The Secure File System Under Windows NT

By
Matthew Scott Rimer

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 1998

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 1999

Author ....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by ....
M. Frans Kaashoek
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The Secure File System Under Windows NT

By
Matthew Scott Rimer

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

SFS/NT is a Microsoft Windows NT client for the MIT Laboratory for Computer Science's Secure File System, a decentralized, distributed file system which utilizes strong public-key cryptography for authentication and encryption. SFS/NT is implemented as a Common Internet File System loopback server via the Framework for Implementing File Systems, and is compatible with Secure File System servers based on the UNIX operating system. SFS/NT provides support for symbolic links and case-sensitive names, which are features not commonly supported under Windows NT. It is structured as a stackable file system. Each component is located in a separate layer, which makes it simple to add or remove components, as well as to reuse components in other file systems.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank David Mazières for providing invaluable assistance with the Secure File System, and M. Frans Kaashoek for his support of this project.

1 Introduction

This thesis presents SFS/NT, an implementation of the MIT Laboratory for Computer Science's Secure File System for the Microsoft® Windows NT® operating system. This implementation is among the first significant academic research file systems to be brought to the Windows NT platform. In the past, the utility of Windows NT in the research of such systems was severely hampered by the complexity of the file system driver interface and researchers' relative lack of familiarity with it as compared to UNIX®. Recently though, a new framework, FIFS, was introduced for the creation of Windows NT file systems in the hope that it would simplify the process enough to make the platform a viable file system research alternative to UNIX. In this thesis, we seek to validate this claim via the implementation of SFS under Windows NT using the FIFS framework. We also seek to advance further development of the Secure File System by bringing it to a new and widely-used platform.

In the remainder of this section, we first briefly examine the Secure File System and the FIFS framework. This is followed by a discussion of the goals and contributions of the SFS/NT project.

1.1 The Secure File System

The Secure File System (SFS) was designed by and is currently under continued development by David Mazières of the Laboratory for Computer Science's Parallel and Distributed Operating Systems group. The version of SFS which is considered in this thesis is release 1.0, which is a later version of the system described in [11]. SFS is a distributed file system, which utilizes strong public-key cryptography to provide authentication and privacy. Unlike the Andrew File System (AFS) and the Distributed File System (DFS), SFS neither requires nor uses a centralized system to maintain a list of known servers in the distributed system or to prove the authenticity of those servers [8, 18]. Nor does it require a centralized directory to keep track of authorized users. Each individual SFS client and server negotiate with each other in order to confirm the identify of the user and the server.

Central to SFS is the notion of a *self-certifying pathname*. Each network-accessible file system has an associated public and private key pair, and is identified by such a pathname. In essence, the self-certifying pathname contains not only the network address of the server responsible for the file system, but also the server's public key. This permits the server to authenticate itself to a client by demonstrating that it possesses the private key associated with the public key named in the pathname of the client's requested file system.

Each user also has an associated public and private key pair. The public key serves as the user's "name" by which the server knows him. By demonstrating to the server possession of the private key, a user proves that he is the rightful owner of the public key. The server can then grant or deny access permissions based on the user's proven identity.

For security, the user's private key is never revealed to the SFS client. It is stored inside of a user agent, which is kept separate from the client. The client can request that the agent digitally sign an authentication request that can then be used by the client to prove the user's identity to the server. A rogue client, planted by an adversary, can never steal the user's private key because it never sees the private key. Although the agent must know the key, each user is free to supply their own trusted agent.

Following the exchange of identities between the client and the server, access to the secured file system is permitted. Such access is encrypted using symmetric key encryption for privacy. The symmetric keys used are agreed upon during the server authentication phase of the protocol.

In a global file system, it is useful to be able to tie file systems exported by different servers together into the appearance of a single namespace. In SFS, these ties are provided by symbolic links, which point to the self-certifying pathnames of other servers. When the user follows such a symbolic link, the client is expected to automatically establish a connection with the named server. The user is unaware of this transition, except to the extent that he may have different access rights on the new file system.

Originally, reference implementations of the SFS client and server were written for the OpenBSD platform. The client was implemented as a Network File System (NFS) loopback server. As a result, an SFS file system appears to the workstation to be a mounted NFS file system which is being served by an NFS server (namely, the SFS client) located on the local machine. The client translates NFS requests into SFS protocol requests and communicates with the SFS server across the network using the Open Network Computing Remote Procedure Call (ONC RPC) mechanism[1] and the External Data Representation (XDR) [19, 20].

---

[1] Formerly known as, and still frequently referred to as, the Sun RPC mechanism

1.2 The FIFS Framework

An NFS loopback server is a sensible implementation for SFS under UNIX and its descendents, given those platforms' built-in NFS capabilities. However, it is a poor choice for supporting SFS under Windows NT, as the operating system lacks any integrated NFS clients.

Nor is the alternative of building a native file system driver for Windows NT particularly attractive. Windows NT device drivers must conform to a complex, asynchronous I/O interface, be fully re-entrant, and use only a minimal stack. They must interface with Windows NT's I/O Manager, Cache Manager, and Virtual Memory Manager (see Figure 1). Being kernel-mode code, debugging such drivers can be quite difficult, and any failures will bring down the entire system [7]. Information about implementing file systems under Windows NT is scarce, and generally requires licensing an expensive, non-redistributable development kit from Microsoft, a barrier even in a research environment [12, 13].

These difficulties led to the development of a Framework for Implementing File Systems (FIFS) for Windows NT at the MIT Laboratory for Computer Science's Parallel and Distributed Operating Systems group [1]. FIFS runs entirely in user-mode and implements a Common Internet File System (CIFS) loopback server. Its design is shown in Figure 2. CIFS, also known as the Server Message Block protocol, is Window NT's native network file system protocol, and a client for it is supplied by Microsoft as part of the operating system [10]. This client communicates with the FIFS loopback server running on the local machine. In turn, FIFS makes function calls to a Windows dynamic link library, responsible for implementing the actual file system, via a well-defined file

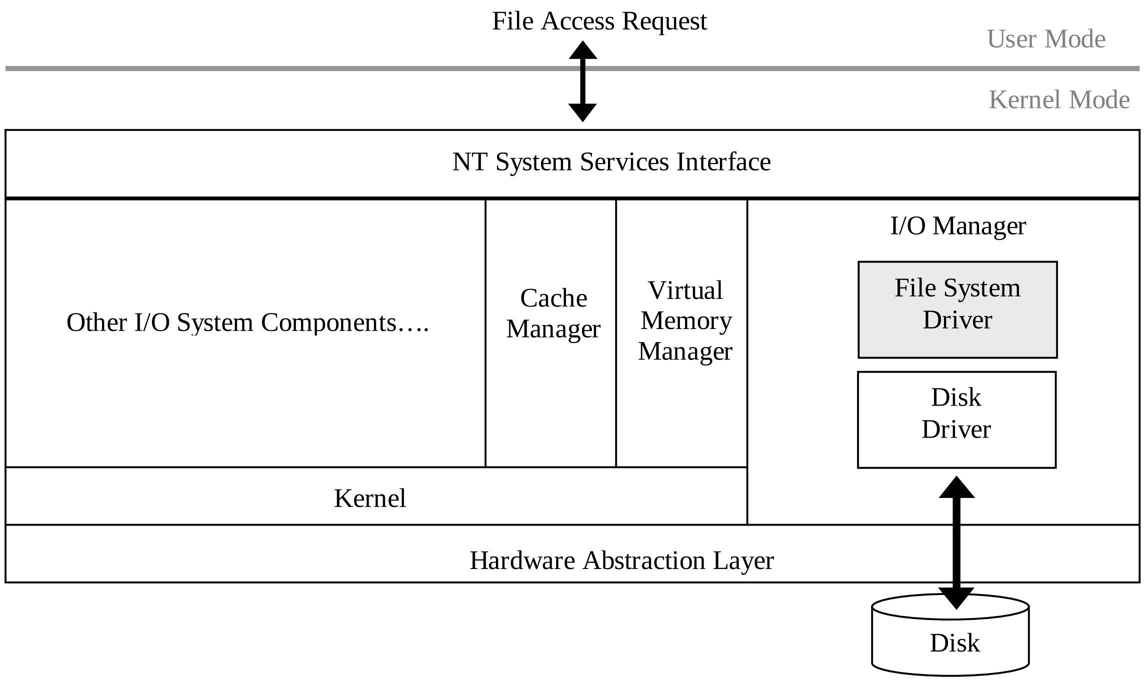


Figure 1: Windows NT File System Architecture (Adapted from [17])

