CMU-ITC-85-039

Published in:

Proceedings of the Tenth ACM Symposium on Operating Systems Principles

Orcas Island, Washington
1-4 December 1985
Volume 19, Number 5

The ITC Distributed File System: Principles and Design

M. Satyanarayanan
John H. Howard
David A. Nichols
Robert N. Sidebotham
Alfred Z. Spector
Michael J. West

INFORMATION TECHNOLOGY CENTER
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA 15213

Abstract

This paper presents the design and rationale of a distributed file system for a network of more than 5000 personal computer workstations. While scale has been the dominant design influence, careful attention has also been paid to the goals of location transparency, user mobility and compatibility with existing operating system interfaces. Security is an important design consideration, and the mechanisms for it do not assume that the workstations or the network are secure. Caching of entire files at workstations is a key element in this design. A prototype of this system has been built and is in use by a user community of about 400 individuals. A refined implementation that will scale more gracefully and provide better performance is close to completion.

1. Introduction

A campus-wide network of personal computer workstations has been proposed as an appropriate solution to the long-term computing needs of Carnegie-Mellon University (CMU) [8]. An overview paper [14] presents the rationale for this decison, along with other background information. Most pertinent to this paper is the requirement that there be a mechanism to support sharing of information between workstations. We have adopted a distributed file system with a single name space spanning all the workstations on campus as the sharing mechanism.

In this paper we present the ITC distributed file system as a solution to a system design problem. Sections 1.1 and 1.2 characterize the usage environment and discuss the considerations which led to our design. Given this context, Sections 2 and 3 describe the solution and the reasons for specific design decisions. Section 4 is retrospective in nature and discusses certain general principles that emerged during the course of the design. Section 5 describes our experience with a prototype implementation. To place our design in proper perspective, Section 6 compares it with a representative sample of other distributed file systems. Finally, Section 7 reviews the highlights of the paper.

1.1. Usage Environment

CMU is composed of approximately 6,000 individuals, each of whom could eventually own a workstation. In addition there will be some workstations at public facilities such as libraries. These observations bound the scale of the system to be between 5,000 and 10,000 network nodes.

It is usually the case that individuals confine their computational activities to a specific physical area. For example, we expect a faculty member to typically use the workstation in his office. Students, on the other hand, will tend to use workstations in their dormitories. However, it is not acceptable to *insist* that an individual restrict his activities to one workstation. The sharing mechanism we provide should not inhibit natural and spontaneous movement of users.

Most computing at CMU is related to education or research and typically involves text-processing or programming activities. The use of computers for electronic mail and bulletin boards is also common. There is some numerically-oriented computation related to simulation in departments such as physics, chemistry and electrical engineering. Finally, computers play a small but increasing role in the administration of the university.

While we expect our system usage to be initially consistent with this profile, widespread use of a campus-wide personal computer network may change established usage patterns. To meet these changes, a certain degree of system evolution will be inevitable.

1.2. Design Considerations

The most daunting aspect of this system is its scale. The projected final size of 5,000 or more nodes is at least an order of magnitude larger than any existing distributed file system. Concern for scalability has strongly motivated many key aspects of our design.

The physical compactness of the CMU campus makes it possible to use local area network (LAN) technology. A larger or more physically fragmented institution might have had to resort to lower-bandwidth networking technology.

The size of the system and its distribution across a campus make it impossible to make the entire system physically secure. It is reasonable to require that a small number of selected components of the system be located in physically secure areas. However, cross-campus LAN segments will be exposed, and user-owned workstations will be outside our administrative control. Further, individual ownership of workstations carries with it the risk that owners may choose to modify the hardware and software on their workstations in arbitrary ways. Hence workstations cannot be called upon to play any trusted role in preserving the security of the system.

We believe it important to be able to support many different kinds of workstations, and regard heterogeneity as the rule rather than the exception. Therefore, although we have focussed on a homogeneous environment for initial implementation and deployment experience, our design is extensible to an environment with diverse workstation hardware and operating systems.

2. High-level Design

2.1. The Sharing Mechanism

Why did we choose a distributed file system as our sharing mechanism?

The design alternatives for sharing in a network fall into three broad classes, ordered below according to decreasing degrees of transparency, complexity and communication requirement:

- *Distributed operating systems* such as the V kernel [4] and Accent [10] provide total network transparency for a substantial fraction of their primitives. In particular, they provide transparent access to remote files.
- *Distributed file systems* such as the Cedar file system [15] and IBIS [17] allow application programs to use remote files exactly as if they were stored locally. Such network transparency does not, however, extend to other resources such as processes and virtual memory.
- *Loosely-coupled networks*, such as the Arpanet, do not offer network transparency at all. Sharing in such a system involves explicit user actions to transfer data.

Given our desire to make sharing as effortless as possible, we rejected a loosely-coupled network approach. On the other hand, we found the constraints on our design in more serious conflict with a distributed operating system approach than a distributed file system approach. A number of considerations led us to this conclusion:

Complexity — Since a file system is only one component of an operating system, distributing it is likely to be easier than distributing the entire operating system.

Scale — A distributed operating system is likely to require more frequent interactions between its components for resource management. The anticipated scale of the system makes even the design of a distributed file system a formidable task. Its implications for a distributed operating system are more severe.

Security Workstations are not trustworthy. The building of a distributed operating system from untrustworthy components is a much harder problem.

Heterogeneity It seems a more difficult proposition to span a spectrum of workstations with a distributed operating system than with a distributed file system.

Further encouragement for adopting a distributed file system approach comes from the fact that the most common and well-understood mode of sharing between users on timesharing systems is via the file system.

2.2. File System Goals

The observations presented earlier motivate the following goals of our system:

Location Transparency There should be a single name space for all shared files in the system. Given the size of the system, we consider it unacceptable to require users to remember details such as the current location of a file or the site where it was created. Consequently, the naming scheme should not incorporate any information pertaining to the location of files. Further, the resolution of file names to network storage sites should be performed by the file system.

User Mobility Users should be able to access any file in the shared name space from any workstation. The performance characteristics of the system should not discourage users from accessing their files from workstations other than the one at which they usually work.

Security The file system cannot assume a benevolent user environment. To encourage sharing of files between users, the protection mechanism should allow a wide range of policies to be specified. Security should not be predicated on the integrity of workstations.

Performance Acceptable performance is hard to quantify, except in very specific circumstances. Our goal is to provide a level of file system performance that is at least as good as that of a lightly-loaded timesharing system at CMU. Users should never feel the need to make explicit file placement decisions to improve performance.

Scalability It is inevitable that the system will grow with time. Such growth should not cause serious disruption of service, nor significant loss of performance to users.

Availability Single point network or machine failures should not affect the entire user community. We are willing, however, to accept temporary loss of service to small groups of users.

