbolic links. While few programs use them explicitly, they are quite useful in administering the Unix system. They allow sharing of directories, and are frequently used to convince existing software with wired-in pathnames to look elsewhere for files. We are quite inconvenienced by their absence from Vice. For some of our problems, symbolic links on the local disk of the workstation pointing into Vice are sufficient. However, even when this scheme works, the solutions are more complicated than they would be if we could put symbolic links in Vice.

5.2.4. Rename

The rename system call is allowed in Vice only for ordinary files, not for directories. The inability to support rename on directories is a unanticipated side-effect of our decision to use full pathnames in the Vice/Venus interface. Since we have no low-level identifiers for files in Vice, the names used to refer to files in the cache change during a rename, making it impossible to reliably validate these cache entries.

As is the case with symbolic links, we have found the inability to rename directories more a user inconvenience than a source of incompatibility for application programs.

6. Vice II

In response to the problems we encountered with the prototype we are making a number of changes in the next system. The concept of file system volumes answers many of our operational problems. The use of file identifiers in the Vice/Venus interface will allow us to rename directories and implement symbolic links. Using a single-process file server and changing the server's data representation should substantially reduce the system overhead we encounter on the server.

6.1. Volumes

A *volume* is a collection of files comprising a partial subtree of the file system hierarchy. The volume is the administrative unit: the files within a volume are owned by a single user and charged to a single account. Volumes will have disk space quotas, an expiration date, access control (to be used in addition to normal access control on files), and other useful attributes. Volumes will typically be quite small (one student's files, for example) and will be easy to move between servers. Snapshots of volumes, called *read-only volumes*, will be replicable to any subset of the servers in the system; these will be used to distribute highly available but slowly changing public files.

6.2. Vice/Venus interface

The new system will use unique file identifiers (*fids*) to identify files. A fid contains a volume number, a key into the volume index, and a additional field to ensure uniqueness within the volume. The interface will remain basically the same, with fids taking the place of pathnames and some restrictions removed.

The unit of file transfer will still be the entire file. When a file is fetched, however, a *callback* is generally obtained, unless the file is being fetched from a read-only volume. When the file changes, Vice will notify all interested parties. This should greatly reduce the number of interactions between Vice and Venus as cache validation tests account for over 60% of the calls in the present system.

6.3. Vice

The changes for the next design are intended to speed up the system by reducing the overheads caused by using many processes and storing files in the Unix file system hierarchy. Our use of one process per workstation connection has frequently pushed Unix to its limits. We had to reconfigure our server kernels several times to increase various table sizes.

6.3.1. File storage

The current system uses the Unix file system for naming and storing information, and tacks on extra files for status information. The new system will keep a volume index, indexed by fid, containing status information and file addressing information. We propose to simply use a Unix inode number for the latter; in order to do this some minor kernel modifications are required. Each server will store a replicated volume location database which will be used to direct clients to the appropriate server machine.

6.3.2. Single Process Server

The current implementation uses many Unix processes per server. This results in context switching overhead and high virtual memory paging demands. It also limits our ability to explicitly cache information or to share information without introducing yet another process (the lock process, for example) or storing the shared information in the file system. To address both of these issues, we have decided to implement the main file

server functions within a single process. In order to allow this process to handle requests rapidly, it will control a small number of file transfer processes which will perform the actual bulk transfer of data to/from the workstation. The majority of requests will be handled directly by the server.

6.3.3. Venus

Venus will change to handle the new file identifier interface. This interface forces Venus to handle pathname lookups, and enables us to support symbolic links that cross server boundaries. The current implementation of Venus handles only one request at a time; the next version of Venus will overlap requests from several processes. The system call intercept code in the kernel will remain essentially the same.

6.4. RPC

The current RPC package restricts server processes to serving at most one client connection at a time. The new implementation will allow multiple clients to connect to the same server process concurrently, which will enable us to build the single process server for Vice.

7. Summary

The work described in this paper was motivated by a desire to validate the high-level decisions we had made in the design of a large-scale, location-transparent distributed file system. The most important of these decisions are the transfer of entire files to and from servers, and caching at workstations.

Our experience with the prototype has been mostly positive. A workstation using the distributed file system is quite usable despite the fact that remote file access is noticeably slower than local access. We have met the goals of location transparency and user mobility, and are able to emulate the Unix file system closely enough to be able to run application programs unchanged.

The experience of running the prototype with an actual group of users has given us confidence in our high-level design. It has also revealed that certain detailed design decisions need to be changed in order to allow the system to scale better, and to provide a more accurate emulation of Unix.

Work on a refined implementation is currently under way, and should be complete within a year.

References

[1] D.M. Ritchie and K. Thompson.
The UNIX Time-Sharing System.
Bell System Technical Journal 57(6), July-August, 1978.

[2] M. Satyanarayanan, John H. Howard, Alfred Z. Spector, Michael J. West.
The ITC Distributed File System: Principles and Design.
Companion paper, submitted to SOSP-10.