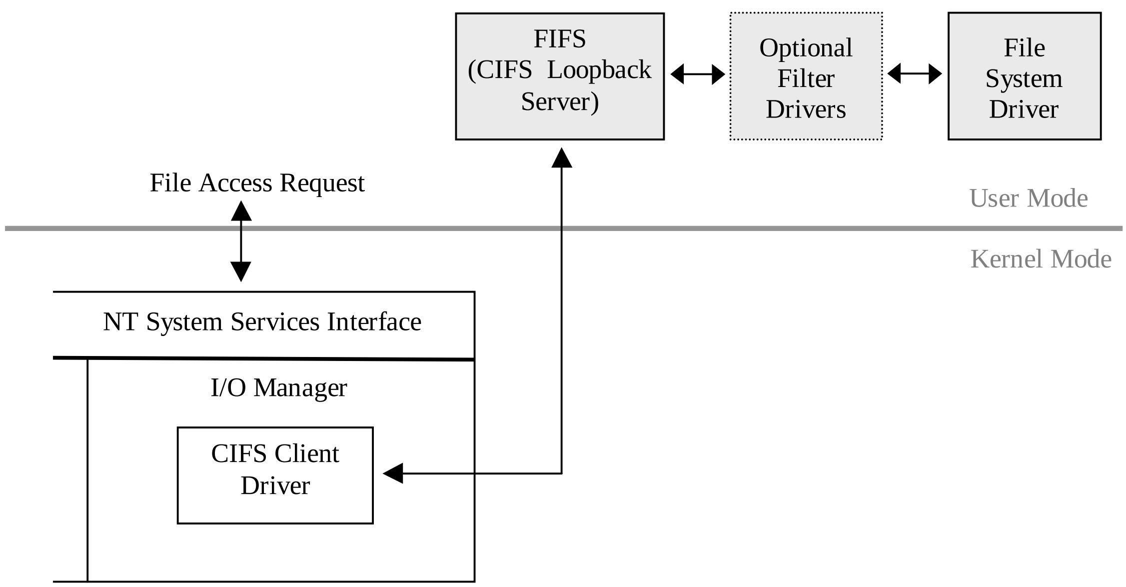


Figure 2: FIFS Architecture

system dispatch interface. This interface consists of high-level file operations such as create, read, and delete, which each file system “driver” implements as appropriate for its file system. Thus, new file systems can be developed as user-level code and then “dropped-in” to the FIFS framework for use.

In addition, the FIFS driver model uses stackable layers. It is possible to interpose one or more filter drivers between the FIFS loopback server and the file system driver. This makes it possible to encapsulate functionality in a way that is independent of both the loopback server and the particular file system driver being used. For example, a caching filter driver could be created, which would provide caching for any FIFS-implemented file system, without the need to rewrite the FIFS loopback server itself to implement caching.

1.3 Goals

The goals of SFS/NT are threefold. The first is to solve several issues that are important in any file system, but that are not addressed by the FIFS framework. The second is to bring SFS to the Windows NT operating system, thereby creating new opportunities to extend and evaluate the file system. The third is to validate the utility of FIFS itself as a tool for file system research. These goals are examined in more detail in the next three sections.

1.3.1 Resolve Issues Not Addressed by FIFS

While FIFS simplifies the process of developing Windows NT file systems, it leaves several issues open. These problems had to be solved in order for FIFS to fulfill its purpose as a tool for file system research.

First, FIFS lacks a locking mechanism and support for notification of changes to files or directories. As a result, it does not cache any information retrieved from the file system because there is no way for a file system driver to notify it when the information has changed. Caching, however, is generally considered critical to the performance of a distributed file system like SFS. If the file server is located on the other side of the planet, requiring the local computer to access it for every file operation could pose an unreasonable overhead.

Second, provision is made for neither symbolic links nor case-sensitive file names in FIFS. Both are identified as issues for further research in the original FIFS paper [1]. Both are essential if a FIFS-implemented distributed file system is to operate in a heterogeneous environment. Case sensitivity is vital to resolving Window NT's expectation of case-insensitive file names with the requirements of case-sensitive UNIX servers. Symbolic links are especially critical in SFS, since they permit a mapping from a human-readable name to a self-certifying pathname, and thus can link the file system on one SFS server to an SFS file system on a different server. Unfortunately, though, the CIFS protocol on which FIFS is built does not support symbolic links, so while FIFS permits a file system driver to expose functions for reading and creating links, it will never actually make use of those functions.

In this thesis, these issues are addressed and solved. Means of supporting symbolic links and case-sensitive names within any Windows NT FIFS-implemented file system are developed, as is a caching mechanism for the SFS/NT client.

1.3.2 Bring the Secure File System to a New and Widely-Used Platform

The best way to identify weaknesses, benefits, and opportunities for further research within a new system like SFS is to use it. SFS is an on-going research project in the Parallel and Distributed Operating Systems group, and part of that work involves expanding the range of systems which it supports, with the goal of eventually replacing NFS on all the group's systems. Adding SFS support to NT is an outgrowth of that project and was, in fact, the original reason for creating FIFS.

SFS/NT also makes it possible to test SFS in environments that do not base their infrastructure on the UNIX operating system. With some notable exceptions, such as AFS, file systems that originated in the research community have not made the transition to wide-spread deployment in the world at large. Part of this is no doubt due to the fact that many of these systems have been only available for UNIX, while much of the computing world is dominated by other platforms. By making SFS available to this wider community, it is hoped that the additional feedback and experience gained will help identify new opportunities for research.

1.3.3 Validate FIFS as a Tool for File System Research

The tests which have been done with FIFS prior to this thesis have focused on measuring its performance overhead as compared to a kernel-mode driver. It has been tested via the creation of a Windows NT file system driver, which simply accesses the machine's local file system via the Win32 API. A prototype NFS version two driver was also created for it, minus some functionality such as caching and support for symbolic links. However, no attempt had been made to use FIFS to implement a new and novel file system, its intended purpose. By using FIFS to create an SFS client, we establish its utility as a tool for researching and developing new file systems using the Windows NT operating system.

1.4 Contributions

This thesis contributes SFS/NT, an SFS client for the Windows NT operating system created using the FIFS framework. SFS/NT is designed as a stackable file system. It is composed of FSSFS, a layer that provides an initial implementation of the SFS client driver for FIFS, as well as FSSYMLINK and FSCASE, filter driver layers that provide support for symbolic links and case-sensitive file names in any FIFS-implemented file system. This implementation of SFS is believed to be among the first implementations of a significant academic research file system under the Windows NT operating system.

The SFS/NT client is fully interoperable with UNIX-based SFS servers. It supports SFS's encryption capabilities, as well as server and user authentication. It also supports case-sensitive names and the use of symbolic links to reference files and directories stored on either the same server as the link or on a different server.

At present, caching support in SFS/NT is implemented in FSSFS, and provides for write-through caching of attributes, names, and data. No write-back caching is performed, so all operations which modify the file system must be performed synchronously. FSSFS also lacks support for concurrent operations that require accessing the server. This reflects the lack of thread-safety in the only available freely-distributable ONC RPC implementation for Windows NT, and is not attributable to any fundamental design limitation of FSSFS [5].

1.5 Organization

In Section 2, previous work related to file system research using Windows NT is discussed. Section 3 examines the Secure File System and its related protocols in greater detail. The design of SFS/NT is discussed in Section 4, and its implementation is considered in Section 5. Section 6 looks at the performance of the initial implementation of SFS/NT. Finally, opportunities for future work are considered in Section 7.

2 Related Work

Minimal file system research has been done on the Windows NT platform. This may be partly due to the fore-mentioned difficulties involved in programming for its device driver interface, and partly because of the traditional focus on UNIX and UNIX-like operating systems in academic research. The Windows NT work that has been done has focused primarily on measuring and optimizing the performance of the native NTFS file system, rather than on developing new file systems.

Bradley *et al.* compared the performance of Windows NT, Windows for Workgroups, and NetBSD in the areas of system calls, program load, memory access, graphics bit-blitting, network throughput, and file systems [4]. The Windows FAT and NTFS file systems were compared with the Berkeley Fast File System. NTFS was found to have significantly more overhead when accessing files on disk or in the disk cache, but was faster at operations involving manipulation of file metadata. This latter result occurs because NTFS logs metadata changes in memory and delays writing them to disk, while Berkley FFS forces these changes immediately out to the disk.

Riedel, van Ingen, and Gray investigated means of optimizing NTFS for access to large sequential files using SCSI disks [14, 15]. The study found that a combination of techniques permitted achievement of half the maximum peak advertised performance of the hardware (which does not take into account system overhead, bus contention, and other issues that must be dealt with in realistic use but that are often ignored by system manufacturers in an attempt to make throughput figures look more impressive). This is a significant improvement over the “out-of-the-box” performance. The optimization techniques included using large I/O requests (at least 8 kilobytes each, and preferably 64 kilobytes), disabling file system buffering to minimize processor load on large requests, compensating for the loss of buffering by enabling write-caching on the SCSI disk controllers, and constantly maintaining a queue of asynchronous requests to keep the disk saturated.