Integrity The probability of loss of stored data should be at least as low as on the current timesharing systems at CMU. Users should not feel compelled to make backup copies of their files because of the unreliability of the system.

Heterogeneity A variety of workstations should be able to participate in the sharing of files via the distributed file system. It should be relatively simple to integrate a new type of workstation.

A system that successfully meets these goals would resemble a giant timesharing file system spanning the entire CMU campus.

Noticeably absent from the above list of goals is the ability to support large databases. As mentioned in Section 1.1, there is currently little use of such databases at CMU. The design described in this paper is suitable for files up to a few megabytes in size, given existing LAN transfer rates and workstation disk capacities. Experimental evidence indicates that over 99% of the files in use on a typical CMU timesharing system fall within this class [12]. In the future, we do expect large campus-wide databases to become increasingly important. A separate distributed database design will have to address this issue.

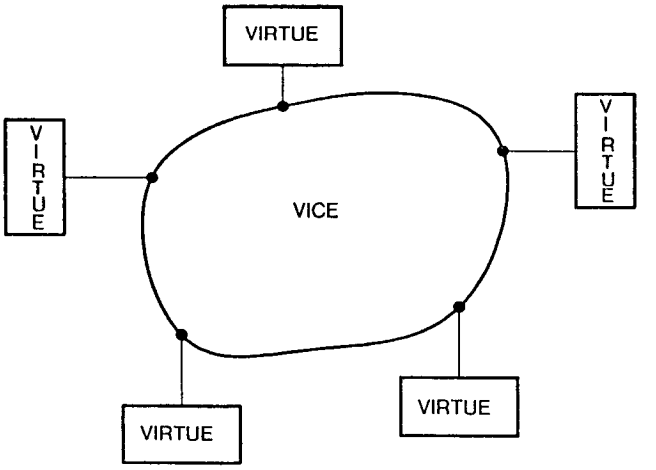
2.3. Vice and Virtue

Figure 2-1 presents a high-level view of the entire system. The large amoeba-like structure in the middle, called *Vice*[1], is a collection of communication and computational resources. A *Virtue* is an individual workstation attached to Vice. Software in Virtue makes the shared files in Vice appear as a integral part of the workstation file system.

There is a well-defined file system interface between Vice and Virtue. This interface is relatively static and enhancements to it occur in an upward-compatible manner as the system evolves. A stable interface is the key to supporting heterogeneity. To integrate a new type of workstation into the distributed file system, one need only implement software that maps the file system interface of that workstation to the Vice interface.

Vice is the boundary of trustworthiness. All computing and communication elements within Vice are assumed to be secure. This guarantee is achieved through physical and administrative control of computers and the use of encryption on the network. No user programs are executed on any Vice machine. Vice is therefore an internally secure environment unless Trojan horses are introduced by trusted system programmers.

[1] It is rumored that Vice stands for "Vast Integrated Computing Environment"

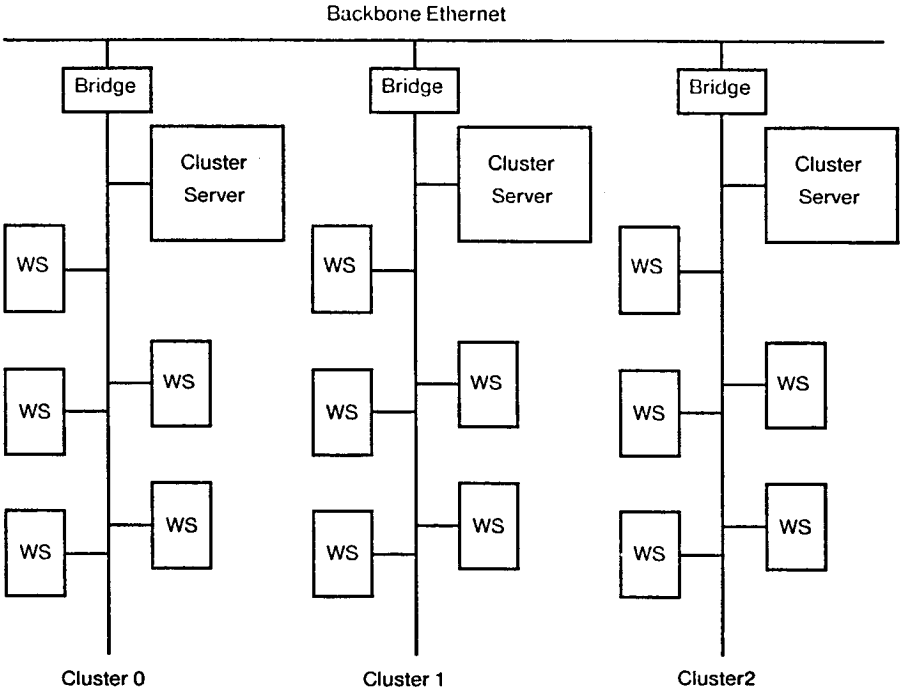


Each *Virtue* is an individual workstation. *Vice* has fine structure that is shown in detail in Figure 2-2. This diagram is certainly not to scale, since Vice will encompass an entire campus!

Figure 2-1: Vice and Virtue

Virtue, however, is under the control of individual users and is never trusted by Vice. After mutual authentication Vice and Virtue communicate only via encrypted messages. It is encryption that maintains security in spite of the fact the network is not physically secure.

Viewed at a finer granularity than Figure 2-1, Vice is composed of a collection of semi-autonomous *Clusters* connected together by a backbone LAN. Figure 2-2 illustrates such an interconnection scheme. Each cluster consists of a collection of Virtue workstations and a representative of Vice called a *Cluster Server*.



Each *WS* is a Virtue workstation. We expect a cluster to contain between 50 and 100 workstations. The final system that spans the CMU campus will have a total of about 100 clusters

Figure 2-2: Vice Topology

Each of the workstations in Figure 2-2 logically posseses a local disk. Whether this logical disk is physically associated with the workstation or is provided by a disk server is an issue that is orthogonal to the design presented here. Using a disk server may be cheaper, but will entail performance degradation. Scaling to 5000 workstations is more difficult when these workstations are paging over the network in addition to accessing files remotely. Further, security is compromised unless all traffic between the disk server and its clients is encrypted. We are not confident that paging traffic can be encrypted without excessive performance degradation. Finally, nontechnical considerations such as the need to allow students to take their workstations away from CMU during vacations and upon graduation have further motivated our requirement that workstations possess physical disks.

The *Bridges* which connect individual clusters to the backbone in Figure 2-2 serve as routers. It should be emphasised that the detailed topology of the network is invisible to workstations. All of Vice is logically one network, with the bridges providing a uniform network address space for all nodes.

Vice is decomposed into clusters primarily to addresses the problem of scale. For optimal performance, Virtue should use the server on its own cluster almost all the time, thereby making cross-cluster file references relatively infrequent. Such an access pattern balances server load and minimizes delays through the bridges. This problem of localizing file references is reminiscent of the problem of localizing virtual memory references in hierarchically structured multiprocessors such as Cm* [6].