Borr presented a mechanism for resolving the differences between the advisory locking mechanism of NFS and the Network Lock Manager, and the mandatory locking required by CIFS [2, 3, 10]. In an environment where servers for both protocols supply access to the same files, care must be taken to ensure that an NFS client does not violate the stricter locking semantics expected by CIFS clients. SecureShare, the system presented in the paper, implements and enforces a uniform locking protocol in such a mixed environment.

Stackable file system architectures, like those supported by FIFS and used in SFS/NT, are an active area of research. Heidemann and Popek set forth two basic principles of stackable architecture: symmetric interfaces and extensibility [6]. Symmetric interfaces require that the interface into a particular layer be the same as the interface out of that layer into the layer below. This makes it possible to add, remove, and reorder layers arbitrarily, without any syntactic constraints imposed by differing interfaces. Extensibility states that it should be easy to add new layers. Existing layers should not need to be rewritten every time a new layer is added. Similar principles had previously been incorporated into an extensible file system architecture for the Spring system [9]. A layered architecture is also used in the Windows NT I/O subsystem, making it possible, for example, to improve fault tolerance simply by inserting a disk mirroring layer between the NTFS file system driver and the disk hardware driver, without modifying either driver [17].

3 Secure File System

Each SFS server maintains a public and private key pair (p_s, s_s), and each SFS client maintains a key pair (p_c, s_c). The server's *self-certifying pathname* is of the form `/sfs/`**`Location`**`:`**`HostID`**. **`Location`** is the network name of the file server that exports the file system. **`HostID`** is a cryptographic hash of the server's host name and public key p_s. These pathnames are self-certifying in that an SFS server can prove its identity to the client, without relying on a third-party certificate authority or other form of centralized control, by demonstrating that it possesses the private key s_s associated with p_s. Any server other than the server for the named file system will not know s_s, and will therefore not be able to decrypt messages sent using the key p_s named in **`HostID`**. Since **`HostID`** is a cryptographic hash, it is computationally hard for a server to discover some key pair (p_s', s_s') such that p_s' and p_s hash to the same **`HostID`**. The best known techniques for doing so require trying a number of potential values for p_s' that is exponential in the length of the hash, and therefore require an infeasible amount of time for a hash of significant length.

Upon initiation of access to the file system by a client, the server proves its identity by supplying its public key p_s to the client, who verifies that a hash of p_s and the host name matches the hash, **`HostID`**, found in the name of the desired SFS server (see the top portion of Figure 3). The client then randomly chooses values for its portion of each of two session keys, k_{cs} and k_{sc}. These session keys are used to perform symmetric key encryption of data exchanges from the client to the server and the server to the client, respectively. Both the client and server contribute part of each session key, so that a faulty implementation of either will not compromise the security of the entire system.

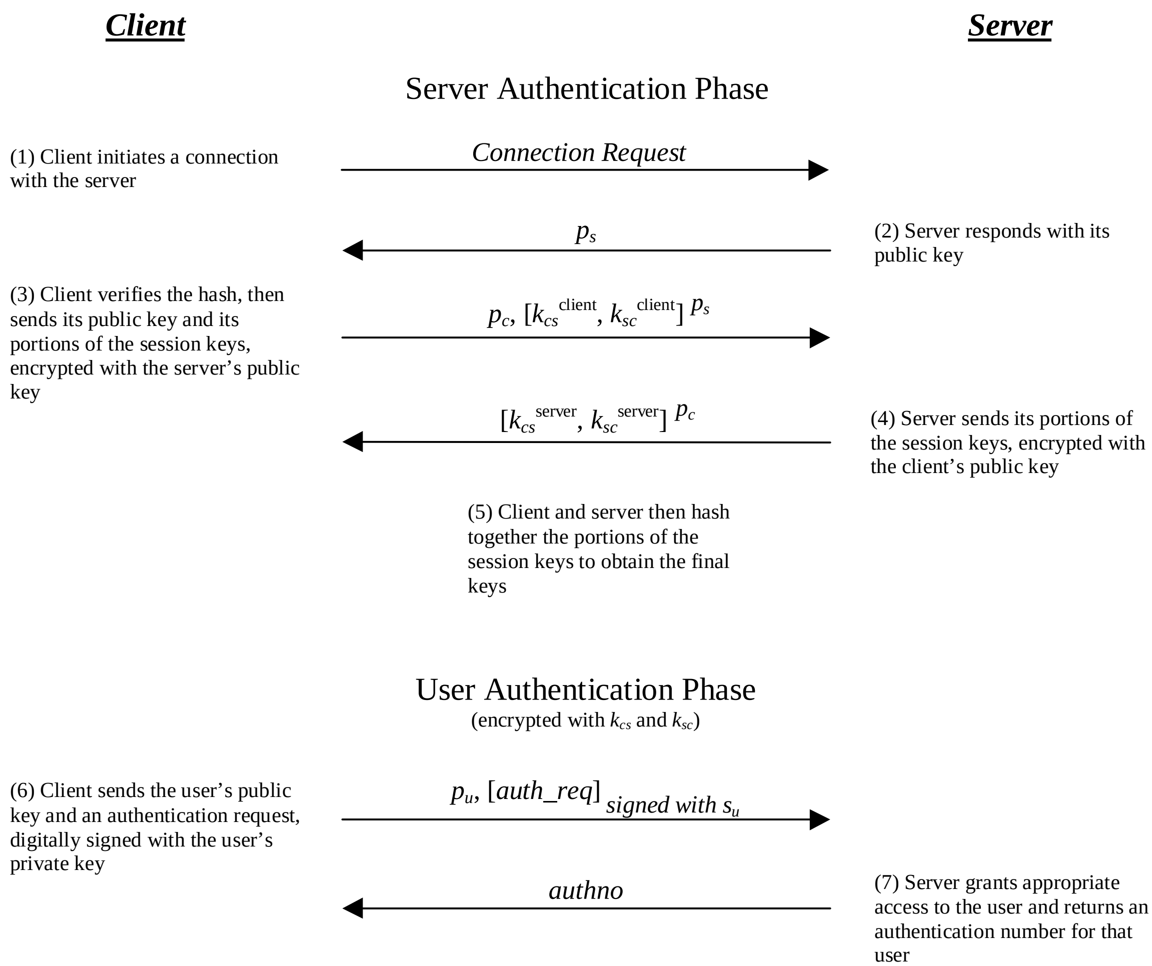


Figure 3: SFS Encryption and Authentication Protocol

The client sends p_c and its portion of the session keys to the server, encrypted using p_s. The server's ability to decrypt the session key components using the private key s_s, which only it possesses, is the proof of the server's authenticity. The server then chooses its own components of k_{cs} and k_{sc}, and sends them to the client encrypted with p_c for security. Both the client and server then hash together the session key components each decided on to arrive at the final k_{cs} and k_{sc}, which are used to encrypt all subsequent communications.

At this point, the client can then prove the identity of its user to the server, if necessary. This is shown in the bottom portion of Figure 3. Each user has their own public and private key pair (p_u, s_u) that may differ from the client's key pair. This key pair is used to uniquely identify a particular user of the client. In the UNIX implementation of SFS, the key pair is stored in a user agent, which is a separate process from the client and communicates with the client via remote procedure calls. These calls permit the client to request that the agent perform actions on its behalf, such as digitally signing a message, but do not permit the client to request a copy of the user's private key. Since the user's private key never leaves the confines of the agent, rogue clients are prevented from stealing the key and impersonating the user. Each user may provide their own agent if they do not wish to trust the default agent supplied with the SFS client.

To perform user authentication, the client provides the server with the user's public key p_u along with an authentication request that the user's agent has digitally signed using s_u. The server can confirm that this request did in fact come from the user who owns p_u by verifying the digital signature, and can grant appropriate access permissions based on the user's identity. It then returns to the client an opaque authentication number to be used as proof of the user's identity in all future requests. All communications during the user authentication phase are encrypted using the session keys that were chosen during the server authentication phase.

Once the server and client authentication is complete, file system access is provided via the SFS read/write protocol. This protocol is closely based on the NFS version three protocol with the addition of symmetric key data encryption, using the session keys, to provide privacy [3]. However, SFS is designed to be extensible, and it is possible to replace this read/write protocol with another protocol while keeping the same authentication and encryption protocol.

4 Design

In designing SFS/NT, we chose to make use of FIFS's layered driver architecture. SFS/NT consists of three separate drivers: FSSFS, FSSYMLINK, and FSCASE. This architecture is shown in Figure 4. The first implements the SFS protocol and provides access to the remote file system. The second is a filter driver which sits between FSSFS and FSCASE. It provides a symbolic link emulation layer that supports symbolic links to files and directories, including those whose targets are located on different SFS servers than the link itself. The third, FSCASE, is another filter driver, and adapts the case-insensitive naming convention of CIFS to the case-sensitive names of SFS.