Physical security considerations may dictate that cluster servers be co-located in small groups in machine rooms, even though each cluster server is logically associated with the workstations in its cluster.

3. Detailed Design

In designing a distributed file system one has to answer a number of fundamental questions. Chief among these are:

- How are files named? Is the location of a file in the network apparent from its name? If not, how are files located in the system?
- Can multiple copies of a file be stored at different sites? How are these copies updated and kept consistent?
- What are the primitives available to application programs to operate on remote files? Are they identical to the primitives provided for local files?
- How do network nodes access files? What are the inter-machine primitives available?
- How is security enforced in the system? Can the nodes on the network trust each other? Can the network be assumed immune from eavesdropping?
- What is a feasible implementation strategy for the design?

This list is by no means exhaustive, but it does characterize a core of design issues that any distributed file system design must address either explicitly or by default. No single set of answers to these questions can be considered optimal for all situations. The choice depends on the goals of the design and the external constraints placed upon it.

In Sections 3.1 to 3.5 we describe our design by examing the choices that we have made in answering the questions listed above. This organization is exactly mirrored in Section 6, where we compare our system to other distributed file systems.

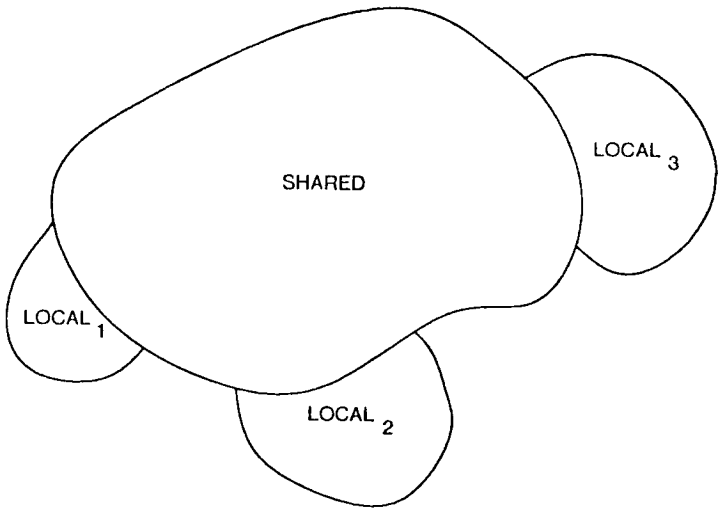
3.1. Naming and Location

From the point of view of each workstation, the space of file names is partitioned into a *Local* name space and a *Shared* name space. Figure 3-1 illustrates this partitioning. The shared name space is the same for all workstations, and contains the majority of files accessed by users. The local name space is small, distinct for each workstation, and contains files which typically belong to one of the following classes:

1. System files essential for initializing the workstation and for its functioning prior to connection to Vice.
2. Temporary files, such as those containing intermediate output from compiler phases. Placing such files in the shared name space serves no useful purpose.
3. Data files that the owner of the workstation considers so sensitive that he is unwilling to entrust them to the security mechanisms of Vice. In practice we expect few such files. Since these files cannot be accessed from any other workstation they hinder user mobility.
4. A small number of frequently used, but rarely updated, system programs. Such programs may be stored locally to improve performance and to allow at least a modicum of usability when Vice is unavailable.

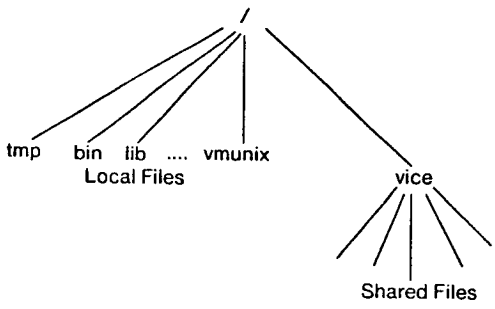
Shared files are stored in Vice in a hierarchically structured name space, similar to Unix [11]. It is the responsibility of Virtue to map this hierachical structure to a format consistent with the local name space. To simplify exposition, we assume throughout this section that Virtue is a Unix workstation.

In Unix terminology, the local name space is the *Root File System* of a workstation and the shared name space is *mounted* on a



This is a Venn diagram depicting local and shared name spaces. $Local_1$, $Local_2$, and $Local_3$ correspond to files in the local file systems of three different workstations. *Shared* corresponds to the shared set of files in the system, and is identical for all the workstations.

Figure 3-1: Shared and Local Name Spaces



The files in the "/vice" subtree are common to all the workstations in the system. Certain directories and files in the local name space, such as "/bin" and "/lib", are symbolic links into "/vice". All other local files are inaccessible to other workstations.

Figure 3-2: A Unix Workstation's View of the File System

known leaf directory during workstation initialization. Figure 3-2 depicts this situation, with "/vice" being the directory on which Vice is mounted. File names generated on the workstation with "/vice" as the leading prefix correspond to files in the shared space. All other names refer to files in the local space.

The presence of different types of workstations introduces an additional complication in the naming scheme. Consider, for example, an environment consisting of Sun and Vax workstations, both running Unix. Regardless of the hardware it is run on, Unix expects to find system binaries in a directory whose pathname is "/bin". If these system binaries are stored in the shared name space, there has to be a mechanism to differentially redirect file accesses from different types of workstations. We use *Symbolic Links* in Virtue for this purpose. On a Sun workstation, the local directory "/bin" is a symbolic link to the remote directory "/vice/unix/sun/bin"; on a Vax, "/bin" is a symbolic link to "/vice/unix/vax/bin". The extra level of indirection provided by symbolic links is thus of great value in supporting a heterogeneous environment.

Since location transparency is one of our goals, Virtue cannot deduce the server on which a shared file is stored by merely examining the name of the file. It is Vice that provides the necessary file location mechanism. Each cluster server contains a complete copy of a location database that maps files to *Custodians*. The custodian of a file is the cluster server that has complete responsibility for storage of the file and the servicing of requests for it. A workstation has to find the custodian for a file

before operating on it. If a server receives a request for a file for which it is not the custodian, it will respond with the identity of the appropriate custodian. The size of the replicated location database is relatively small because custodianship is on a subtree basis. If all files in a subtree have the same custodian, the location database has only an entry for the root of the subtree.

File subtrees of individual users are assigned to custodians in a manner that balances server load and minimizes cross-cluster references. A faculty member's files, for instance, would be assigned to the custodian which is in the same cluster as the workstation in his office. This assignment does not affect the mobility of that individual, because he can still access his files from any other part of the campus, albeit with some performance penalty.

An important pro erty of the location database is that it changes relatively slowly. There are two reasons for this. First, most file creations and deletions occur at depths of the naming tree far below that at which the assignment of custodians is done. Hence normal user activity does not alter the location database. Second, the reassignment of subtrees to custodians is infrequent and typically involves human interaction. For example, if a student moves from one dormitory to another he may request that his files in Vice be moved to the cluster server at his new location. Alternatively, we may install mechanisms in Vice to monitor long-term access file patterns and recommend changes to improve performance. Even then, a human operator will initiate the actual reassignment of custodians.

Changing the location database is relatively expensive because it involves updating all the cluster servers in the system. The files whose custodians are being modified are unavailable during the change. As explained in the previous paragraph, our design is predicated on the assumption that such changes do not occur frequently. This assumption does not compromise our goal of allowing user mobility with reasonable performance because a different mechanism, described in the next section, addresses this issue.

3.2. Replication

Caching is the main form of replication in our design. Virtue caches entire files along with their status and custodianship information. Caching and whole-file transfer are key mechanisms in meeting the design objectives of performance, mobility and scalability.

Part of the disk on each workstation is used to store local files, while the rest is used as a cache of files in Vice. When an application program on a workstation opens a file in the shared name space, Virtue locates the appropriate custodian, fetches the file, and stores it in the cache. This fetch is avoided if the file is already present in the cache. After the file is opened, individual read and write operations are directed to the cached copy. Virtue does not communicate with Vice in performing these operations. When the file is closed, the cache copy is transmitted to the appropriate custodian. Note that all interactions with Vice are transparent to application programs. Other than performance, there is no difference between accessing a local file and a file in the shared name space.

Cache validation involves communication between the custodian of a file and the workstations which have that file cached. This may either be initiated by Virtue before each use of the cached copy, or by Vice whenever the file is modified. The choice trades longer file open latencies and increased server loads in the former case, for larger server state and slower updates to files in the latter case. Since files tend to be read much more frequently than written, better performance is likely with the latter. Our current design uses check-on-open to simplify implementation and reduce server state. However, experience with a prototype has convinced us that the cost of frequent cache validation is high enough to warrant the additional complexity of an invalidate-on-modification approach in our next implementation.

Changes to a cached file may be transmitted on close to the corresponding custodian or deferred until a later time. In our design, Virtue stores a file back when it is closed. We have adopted this approach in order to simplify recovery from workstation crashes. It also results in a better approximation to a timesharing file system, where changes by one user are immediately visible to all other users.

The caching mechanism allows complete mobility of users. If a user places all his files in the shared name space, he can move to any other workstation attached to Vice and use it exactly as he would use his own workstation. The only observable differences are an initial performance penalty as the cache on the new workstation is filled with the user's working set of files and a smaller performance penalty as inter-cluster cache validity checks and cache write-throughs are made.

The caching of entire files, rather than individual pages, is fundamental to our design. It has a strong positive influence on performance for a number of reasons. First, custodians are contacted only on file opens and closes, and not on individual reads and writes. Second, the total network protocol overhead in transmitting a file is lower when it is sent *en masse* rather than in

a series of responses to requests for individual pages. Finally, disk access routines on the servers may be better optimized if it is known that requests are always for entire files rather than for random disk blocks.

The use of whole-file transfer may also simplify the support of heterogeneous workstations. It is likely to be easier for Virtue to transform a file from the form in which it is stored in Vice to a form compatible with the native file system of the workstation when the entire file is available in the cache. For instance, a directory stored as a Vice file is easier to interpret when the whole file is available.

In addition to caching, Vice also supports read-only replication of subtrees at different cluster servers. Files which are frequently read, but rarely modified, may be replicated in this way to enhance availability and to improve performance by balancing server loads. The binaries of system programs are a typical example of this class of files.

In our prototype, described in Section 5, the updating of a read-only subtree is performed asynchronously by its custodian. Our revised implementation will make read-only subtrees truly immutable. The creation of a read-only subtree is an atomic operation, thus providing a convenient mechanism to support the orderly release of new system software. Multiple coexisting versions of a subsystem are represented by their respective read-only subtrees. Caching of files from read-only subtrees is simplified since the cached copies can never be invalid.

3.3. Functionality of Interfaces

There are two distinct programming interfaces in this design: the Vice-Virtue interface, which is primarily of concern to implementors wishing to attach new types of workstations to Vice, and the Virtue file system interface, which is visible to application programs.

Vice provides primitives for locating the custodians of files, and for fetching, storing, and deleting entire files. It also has primitives for manipulating directories, examining and setting file and directory attributes, and validating cached copies of files.

The interface provided by Virtue is workstation-specific. In the prototype discussed in Section 5, the primitives supported are the standard Unix file system primitives, supporting directory manipulation and byte-at-a-time access to files.

In an ideal implementation, Virtue will provide identical interfaces for shared and local files. The degree to which this ideal is met is one measure of quality of workstation attachment software. We are highly encouraged by our experience in attaching Unix workstations to Vice. Though we have no experience as yet in attaching other kinds of workstations, we do not forsee any fundamental problems on account of our design.

Besides the need to bridge the semantic gap between the file system interfaces of Vice and Virtue, there is also an assumption in our design that workstations possess adequate resources to effectively use Vice. For example, workstations need to have disks (real or virtual) large enough to cache a typical working set of files. They also need a high-performance hardware interface to the campus-wide LAN. It would be desirable to allow workstations that fail to meet these minimal resource requirements to access Vice, perhaps at lower performance or convenience.

An approach we are exploring is to provide a *Surrogate Server* running on a Virtue workstation. This surrogate would behave as a single-site network file server for the Virtue file system. Clients of this server would then be transparently accessing Vice files on account of a Virtue workstation's transparent Vice attachment. The software interface to this server would be tailored to meet the specific needs of the low-function workstations in question and it could run on a machine with hardware interfaces to both the campus-wide LAN and a network to which the low-function workstations could be cheaply attached. Work is currently in progress to build such a surrogate server for IBM PCs. We believe that this approach is also applicable to machines such as the Apple Macintosh.

3.4. Security

Voydock and Kent [18] classify breaches of security in a network as the unauthorized *release of information*, *modification of information*, or *denial of resource usage*. In this design we only address release and modification of information. Resource denial is trivial when a user can modify the hardware and software of a workstation. For example, a workstation on an Ethernet can be made to generate collisions whenever a packet is transmitted by any other workstation. This would effectively deny network services to all other workstations. We believe that peer pressure and social mores are the only effective practical weapons to deal with such situations in our environment. Fortunately, most cases of resource denial are relatively easy to detect.

In this section we describe how our design provides authentication, access control and secure network transmission. These components jointly provide the mechanism needed to

prevent the unauthorized release or modification of files stored in Vice.

Vice uses encryption extensively as a fundamental building block in its higher level network security mechanisms. To build a truly secure distributed environment, we are convinced that encryption should be available as a cheap primitive at every network site. Fortunately, VLSI technology has made encryption chips available at relatively low cost.