The FSMUNGE layer shown in the figure is not part of SFS/NT, but rather is a filter driver supplied by FIFS. Given a file operation which names a file or directory using a pathname containing multiple parts (such as `/user/smith/thesis.txt`), it splits the pathname into its component parts and retrieves in turn a directory handle for each part of the pathname up to but not including the final part. It then performs the requested operation using the last directory handle retrieved and the final component of the pathname (e.g., the file named `thesis.txt` with the directory handle for `smith`). This saves the underlying layers from having to process multiple part pathnames, simplifying their implementation.

This layered approach avoids overburdening each layer with too much functionality, which would be both hard to maintain and conducive to errors. It also makes it easy to replace layers (for example, if a more efficient implementation is devised) or remove them altogether (either for testing purposes or, in the case of FSSYMLINK and FSCASE, if the underlying system were to be revised to natively

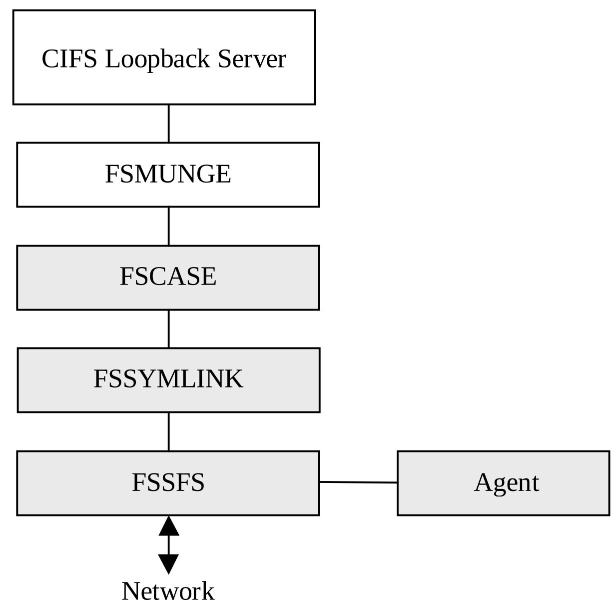


Figure 4: SFS/NT Architecture

support these features). Previous research on stackable file systems supports the idea that such a design is both practical and flexible [6, 9].

4.1 FSSFS

The FSSFS file system driver encapsulates all SFS-specific functionality. It is responsible for mounting the remote SFS servers, including negotiating encryption and user authentication. It is also responsible for translating FIFS file access requests presented via its file system dispatch interface to the NFS-like requests used by SFS, and for translating the results of these requests into a format understandable by the FIFS loopback server.

As part of user authentication, FSSFS is required to digitally sign an authentication request on behalf of the user using the user's private key, as proof of the user's identity. However, since multiple users may share a single computer and a single copy of FSSFS, it is questionable whether all of those users would (or should) want to share their private keys with FSSFS. Doing so would create the risk of one user secretly replacing FSSFS with a program that collects other user's private keys and steals their identity.

To prevent such an attack, we adopted a solution similar to that used by the UNIX version of the SFS client. FSSFS is never permitted to know the user's private key, which is encapsulated inside of a user agent. Instead, it must ask the agent to sign the authentication request for it. The user may supply her own agent, and so can be assured that no secret attempt to steal her private key is underway. The agent is discussed in more detail in Section 4.4.

4.2 FSSYMLINK

The FSSYMLINK filter driver is a symbolic link emulation layer. It provides support for symbolic links in a manner completely transparent to the FIFS loopback server and the Windows NT CIFS client which accesses the loopback server. When the FIFS loopback server retrieves a handle to a symbolic link, it is given a handle to the symbolic link itself. However, subsequent attempts to use that handle are mapped to operations either on the link or on the target of the link, as appropriate, by FSSYMLINK. FSSYMLINK provides the usual UNIX "semi-transparent" symbolic link semantics (e.g., a read operation on a symbolic link reads from the file pointed to by the link, while a rename operation alters the name of the symbolic link, not the name of the object that is the link's target). It also supports symbolic links whose targets are other symbolic links, and will follow the chain of links as necessary to reach the final target.

Since FSSYMLINK is a separate layer from FSSFS, and contains no SFS-specific details, it could also be used to supply symbolic link support in other FIFS-implemented file systems. On the other hand, if the CIFS protocol is ever extended to include symbolic link support, and if the FIFS loopback server is modified accordingly, the FSSYMLINK layer can be removed. No code changes to FSSFS will be required, and the user will see no change other than a possible performance gain from being able to bypass this emulation layer.

4.3 FSCASE

The FSCASE filter driver provides support for file systems which use case-sensitive file names. The CIFS protocol is case-insensitive but case-preserving. Unfortunately, though, implementation decisions made in the FIFS loopback server sometimes cause a failure to preserve the case of the pathname when requesting access to a file or directory [1]. Since the NFS protocol currently used by SFS is case-sensitive, presenting a name to it whose case has been "mangled" in this fashion will lead to erroneous file not found errors.

To solve this problem, upon receiving a case-insensitive name from FIFS, FSCASE reads the directory where the object is located and does a case-insensitive match between the desired name and the names returned by the read. If it finds exactly one match, it passes the name with the correct case to the driver beneath it. Otherwise, it returns an appropriate error code. Note that if there is more than one file or directory which matches, it returns an error as it has no way to determine which one the user is referring to. Making a mistake, such as deleting the wrong file, could be catastrophic, so this is the safest course of action. Experience suggests that directories which contain multiple entries whose names are identical except for their case are fairly uncommon. An alternative means of addressing this limitation is discussed in Section 7.3.

Operation	Description
AgentInitialize	Given the name of a user and a string identifying a key, returns a key handle for that user's public and private key pair
AgentDone	Given a key handle, terminates access to the key pair associated with that handle
AgentGetPublicKey	Given a key handle, returns the public key associated with that handle
AgentSignAuthinfo	Given a key handle and an authentication request, returns the authentication request signed using the private key associated with that handle
AgentPostSignAuthinfo	Called once the signed authentication request is no longer needed, to clean up any dynamically-allocated resources

Figure 5: SFS Agent Operations

4.4 SFS Agent

The agent is used during user authentication to digitally sign an authentication request, which permits an SFS server to check whether the request is in fact being made by the user who claims to be making it. While FSSFS itself could sign the request, doing so would require it to know the user's private key. An adversary could surreptitiously replace FSSFS with a modified version that collects and records users' keys, and thereby steals users' identities and accesses their private files. Clearly, this is an undesirable trait in a supposedly secure file system.

The solution is to separate FSSFS from the user's private key, and this is the purpose of the agent. In the UNIX implementation of the SFS client, the agent is a separate process which communicates with the client by means of remote procedure calls. In SFS/NT, the agent is a dynamically-linked library which supports the operations described in Figure 5. None of these operations require revealing the private key to the caller. Instead, after initializing the agent, an opaque handle to the key is provided to FSSFS, which can be used later to ask that a given key be used to sign a particular authentication request. The use of these handles permits a single agent to support multiple keys for multiple users.

Each user, if they wish, can specify their own agent to be called by FSSFS, as part of SFS/NT's per-user configuration settings. Furthermore, while if multiple agents are installed on one machine it is possible for one user to specify that her requests be signed by another user's agent, a well-designed agent should provide password protection or another suitable means of restricting access to the private key. Thus, while any user can specify any agent, only the user who legitimately owns the private key will be able to use it.

5 Implementation

The implementation of SFS/NT consists of a caching SFS file system driver, FSSFS; the symbolic link emulation driver, FSSYMLINK; and the case-sensitive driver, FSCASE. It also includes a prototype implementation of a user agent.

5.1 FSSFS Implementation

The implementation of FSSFS includes support for automatically mounting remote SFS servers on an as-needed basis. The client internally maintains a table of currently mounted servers. Whenever it tries to access an object named by a self-certifying pathname (for example, `/sfs/flex.lcs.mit.edu:97bch9th269cyb73u8896ihggztidpte/user/smith/thesis.txt`), it consults the table to determine whether the server (`flex.lcs.mit.edu:97bch9th269cyb73u8896ihggztidpte`, in this case) has already been mounted. If it has not, it mounts the named server, adds it to the table of mounted servers, and accesses the desired object on the new server. If the server is already mounted, FSSFS proceeds directly to accessing the named object.