The authentication and secure transmission functions are provided as part of a connection-based communication package, based on the remote procedure call paradigm. At connection establishment time, Vice and Virtue are viewed as mutually suspicious parties sharing a common encryption key. This key is used in an authentication handshake, at the end of which each party is assured of the identity of the other. The final phase of the handshake generates a session key which is used for encrypting all further communication on the connection. The use of per-session encryption keys reduces the risk of exposure of authentication keys.

When a user initiates activity at a workstation, Virtue authenticates itself to Vice on behalf of that user. Since the key used for this is user-specific it has to be obtained from the user. One way to do this is by transformation of a password. Note that the password itself is not transmitted, but is only used to derive the encryption key. Alternative approaches, such as equipping each workstation with a peripheral to read encryption keys from magnetically encoded cards carried by users, are also possible.

In addition to authentication, a mechanism to control access is needed within Vice. Sharing in a large user community implies that such a mechanism must allow the specification of a wide range of protection policies and must provide for easy revocation. Our design uses access lists for this purpose.

Entries on an access list are from a protection domain consisting of *Users*, who are typically human beings, and *Groups*, which are collections of users and other groups. The recursive membership of groups is similar to that of the registration database in Grapevine [1]. It simplifies administration and leads to shorter access lists at the cost of complicating the implementation of group manipulation primitives.

Information about users and groups is stored in a protection database which is replicated at each cluster server. Manipulation of this database is via a protection server, which coordinates the updating of the database at all sites.

The rights possesed by a user on a protected object are the union of the rights specified for all the groups that he belongs to, either directly or indirectly. This subset of groups is referred to as the *Current Protection Subdomain* (CPS) of the user. A user may be given access to an object either by making him a member of a group that already has appropriate access rights on that object, or by explicitly adding that user to the access list.

Access is revoked by removing a user from all groups which have access to the object in question. Because of the distributed nature of the system and the recursive membership of groups, this operation may be unacceptably slow in emergencies. We therefore support the concept of *Negative Rights* in access lists. The union of all the negative rights specified for a user's CPS is *subtracted* from his positive rights. To revoke a user's access to an object, he can be given negative rights on that object. Negative rights are intended as a rapid revocation mechanism for limiting the damage caused by a user who has been discovered to be untrustworthy.

In our prototype the protected entities are directories, and all files within a directory have the same protection status. Per-directory protection reduces the storage overheads of access lists and also reduces the amount of protection state that users have to keep track of mentally. The rights associated with a directory control the fetching and storing of files, the creation and deletion of new directory entries, and modifications to the access list. For reasons discussed in Section 5, we will incorporate a hybrid scheme with access lists on directories and additional per-file protection bits in our reimplementation of the file system.

3.5. Implementation Strategy

Since this paper focuses on high-level issues, we only briefly touch upon how this design is implemented. The description in this section is organized around three basic questions pertaining to implementation:

1. How does Virtue transparently interpose cached copies of files to application programs?
2. What is the structure of a server?
3. How do servers and clients communicate?

As we will be making some changes on the basis of experience with a prototype, we indicate both our original approach and the modifications.

3.5.1. File Intercept and Cache Management

Virtue is implemented in two parts: a set of modifications to the workstation operating system to intercept file requests, and a user-level process, called *Venus*. Venus handles management of the cache, communication with Vice and the emulation of native

file system primitives for Vice files. The modifications to the operating system are minimal since Venus provides much of the needed functionality.

It is possible to implement the interception of file system calls by recompiling or relinking application programs with a special library of input-output subroutines. Such a mechanism avoids modifications to the workstation operating system. We have not adopted this approach because of our desire to support proprietary software for which only the executable binaries may be available. Further, new releases of the file system software do not require us to relink any user or system software. This saves us from a potential administrative nightmare in a 5000 node network.

In our prototype, Venus uses a simple LRU cache management algorithm with a directory in a workstation's local Unix file system as cache storage. Since files are cached in their entirety, the amount of state needed to represent the cache contents is significantly smaller than in a typical virtual memory cache or in a file cache where pages of files are individually cached. Venus limits the total number of files in the cache rather than the total size of the cache, because the latter information is difficult to obtain from Unix. In view of our negative experience with this approach, we will incorporate a space-limited cache management algorithm in our reimplementation.

3.5.2. Server Structure

Our prototype implements a cluster server with a collection of Unix processes. On each server there is one Unix process to deal with each user on each client workstation communicating with that server. Due to the limitations imposed by Unix, these per-client processes cannot share data structures in virtual memory. File server functions which require such sharing are implemented using a single dedicated Unix process for each such function. For example, there is a single lock server process which serializes requests and maintains lock tables in its virtual memory.

Experience with the prototype indicates that significant performance degradation is caused by context switching between the per-client Unix processes. In addition, the inability to share data structures between these processes precludes many strategies to improve performance. Our reimplementation will represent a server as a single Unix process incorporating a lightweight process mechanism to provide independent per-client threads of control. Global data in that Unix process will be used to represent data structures shared by the lightweight processes.

The prototype file server uses the underlying Unix file system for storage of Vice files. Each Vice file is physically represented as two Unix files: one containing uninterpreted data and the other, the *.admin* file, containing Vice status information. The location database in our prototype is not explicit but is represented by stub directories in the Vice file storage structure.

The reimplementation will use a separate data structure for the location database. We will still use the Unix file system to store Vice files, but will modify Unix on the servers to allow us to access files via their low-level identifiers rather than their full Unix pathnames. Our observations of the prototype indicate that this modification is likely to yield significant performance improvement.

The prototype does not have a protection server, but relies on manual updates to the protection database by the operations staff. The reimplementation will incorporate a protection server.

3.5.3. Client-Server Communication

Virtue and Vice communicate by a remote procedure call mechanism (RPC) [2]. The prototype RPC implementation uses a reliable byte-stream protocol supported by Unix. Whole-file transfer is implemented as a side effect of a remote procedure call.

To overcome Unix resource limitations and thus allow large client/server ratios, the revised RPC implementation uses an an unreliable datagram protocol supported by Unix. This implementation closely integrates RPC with the lightweight process mechanism mentioned in Section 3.5.2. This allows a Unix processs to concurrently perform and service multiple remote procedure calls, while still maintaining the synchronous semantics of RPC with respect to individual lightweight threads of control within that Unix process. Generalized side-effects are supported, whole-file transfer being a particular kind of side-effect.

Mutual client/server authentication and end-to-end encryption facilities are integrated into the RPC package. These functions are an integral part of the overall security of Vice and Virtue.

3.6. Other Design Issues

Vice provides primitives for single-writer/multi-reader locking. Such locking is advisory in nature, and it is the responsibility of each application program to ensure that all competing accessors for a file will also perform locking. This decision is motivated by our positive experience with Unix, which does not require files to be locked before use. Action consistency for fetch and store operations on a file is guaranteed by Vice even in the absence of

locks. A workstation which fetches a file at the same time that another workstation is storing it, will either receive the old version or the new one, but never a partially modified version.

An unfortunate side-effect of trying to emulate the timesharing paradigm is the need to provide mechanisms to restrict and account for the usage of shared resources. The resource we are most concerned with is disk storage on the cluster servers. We intend to provide both a quota enforcement mechanism and a file migration facility in our reimplementation; these facilities are not available in our prototype. As use of this system matures, it may become necessary to account for other resources, such as server CPU cycles or network bandwidth. Until the need for such accounting is convincingly demonstrated, however, we intend to treat these as free resouces.

Another area, whose importance we recognize, but which we have not had the opportunity to examine in detail yet is the development of monitoring tools. These tools will be required to ease day-to-day operations of the system and also to recognize long-term changes in user access patterns and help reassign users to cluster servers so as to balance server loads and reduce cross-cluster traffic.

4. Design Principles

A few simple principles underlie the design presented in this paper. It should be emphasised these are being presented *a posteriori*, and that the design did not proceed by stepwise refinement of these principles. Rather, the principles evolved during the course of the design. In the rest of this section we discuss each of these principles and point out instances of their application in our design.

- *Workstations have the cycles to burn.*

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

  Vice requires that each workstation contact the appropriate custodian for a file before operating on it. There is no forwarding of client requests from one cluster server to another. This design decision is motivated by the observation that it is preferable to place the burden of locating and communicating with custodians on workstations rather than servers.

  We will further exploit this principle in the second implementation of the system. Currently, workstations present servers with entire pathnames of files and the servers do the traversing of pathnames prior to retrieving the files. Our revised implementation will require workstations to do the pathname traversal themselves.

- *Localize if possible*

  If feasible, use a nearby resource rather than a distant one. This has the obvious advantage of improved performance and the additional benefit that each part of the distributed system is less susceptible to events such as overloading in other parts. Potentially in conflict with this principle is the goal of user mobility, which requires data to be easily locatable. A successful design has to balance these two considerations.

  The decomposition of Vice into clusters is an instance where we have tried to localize resource usage. Another example is the replication of read-only subtrees, thereby enabling system programs to be fetched from the nearest cluster server rather than its custodian. Caching obviously exploits locality, but we discuss it separately because it is so fundamental to our design.

  One may view the decision to transfer entire files rather than individual pages as a further application of this principle. Read and write operations are much more frequent than opens and closes. Contacting Vice only on opens and closes reduces our usage of remote resources.

- *Exploit class-specific file properties.*

  It has been shown [13] that files in a typical file system can be grouped into a small number of easily-identifiable classes, based on their access and modification patterns. For example, files containing the binaries of system programs are frequently read but rarely written. On the other hand temporary files containing intermediate output of compiler phases are typically read at most once after they are written. These class-specific properties provide an opportunity for independent optimization, and hence improved performance, in a distributed file system design.

  The fact that system binaries are treated as replicatable, read-only files is a case where this priniciple is being used. We may further exploit this principle by allowing a subset of the system binaries to be placed in the local file systems of individual workstations. Since such files change infrequently, explicit installation of new versions of these files by users is acceptable. The storage of temporary files in the local, rather than shared, name space of a workstation is another instance of a file-specific design decision.

- *Cache whenever possible.*

  Both the scale of the system and the need for user mobility motivate this principle. Caching reduces contention on centralized resources. In addition, it transparently makes data available wherever it is being currently used.

Virtue caches files and status information about them. It also caches information about the custodianship of files. Though not discussed in this paper, our reimplementation will use caching extensively in the servers.

- *Avoid frequent, system-wide rapid change.*

The more distributed a system is, the more difficult it is to update distributed or replicated data structures in a consistent manner. Both performance and availability are compromised if such changes are frequent. Conversely, the scalability of a design is enhanced if it rarely requires global data to be consistently updated.

As discussed earlier, the replicated custodian database in Vice changes slowly. Caching by Virtue, rather than custodianship changes in Vice, is used to deal with rapid movement of users.

Another instance of the application of this principle is the use of negative rights. Vice provides rapid revocation by modifications to an access list at a single site rather than by changes to a replicated protection database.

5. The Prototype

Our intent in implementing a prototype was to validate the design presented in this paper. The implementation was done by 4 individuals over a period of about one year. In this section we describe the current status of the system, its performance, and the changes we are making in the light of our experience.

5.1. Status

The prototype has been in use for about a year, and has grown to a size of about 120 workstations and 6 servers. More than 400 individuals have access to this system at the present time. The prototype meets the goals of location transparency and user mobility unequivocally. Our initial apprehensions about relying solely on caching and whole-file transfer have proved baseless. Application code compatibility has been met to a very high degree, and almost every Unix application program is able to use files in Vice. None of these programs has to be recompiled or relinked to work in our system.

The mechanisms for authentication and secure transmission are in place, but await full integration. We are awaiting the incorporation of the necessary encryption hardware in our workstations and servers, since software encryption is too slow to be viable.

The access list mechanism has proved to be a flexible and convenient way to specify protection policies. Users seem quite comfortable with per-directory access list protection. However, we have encountered certain difficulties in mapping the per-file protection supported by Unix to the per-directory protection semantics of Vice. A few programs use the per-file Unix protection bits to encode application-specific information and are hence unable to function correctly with files in Vice. The reimplementation will have per-file protection bits in addition to access lists on directories.

The prototype fails to emulate Unix precisely in a few other areas too. Two shortcomings that users find particularly irksome are the inability to rename directories in Vice, and the fact that Vice does not support symbolic links[2]. These limitations are subtle consequences of the implementation strategy we chose in the prototype, and will be rectified in our revised implementation.

5.2. Performance

For a rapid prototyping effort, performance has been surprisingly good. The prototype is usable enough to be *the* system on which all further development work is being done within our user community.

Measurements indicate an average cache hit ratio of over 80% during actual use. Server CPU utilization tends to be quite high: nearly 40% on the most heavily loaded servers in our environment. Disk utilization is lower, averaging about 14% on the most heavily loaded servers. These figures are averages over an 8-hour period in the middle of a weekday. The short-term resource utilizations are much higher, sometimes peaking at 98% server CPU utilization! It is quite clear from our measurements that the server CPU is the performance bottleneck in our prototype.

A histogram of calls received by servers in actual use shows that cache validity checking calls are preponderant, accounting for 65% of the total. Calls to obtain file status contribute about 27%, while calls to fetch and store files account for 4% and 2% respectively. These four calls thus encompass more than 98% of the calls handled by servers. Based on these observations we have concluded that major performance improvement is possible if cache validity checks are minimized. This has led to the alternate cache invalidation scheme mentioned in Section 3.2.