In conjunction with FSSYMLINK, this automounting capability makes it possible to tie multiple SFS servers into a single, unified distributed file system via the use of symbolic links. A link located on one server can name a target that is the self-certifying pathname of a file or directory on a different server. FSSFS will automatically mount the necessary server upon the first attempt to access the link. From the point of view of a user, accessing a symbolic link that refers to an object on another server is no different than accessing a symbolic link whose target is located on the same server, except that her access rights on each of the servers may differ.

The automounting capability is also used to automatically recover from server failures, such as when a server crashes. When FSSFS is unable to access a previously mounted server, it marks it as dead in the server table and returns an error. Subsequent attempts to access a server labeled dead will cause FSSFS to try to remount the server. If it succeeds, unimpeded access to the server is restored. Otherwise, a suitable error is returned to the user, and the user may retry the operation, possibly at a later time, if she wishes.

The current implementation of FSSFS provides write-through caching of file data and of file and directory attributes. It also caches the contents of directories, to reduce the amount of time it takes to perform a directory read. Section 5.5 discusses the caching mechanisms of FSSFS.

5.2 FSSFS Configuration

The FSSFS driver is configured via settings stored in the Windows NT registry. These settings include what network protocol and port to use (only TCP is supported at the present time), as well as the self-certifying pathname of the initial SFS server to mount at startup. Settings are also provided to adjust the size of the caches and the length of time data is kept in the caches.

In addition to these global settings, additional settings are stored per user. These include the name of the user's agent, a string to be passed to the agent that identifies the user's key (to permit one agent to support multiple keys), and options for encryption and user authentication. These options permit the user to specify that the connection to the SFS server either must be encrypted, in which case failure to establish encryption during the mount procedure will cause the mount to fail; should preferably be encrypted, in which case fallback to a non-encrypted session is permitted; or should never be encrypted. A similar set of options are provided to control user authentication. These options exist to permit the individual user to choose between maximum security (permitting only encrypted connections) and convenience (falling back to an unencrypted session if encryption is unavailable), instead of hard-coding this decision into FSSFS. In the case of user authentication, the "never authenticate" option is useful if a user only needs access to publicly-available files and wishes to maintain her anonymity. If the user does not explicitly specify these options, FSSFS defaults to requiring encryption and user authentication. For additional security, compile-time options are provided that can be used to instruct FSSFS to ignore these settings and always require the use of encryption and user authentication.

5.3 FSSYMLINK Implementation

Since the CIFS protocol does not support symbolic links, it does not provide a symbolic link attribute for files. Thus, each directory entry returned by FSSYMLINK must be designated as either a file or a directory. Since Windows NT permits a user to perform different operations on an object depending on whether it has the attributes of a file or a directory, FSSYMLINK must determine whether the target of a symbolic link is a file or directory whenever it needs to return the attributes of that link, so as to be able to give the link the same type attribute as the object which is its target. Otherwise, if it were to return a symbolic link to a directory but mark it as a file, Windows NT would not permit the user to traverse the symbolic link into the target directory. This contrasts with UNIX, where it is not necessary to know the type of the symbolic link's target until a user actually tries to use that link.

Unfortunately, this requirement to know the type of a symbolic link's target up front can require extra RPC calls to the SFS server to be made every time the attributes of a symbolic link are retrieved, including every time the directory containing the symbolic link is listed. These RPC calls are performed by FSSFS to look up the attributes of the link's target in response to requests by FSSYMLINK. In the worst case, the target of the link is located on a different server that FSSFS must first mount. Links to the root directory of a server (e.g., `/sfs/flex.lcs.mit.edu:97bch9th269cyb73u8896ihggztidpte/)` do not require automounting to retrieve their target attributes, however, as they can be handled as a special case since FSSFS always knows that their target is, by definition, a directory.

The extra overhead caused by the frequent need to retrieve the target's attributes is reduced by FSSFS's attribute caching. By caching the file and directory attributes retrieved from the server the number of additional RPC calls required is reduced. This caching is discussed in Section 5.5.

5.4 FSCASE Implementation

Although the implementation of FSCASE is straight-forward, it shares with FSSYMLINK the requirement to perform extra RPC calls. These calls are done to read the directory every time a file or directory name is used, and thus for every file lookup or create operation, in order to search for a case-insensitive match. Worse, each directory read causes additional operations to be performed in FSSFS to retrieve the attributes for each object in the directory, since the directory entry format specified by FIFS includes not just the names of the objects, but their attributes as well. Like in FSSYMLINK, the potentially substantial overhead this could create when repeatedly accessing a large number of files is addressed by caching in FSSFS.

5.5 Caching in FSSFS

The frequent need of FSSYMLINK and FSCASE to retrieve attributes and of FSCASE to retrieve directory listings would impose a burdensome performance overhead if every retrieval caused a remote procedure call to be made. Measurement of an uncached version of FSSFS found that the addition of FSCASE made SFS/NT seventeen times slower when performing the Sprite LFS small file microbenchmark as compared to SFS/NT without FSCASE [16]. Furthermore, informal observations suggest that the Windows NT CIFS client tends to make a significant number of redundant requests in the course of a single operation. For example, in opening a 40 KB file using the WordPad text editor that is included with Windows NT, it retrieves the attributes of the file twelve times. Taken collectively, these results show the importance of caching in achieving acceptable performance from SFS/NT.

Unfortunately, at the time SFS/NT was developed, the SFS read/write protocol, like the NFS version three protocol from which it was derived, provided only weak support for caching. It did not provide a locking mechanism, and did not support strict cache consistency. It did support a weak form of cache consistency, in that operations which modify the state of the file system optionally returned the time the object was last modified, the time its attributes were last changed, and the size of the object prior to the operation. A client could use this information to determine whether a cached object had been changed by another client since it was last cached, and could update its cache if necessary [3].

Nonetheless, the popularity of NFS suggested that even this weak form of consistency was acceptable for many purposes. In a distributed file system like SFS, where servers may be scattered across a wide range of geographical locations with network connections of varying speed and reliability, caching can provide particularly significant performance gains.

Therefore, *ad hoc* caching support was added to FSSFS in order to make it possible to measure the performance offered by a caching SFS/NT client, after initial experiments with an uncached client yielded unacceptable performance. This caching takes three forms. The first form is an attribute cache. File and directory attributes that are returned by file system operations are stored in this cache and used to satisfy future operations that retrieve attributes. In addition, for symbolic links the name of the link's target is stored, to reduce the number of RPC calls required when accessing a file or directory through a symbolic link.

The second cache is a names cache which stores the contents of directories. For each cached directory, the names of every file and subdirectory in that directory are stored in this cache. Directories are added to the cache in response to directory read requests, and the cached results are updated as entries are added or removed to the directory. In conjunction with the attributes cache, once a directory is in the cache, a read request for that directory can be satisfied without the involvement of the SFS server, eliminating much of the overhead of FSCASE.

The third cache is a data cache for the contents of files. The cache manages data in blocks of a fixed maximum size. Each block contains a portion of a file that begins at an offset from the start of the file that is a multiple of the block size. Read operations read an entire block's worth of data at a time, which is stored in the cache to satisfy future reads. Write operations update the cached data as well as the file on the server.

All three caches use write-through semantics, and contain a fixed maximum number of entries at any time. The size of each cache, and the size of the data cache blocks, may be independently configured via the Windows NT registry. When a cache runs out of space, entries are evicted using a least recently used algorithm. Weak cache consistency is maintained both by using the weak cache consistency data supplied by the read/write protocol and by associating an age in seconds with each entry. When an entry's age exceeds the user-configurable maximum age for that cache, the entry is invalidated.

Very recently, work has been done on the SFS 1.0 read/write protocol to improve the support offered for caching. These changes have not yet been incorporated into SFS/NT. Section 7.1 discusses this and other future opportunities to improve caching in SFS/NT.

5.6 SFS Agent Implementation

The prototype implementation of the user agent in SFS/NT reads the user's key pair from disk upon a call to AgentInitialize. Each key pair is stored in a separate file, and the key identification string that is stored in the registry and passed to the agent (see Section 5.2) is the name of the file. The private key is optionally password protected using the Blowfish encryption algorithm to prevent anyone other than its owner from using it.

The key file is initially generated using a separate program, `sfskeygen`, which generates a public and private key pair for the user and, if the user desires, password protects it. The length of the public key in bits may be specified by the user. The format of the key file written by `sfskeygen` is compatible with the UNIX client's version of `sfskeygen`, so a user can transport their identifying keys between the UNIX and Windows NT SFS clients. This transportation may be safely done across an unencrypted communications link if the private key was encrypted at the time of key generation.

Since the agent is a dynamically-linked library used by FSSFS, it runs in the same process, and thus the same address space, as FSSFS. Therefore, it is theoretically possible for a rogue version of FSSFS to try to steal the user's private key by directly reading the agent's memory. However, it is possible to write an implementation of the agent that does not suffer from this weakness. The agent library need not handle the user's private key itself, but could instead be a thin wrapper to a separate agent process that runs in its own protected memory space. Furthermore, if the wrapper library and the agent process were to communicate using remote procedure calls (with appropriate encryption to prevent third-party interception of signed authentication requests), it would be possible for a user to isolate her private keys on a separate machine. Since the UNIX version of SFS uses an agent that is implemented in this fashion, it would be possible for a user to share one agent, running on a secured machine, between both SFS/NT and the UNIX SFS client.

6 Results

To determine whether a caching SFS client implemented using FIFS could yield acceptable performance, we made performance measurements using the four possible combinations of the three SFS/NT components: FSSFS by itself, FSSFS with FSSYMLINK, FSSFS with FSCASE, and FSSFS with both FSSYMLINK and FSCASE. For comparison, we also evaluated the performance of the local NTFS file system, and of accessing the local file system through FIFS using the FSWIN32 driver. FSWIN32 is a test driver supplied with FIFS that uses Win32 function calls to access the computer's local file system, and makes it possible to measure the overhead imposed by the FIFS framework [1].

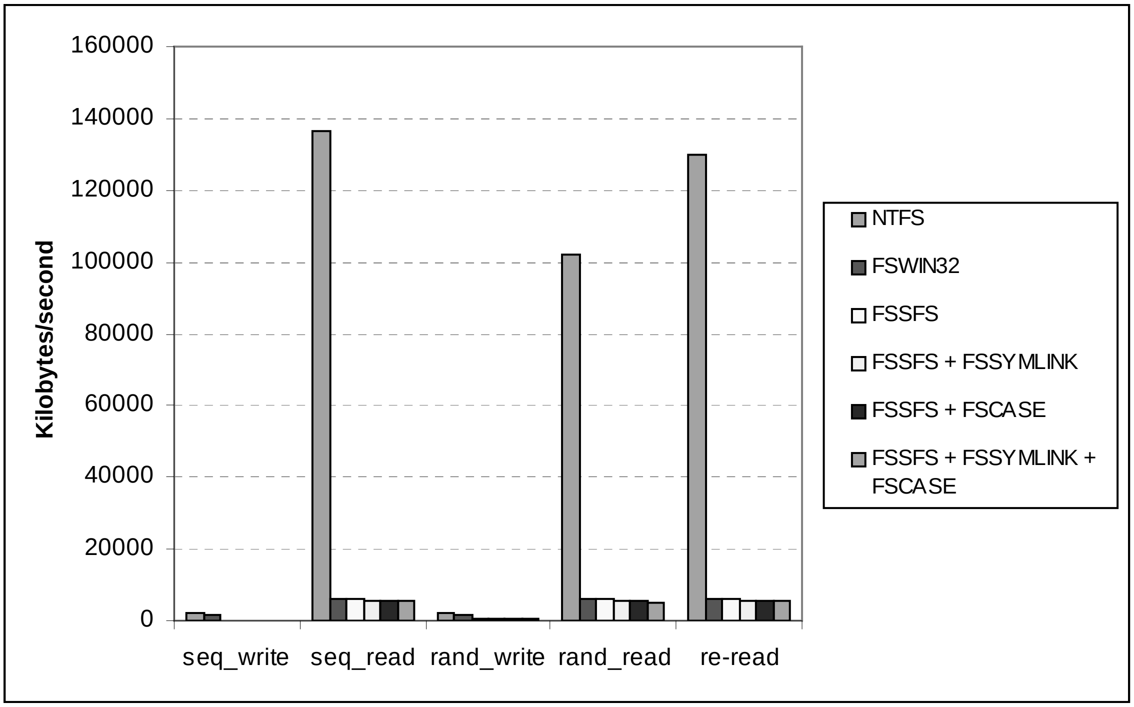
The Sprite LFS large file and small file microbenchmarks were used for the performance measurements [16]. Additional performance measurements were made using an application benchmark that attempts to simulate normal use of the file system. These benchmarks are the same tests which were used in [1], except that an updated copy of the FIFS source code was used in the application benchmark as the older source code could not be compiled with the most recent version of the compiler. The newer Windows NT header files installed by the compiler contained several type declarations that conflicted with declarations in the older version of the FIFS source code, and the addition of namespace qualifiers to the source code was required to resolve these conflicts.

The SFS/NT client machine was a 200 MHz Pentium Pro with 64 MB of RAM, running Windows NT 4.0 with Service Pack 3. The SFS server was a 200 MHz Pentium Pro with 128 MB of RAM, running the OpenBSD 2.5-Beta operating system. Both computers were equipped with SMC EtherPower 10/100 100 Mbps Ethernet cards, and were connected via 100 Mbps Ethernet switches. The SFS/NT data cache was configured to hold 1,024 8 KB blocks, while the sizes of the attribute and names caches were 1,024 and 256 entries, respectively. Blocks in the data cache expired after sixty seconds, and entries in the others caches expired after ten seconds. The cache sizes were chosen to balance performance with memory usage, and the cache timeouts were selected to realistically represent settings that might be used in a situation where file system consistency between multiple users must be considered. All benchmark tests were repeated three to six times to insure that observed results were not due to temporary anomalies, and the results obtained were averaged.

6.1 Large File Microbenchmark

The large file microbenchmark creates an 8 MB file using sequential writes, reads it sequentially, performs 8 MB of random writes, performs 8 MB of random reads, and finally rereads the entire file sequentially. The benchmark was performed using I/O sizes of 8 KB and 256 KB. The results are shown in Figures 6 and 7.

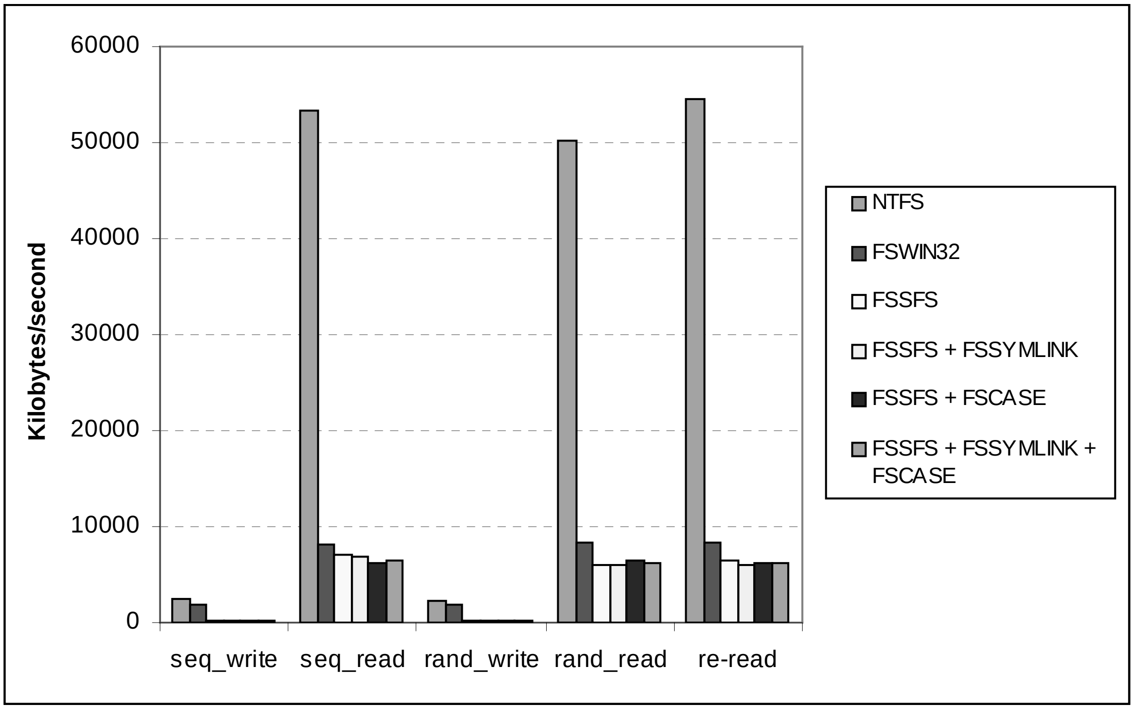
As expected, the SFS/NT throughput in general is less when using 8 KB I/O requests than 256 KB I/O requests, because of the greater number of requests that must be processed. The exception is the random write test, which yields identical results in both cases. It is unclear why this is the case. The results for FSWIN32 do not exhibit this behavior, so it is attributable to either the SFS client or server. Since the SFS/NT client only performs write-though caching, and because measurements of an entirely uncached version of FSSFS exhibited similar results, it may be an interaction with caching on the server.