To assess the performance penalty caused by remote access, we ran a series of controlled experiments with a benchmark. This benchmark operates on about 70 files corresponding to the source code of an actual Unix application. There are five distinct

[2] Note that symbolic links from the local name space *into* Vice are supported.

phases in the benchmark: making a target subtree that is identical in structure to the source subtree, copying the files from the source to the target, examining the status of every file in the target, scanning every byte of every file in the target, and finally compiling and linking the files in the target. On a Sun workstation with a local disk, the benchmark takes about 1000 seconds to complete when all files are obtained locally. Our experiments show that the same benchmark take about 80% longer when the workstation is obtaining all its files from an unloaded Vice server.

In actual use, we operate our system with about 20 workstations per server. At this client/server ratio, our users perceive the overall performance of the workstations to be equal to or better than that of the large timesharing systems on campus. However, there have been a few occasions when intense file system activity by a few users has drastically lowered performance for all other active users.

5.3. Changes

Based on our experience, a redesign and reimplementation effort is currently under way. While retaining the design at the level of abstraction presented in this paper, we will introduce many lower-level changes to enhance performance and scalability, and to allow a more accurate mapping of Unix file system semantics on Vice.

Some of these changes have been mentioned in Sections 3.5 and 3.2. These include:

- a modified cache validation scheme, in which servers notify workstations when their caches become invalid.
- a single-process server structure, with a low-level interface to Unix files.
- a revised RPC implementation, integrated with a lightweight process mechanism.
- a space-limited cache management algorithm in Venus.

Another noteworthy change is the use of fixed-length unique *file identifiers* for Vice files. In the prototype, Venus presents entire pathnames to Vice. In our revised implementation, Venus will translate a Vice pathname into a file identifier by caching the intermediate directories from Vice and traversing them. The offloading of pathname traversal from servers to clients will reduce the utilization of the server CPU and hence improve the scalability of our design. In addition, file identifiers will remain invariant across renames, thereby allowing us to support renaming of arbitrary subtrees in Vice.

In order to simplify day-to-day operation of the system, we will introduce the concept of a *Volume* in Vice. A volume is a complete subtree of files whose root may be arbitrarily relocated in the Vice name space. It is thus similar to a mountable disk pack in a conventional file system. Each volume may be turned offline or online, moved between servers and salvaged after a system crash. A volume may also be *Cloned*, thereby creating a frozen, read-only replica of that volume. We will use copy-on-write semantics to make cloning a relatively inexpensive operation. Note that volumes will not be visible to Virtue application programs; they will only be visible at the Vice-Virtue interface.

Finally, the revised implementation will allow closer emulation of Unix by providing features such as symbolic links, directory rename and per-file protection.

6. Relationship to Other Systems

A number of different network file system designs have been proposed and implemented over the last few years. We consider a representative sample of such systems here and contrast their design with ours. Due to constraints of space we provide only enough detail to make the differences and similarities apparent. The survey by Svobodova [16] provides a more comprehensive and detailed comparative discussion of network file systems.

The systems we compare are:

- *Locus* [9, 19], designed and implemented at the University of California at Los Angeles.
- The *Newcastle Connection* [3], from the University of Newcastle-upon-Tyne.
- The *ROE* file system [5], currently being implemented at the University of Rochester.
- *IBIS* [17], which has been partially implemented at Purdue University.
- The *Apollo* system [7], which is a commercial system marketed by Apollo Computers, Inc.
- The *Cedar File System* [15], implemented at the Xerox Palo Alto Reseach Center.

We compare Vice-Virtue to these systems by presenting their approach to each of the fundamental design issues mentioned in Section 3. Such a comparison brings into focus the position that Vice-Virtue occupies in the distributed file system design space. We do realize, however, that a comparison along specific attributes may omit other interesting features of the systems being compared.

6.1. Naming and Location

All the systems in question support a hierarchical name space, both for local and remote files. In many cases the naming structure is identical to Unix. Roe and the Cedar File System provide, in addition, a version number component to names. Vice-Virtue and Roe provide a Unix-like name structure at the client-server interface and leave open the naming structure on the workstations.

Location transparency is a key issue in this context. In Locus, Vice-Virtue, Apollo and Roe it is not possible to deduce the location of a file by examing its name. In contrast, the Cedar File System and the Newcastle Connection embed storage site information in pathnames. IBIS intends to eventually provide location transparency, though it currently does not do so.

Location transparent systems require a mechanism to map names to storage sites. In Vice-Virtue, there is clear distinction between servers and clients. Every server maintains a copy of a location database which is used to answer queries regarding file location. Clients use cached location information as hints. Roe logically provides a single server which maps names to storage sites, but this server may be implemented as a collection of processes at different nodes. The Apollo system uses a collection of heuristics to locate objects. Looking up a pathname in a directory yields a low-level identifier which contains a hint regarding the location of the object. Locus does not distinguish between servers and clients, and uses a location database that is replicated at all sites.

6.2. Replication

The replication of data at different sites in a distributed system offers two potential benefits. First, it offers increased availability, by allowing alternate copies to be used when the primary copy is unavailable. Second, it may yield better performance by enabling data to be accessed from a site to which access time is lower. The access time differential may arise either because of network topology or because of uneven loading of sites.

The Cedar File System and Vice-Virtue use transparent caching of files at usage sites to improve performance. In Vice-Virtue caching is also important in meeting the goal of user mobility. ROE and a proposed extension of IBIS support both caching and migration of files. Migration differs from caching in that it is explicitly initiated by users and involves only data movement, not replication. IBIS views cachability as a file property, thereby providing the opportunity for users to mark frequently updated shared files as being not cachable. Apollo integrates the file system with the virtual memory system on workstations, and hence caches individual pages of files, rather than entire files.

Systems which cache data need to ensure the validity of their cache entries. In the Cedar File System cached data is always valid, because files are immutable. Higher-level actions by a workstation user, such as an explicit decision to use a new version of a subsystem, are the only way in which a set of cached files is rendered obsolete. In the Vice-Virtue prototype, a cache entry is validated when a file is opened, by comparing its timestamp with that of the copy at the custodian. Apollo uses a similar approach, comparing timestamps when a file is first mapped into the address space of a process. No validation is done on further accesses to pages within the file, even though these may involve movement of data from the site where the file is stored. For reasons mentioned earlier, Vice-Virtue intends to reverse the order of cache validation, requiring servers to invalidate caches on updates.

Replication can take forms other than caching. In Locus, for instance, entire subtrees can be replicated at different sites. Updates are coordinated by only one of these sites. In case of network partition, updates are allowed within each of the partitioned subnets. A conflict resolution algorithm is used to merge updates after the partition is ended. Vice-Virtue also provides read-only replication of subtrees, but does not allow replicated copies to be updated during partition.

ROE uses weighted voting to verify the currency of replicated data and to determine whether a copy of a file can be updated in the presence of network or server failure. IBIS supports replication, but the published literature does not provide details of the mechanism.