	seq_write	*seq_read*	*rand_write*	*rand_read*	*re-read*
NTFS	2238.55	136338.67	2040.44	102254.29	129847.33
FSWIN32	1767.66	6119.95	1583.85	6166.08	6276.47
FSSFS	255.22	5883.51	307.85	5949.37	5933.52
FSSFS + FSSYMLINK	252.99	5401.03	306.73	5342.21	5325.57
FSSFS + FSCASE	249.95	5463.71	306.54	5378.52	5389.07
FSSFS + FSSYMLINK + FSCASE	256.92	5417.17	305.18	5294.70	5364.44

Figure 6: Throughput in kilobytes per second for Large File Microbenchmark (8 MB file using 8 KB I/O size)

In both the 8 KB and 256 KB cases, SFS/NT read throughput greatly exceeds the write throughput. Since SFS/NT uses a write-though cache, which was configured to hold 8 MB of data, the read operations can be performed entirely from local memory. All write operations, however, are performed synchronously and require accessing the SFS server, accounting for the poor performance. Reducing the size of the cache to be smaller than the test file would cause a reduction in read performance, as data would have to be fetched from the server to satisfy the reads. The greatest reduction would occur in the



	seq_write	*seq_read*	*rand_write*	*rand_read*	*re-read*
NTFS	2468.88	53399.05	2394.34	50124.06	54535.17
FSWIN32	1789.53	8033.90	1925.56	8405.14	8278.23
FSSFS	299.51	7022.47	307.44	5980.51	6495.65
FSSFS + FSSYMLINK	299.31	6865.24	306.20	6068.87	6026.04
FSSFS + FSCASE	285.00	6288.88	289.22	6363.44	6314.05
FSSFS + FSSYMLINK + FSCASE	299.02	6394.07	308.29	6339.50	6299.32

Figure 7: Throughput in kilobytes per seconds for Large File Microbenchmark (8 MB file using 256 KB I/O size)

case of the sequential read following the sequential write. Since the sequential read begins with the oldest data, which has already been evicted from the too-small cache to make room for more recently used data, virtually every read will cause a cache miss.

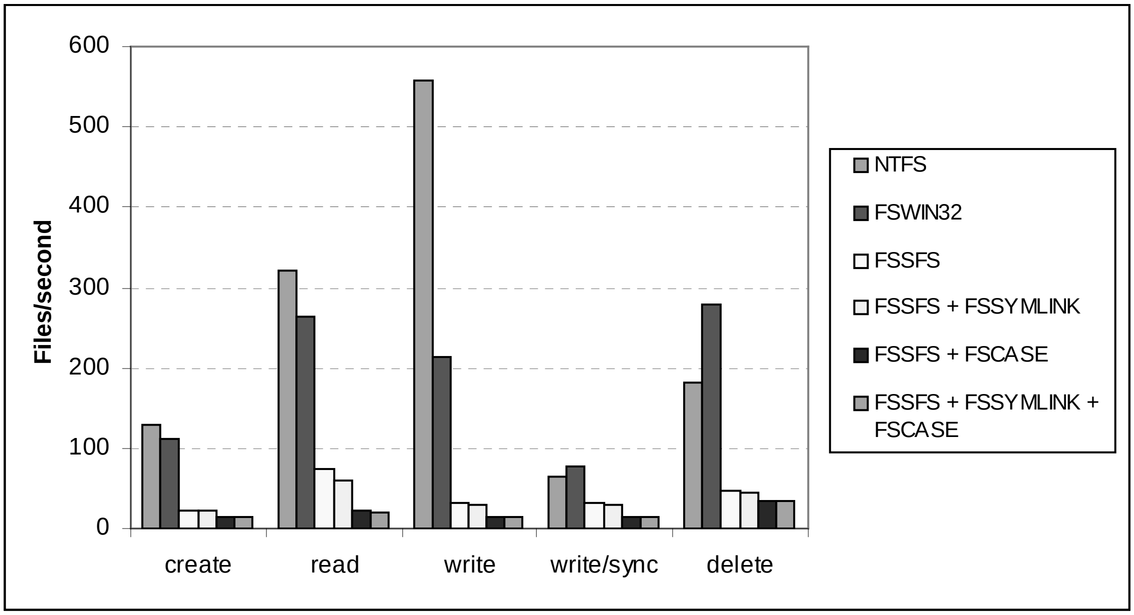
The high read performance for the NTFS local file system is due most likely to an efficient kernel-mode caching system. Since the test system had 64 MB of memory, NTFS can easily cache the entire test file. Interestingly, NTFS exhibits much better throughput in the case of both random and sequential reads when using 8 KB I/O than using 256 KB I/O. Why NTFS should exhibit such behavior is unclear. This result can not be related to the SFS/NT random write result, since the SFS servers do not use NTFS as their underlying file system. However, the FSWIN32 results show that any performance gained by using the smaller I/O size is exceeded by the performance lost by processing the increased number of CIFS requests.

In both the 8 KB and 256 KB cases, there was minimal performance difference between the four SFS/NT configurations. This is because the majority of time is spent performing read and write operations on the file contents, and neither FSSYMLINK nor FSCASE add any overhead to such requests. The variations in performance that are seen are due in part to the fact that the high read throughput relative to the size of the 8 MB test file causes any small deviations in timing (for example, if Windows NT's virtual memory manager should pick an inopportune time to flush a page to disk) to be greatly magnified and affect the overall results even after averaging.

6.2 Small File Microbenchmark

The small file microbenchmark creates 1000 1 KB files across 10 directories. It then reads each file, writes each file, writes each file again flushing the disk cache after each write, and finally re-reads each file. The results of this benchmark are shown in Figure 8.

Unlike the large file microbenchmark, the addition of FSCASE does cause a performance decrease. Even with caching, there is still some overhead associated with FSCASE performing a complete directory read on every operation that specifies a file or directory name, in order to find a case-insensitive match.



	create	*read*	*write*	*write/sync*	*delete*
NTFS	128.57	321.34	556.79	65.01	182.82
FSWIN32	112.71	264.13	214.32	78.14	279.02
FSSFS	23.12	73.53	31.77	31.98	46.70
FSSFS + FSSYMLINK	22.95	58.82	30.20	29.38	45.73
FSSFS + FSCASE	15.13	21.33	15.27	15.18	35.49
FSSFS + FSSYMLINK + FSCASE	14.44	20.42	14.84	14.85	35.21

Figure 8: Performance in files per second for Small File Microbenchmark (1000 1 KB files in 10 directories)

Unlike FSCASE, FSSYMLINK imposes a much smaller performance decrease. While there is some overhead each time a file is created or opened in order to determine if that file is a symbolic link, it adds no other cost when the file is not a symbolic link. Had this benchmark been performed instead with 1,000 symbolic links to other files, an additional performance decrease would have observed because of the need to read each symbolic link to determine its target and to open the target file.

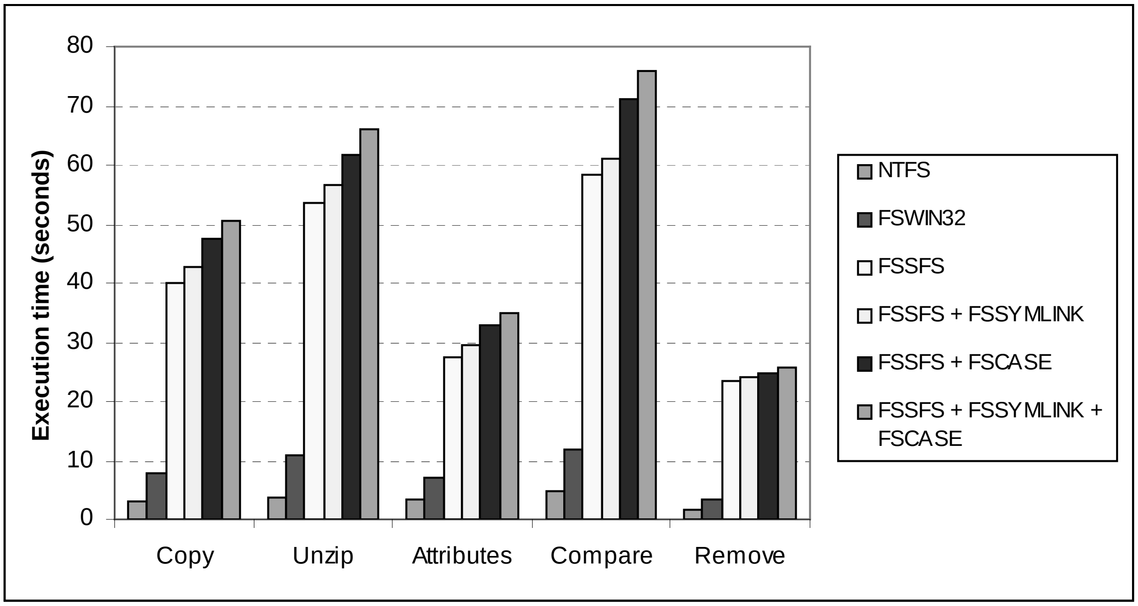
The SFS/NT results for the write and synchronous write phases of the benchmark are virtually identical. This is a consequence of FSSFS currently performing all writes synchronously. On the other hand, NTFS actually yields better synchronous write and delete performance when accessed through FIFS (the FSWIN32 case) than when used directly. The interposition of the CIFS client may provide additional caching that accounts for this result. Since Microsoft does not supply the source code for the CIFS client, however, it is difficult to determine this for certain.