6.3. Functionality of Interfaces

All the systems being compared provide application programs with the same interface to local and remote files. One may, in fact, view this as the defining property of a distributed file system. There is considerable latitude, however, in the manner in which this interface is mapped into the inter-machine interface.

In systems such as Locus and the Newcastle Connection, the inter-machine interface is very similar to the application program interface. Operations on remote files are forwarded to the appropriate storage site, where state information on these files is maintained. The current implementation of IBIS is similar.

The Apollo system maps files into virtual memory. Its remote interface is essentially a page fault/replace interface, with additional primitives for cache validation and concurrency control. ROE's intermachine interface support caching and

migration, but it is also possible to have a file opened at a remote site and have individual bytes from it shipped to the local site.

Cedar and Vice-Virtue are similar in that their inter-machine interfaces are very different from their application program interface. Cedar uses a predefined file transfer protocol to fetch and store files on network servers. This has the advantage of portability, and allows existing file servers to be used as remote sites. Vice-Virtue has a customized interface at this level.

6.4. Security

With the exception of Vice-Virtue, all the systems discussed here trust the hardware and system software on the machines they run on. User processes authenticate themselves at remote sites using a password. The acquisition and forwarding of the password is done by trusted software on the client sites. The remote site is trusted without question by the client.

The IBIS description mentions a connection setup procedure that prevents stealing of connections by malicious processes. However, the procedure assumes the presence of a trusted process at each end, with an existing secure channel of communication between them.

Since workstations are not trusted in Vice-Virtue, mutual authenticity is established by an encryption-based handshake with a key derived from user-supplied information. Once a connection is established, all further communications on it is encrypted.

For access control, Locus, the Newcastle Connection and IBIS use the standard Unix protection mechanism. Apollo, Vice-Virtue, Cedar and ROE use more general access lists for specifying protection policies.

6.5. Implementation Strategy

In IBIS and the Newcastle Connection the interception of file system calls is done by linking application programs with a special library of routines. The intercepted calls are forwarded to user-level server processes at remote sites.

In contrast, Locus is implemented as an extensive modification of a standard Unix system. The operating system itself does the interception of remote file system calls and handles file requests from remote sites. Apollo uses a customized operating system, with builtin remote access capability. The available literature on ROE does not provide implementation details.

File system interception in Virtue is done by the kernel, but most of the functionality needed to support transparent remote access is provided by a user-level cache manager process. Vice is implemented with user-level server processes. As mentioned earlier, the reimplementation will have a small number of kernel modifications, solely for performance reasons.

7. Conclusion

The highlights of this paper are as follows:

- Our primary concern is the design of a sharing mechanism for a computing environment that is a synthesis of the personal computer and timesharing paradigms.
- We support sharing via a campus-wide location transparent distributed file system which allows users to move freely between all the workstations in the system.
- Scale, security and performance are the hardest problems in this system. The need to retrofit our mechanisms into existing operating system interfaces and the need to support a heterogeneous environment are additional constraints on our design.
- Whole-file transfer and caching are important design features that jointly address the issues of performance and scale. Clustering to exploit locality of usage and the replication of read-only system files are two other design features motivated by the same issues.
- The design incorporates mechanisms for authentication and secure transmission that do not depend on trusted workstations or a secure network. A flexible access control mechanism is also provided.
- We have implemented a prototype of this design and it is in day-to-day use by a small user community. Experience with the prototype has been positive, but has also revealed certain inadequacies. These shortcomings arise on account of certain detailed implementation decisions in our prototype rather than fundamental design deficiencies.
- A comparison with other distributed file systems reveals that although this design has individual features in common with some of the other systems, it is unique in the way it combines these features to produce a total design. It is further distinguished from all the other systems in that it does not rely on the trustworthiness of all network nodes.

The success of our prototype has given us confidence in the viability of the design presented in this paper. Our current reimplementation effort is essentially a refinement of this design. We anticipate our user population to grow by an order of magnitude and span the entire CMU campus in the next two years.

Acknowledgements

Dave Gifford and Rick Rashid played an important role in the early discussions that led to the design described here. Dave King was part of the team that implemented the prototype. Richard Snodgrass provided valuable comments on an initial draft of this paper.

This work was funded by the IBM Corporation.

References

[1] Birrell, A., Levin, R., Needham, R. and Schroeder, M.
Grapevine: An Exercise in Distributed Computing.
In *Proceedings of the Eighth Symposium on Operating System Principles*. December, 1981.

[2] Birrell, A.D. and Nelson, B.J.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[3] Brownbridge, D.R., Marshall, L.F. and Randell, B.
The Newcastle Connection.
Software Practice and Experience 12:1147-1162, 1982.

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

[5] Ellis, C.A. and Floyd, R.A.
The ROE File System.
In *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*. October, 1983.

[6] Jones, A.K. and Gehringer, E.F. (Editors).
The Cm Multiprocessor Project: A Research Review*.
Technical Report CMU-CS-80-131, Department of Computer Science, Carnegie-Mellon University, July, 1980.

[7] Nelson, D.L. and Leach, P.J.
The Architecture and Applications of the Apollo Domain.
IEEE Computer Graphics and Applications , April, 1984.

[8] The Task Force for the Future of Computing, Alan Newell (Chairman).
The Future of Computing at Carnegie-Mellon University.
February 1982.

[9] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G.
LOCUS: A Network Transparent, High Reliability Distributed System.
In *Proceedings of the Eighth Symposium on Operating System Principles*. December, 1981.

[10] Rashid, R.F. and Robertson, G.R.
Accent: A communication oriented network operating system kernel.
In *Proceedings of the Eighth Symposium on Operating System Principles*. December, 1981.

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

[12] Satyanarayanan, M.
A Study of File Sizes and Functional Lifetimes.
In *Proceedings of the Eighth Symposium on Operating System Principles*. December, 1981.

[13] Satyanarayanan, M.
A Synthetic Driver for File System Simulation.
In *Proceedings of the International Symposium on Modelling Techniques and Performance Analysis, INRIA, Paris*. North-Holland, 1984.

[14] Satyanarayanan, M.
The ITC Project: A Large-Scale Experiment in Distributed Personal Computing.
In *Proceedings of the Networks 84 Conference, Indian Institute of Technology, Madras, October 1984*. North-Holland, 1985 (to appear).
Also available as ITC tech report CMU-ITC-035.

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

[16] Svobodova, L.
File Servers for Network-Based Distributed Systems.
Computing Surveys 16(4):353-398, December, 1984.

[17] Tichy, W.F. and Ruan, Z.
Towards a Distributed File System.
Technical Report CSD-TR-480, Computer Science Department, Purdue University, 1984.

[18] Voydock, V.L. and Kent, S.T.
Security Mechanisms in High-Level Network Protocols.
Computing Surveys 15(2):135-171, June, 1983.

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