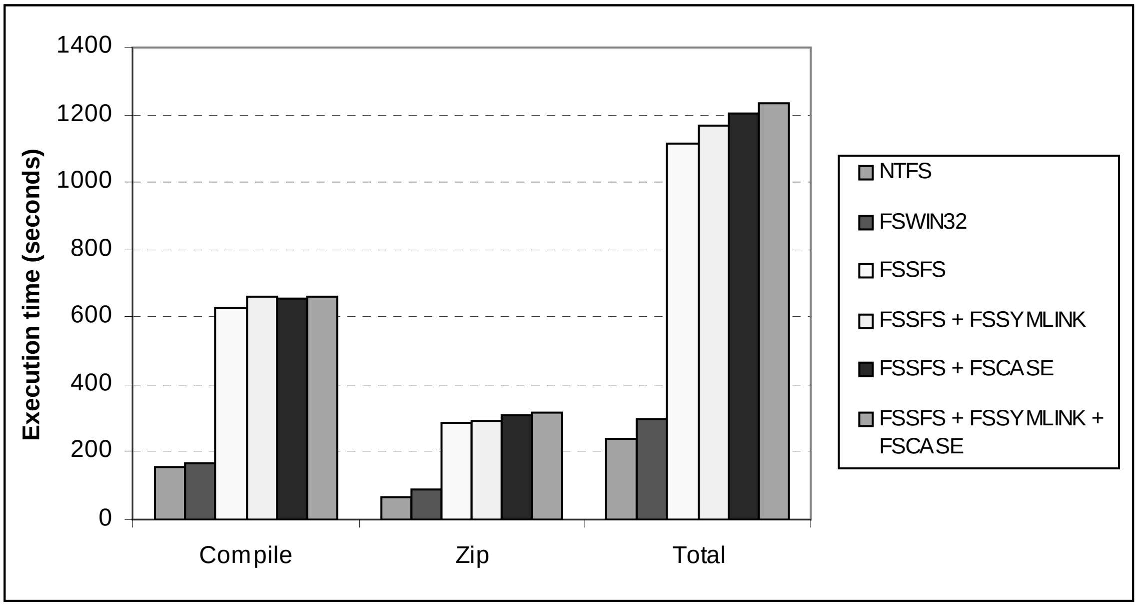
6.3 Application Benchmark

Unlike the LFS microbenchmarks which do not try to directly mirror real-world usage of a file system, the application benchmark attempts to measure performance in a realistic situation. It is modeled after the process of compiling a source tree on a remote machine. The source tree is a copy of the current FIFS source code and measures 870 KB in size. It contains 193 files, with an average size of 4.5 KB. It is stored as a 220 KB zip archive.

The benchmark begins by making a copy of the archive, unzipping it, and copying the source tree to a new directory. It uses the `du` utility to check the size of every file in the source tree, and uses `diff` to compare the new source tree to the old tree. Microsoft Visual C++ 6.0 is then used to build the project. The `du` and `diff` tests are repeated on the built source tree. The built tree is then zipped twice, once as a new archive and once as an update to the original zip file. The two source trees and the updated zip file are then deleted.

The results are shown in Figure 9. The "compare" category corresponds to the `diff` operations, and the "attributes" category reflects the time spent in the `du` operations. The remaining categories are self-explanatory. Overall, performance as compared to NTFS is limited by the relatively slow write throughput of SFS/NT. Stages such as





	Copy	*Unzip*	*Attributes*	*Compare*	*Remove*	*Compile*	*Zip*	*Total*
NTFS	2.95	3.60	3.43	4.65	1.64	155.65	66.04	237.96
FSWIN32	7.94	10.72	7.23	11.77	3.30	165.78	89.66	296.40
FSSFS	40.08	53.70	27.29	58.19	23.45	626.64	287.43	1116.78
FSSFS + FSSYMLINK	42.73	56.46	29.54	61.02	24.02	659.97	292.90	1166.64
FSSFS + FSCASE	47.55	61.78	32.86	71.07	24.89	656.85	309.99	1204.99
FSSFS + FSSYMLINK + FSCASE	50.53	66.24	34.94	75.93	25.69	662.45	317.26	1233.04

Figure 9: Execution times in seconds for Application Benchmark

compare, which do not perform any writing, are limited by the fact that the cached data only stays valid for sixty seconds. By the time the compare stage is reached, the time limit has expired and the data must be refetched. Increasing the data cache timeout so this is no longer an issue decreases the time of the comparison stage by one-third, but increases the risk of cache inconsistency if multiple users are modifying the same files simultaneously.

The results among the four SFS/NT configurations follow a pattern similar to that of the small file microbenchmark. The FSCASE-induced performance decrease is least in those phases of the benchmark which place a strong emphasis on reading and writing, such as compilation or zipping, and greatest in those phases which emphasize just retrieving attributes. In the former case, the addition of FSCASE to FSSFS and SFSYMLINK yields a 8% average slowdown, while in the latter case there is approximately a 25% decrease.

Both of these measurements are well below the average 50% decrease in performance that occurred in the small file microbenchmark, however. This reflects the fact that the application benchmark performs actual work (compiling the source tree) in addition to accessing the file system, rather than merely creating and deleting files as fast as possible. This, in turn, shows that while the raw performance of the file system may be of great importance, as is visible in the difference between NTFS and SFS/NT, it is not the only factor affecting the overall performance of the system. The use to which the system will be put must also be considered. In the case of SFS/NT, if the file system is to be used primarily for sequential reads of large files (streaming video, for example), then the inclusion or removal of FSCASE produces no significant performance difference, as shown by the large file microbenchmark. On the other hand, if the file system is to be used primarily in a situation which more closely resembles that of the small file microbenchmark, then the performance difference may be worth consideration. But in either case, the use of a layered design for SFS/NT permits the user to choose which file system modules to include based on performance and functionality requirements, without the need to edit any source code or to recompile the file system.

7 Future Work

The current implementation of SFS/NT permits access to Secure File System servers by computers running the Windows NT operating system. However, opportunity for future improvement remains. In this section, several of these opportunities are discussed.

7.1 *Improved FSSFS Caching*

The current FSSFS caching can be improved. Replacing the current write-through mechanism with write-back could yield substantial performance improvements by permitting writes to take place asynchronously. This should make the performance of SFS/NT more competitive with other available network file systems for Windows NT that exploit caching, while still avoiding the complexity of a kernel-mode device driver and the Windows NT Cache Manager.

At a more fundamental level, work has recently begun on improving the SFS read/write protocol's caching support, with the aim of making it possible to perform aggressive caching while still maintaining cache consistency. Initially, this work has taken the form of permitting an SFS server to grant a client a lease to cache the attributes of a file or directory for a certain period of time, and to recall that lease if the attributes should be modified during the lease period. These protocol enhancements have not yet been incorporated into SFS/NT. If they are, they should make it possible to provide the user with improved cache consistency while maintaining performance equal to, if not better than, the current caching mechanism.

7.2 Modularity of the SFS Protocol

SFS was designed to be extensible, by providing a means to add support for new file protocols within the existing security framework. The SFS protocol is conceptually separated into two components. The first is responsible for negotiating encryption and authentication, and never changes. The second provides for performing operations on the file system, and may be replaced as desired. For example, the current NFS-derived protocol could be replaced by one optimized for read-only access on an SFS server that exports a static file system.

In its current incarnation, FSSFS is a single module which implements both components of the SFS protocol. However, it would be straight-forward to redesign it to mirror this separation. Doing so would make it easier to write support for new SFS file protocols as they are added. One potential approach is to use FIFS's support for layered drivers, and implement the encryption and authentication portions of the protocol as a filter driver which calls through to an underlying driver that implements a particular file protocol.

7.3 Better Disambiguation for Case-Sensitive Names

As was discussed in Section 4.3, FSCASE has the drawback that when there are two files in one directory with names that differ only in case, it does not permit access to either file. This is for safety, as it has no way to know which file the user intended to operate on as the case information has been lost (which is why FSCASE is needed in the first place). But, while this may be safe, it is clearly not optimal.

Instead, a means of disambiguating such file names without relying on case is needed. Appending an identifier to the names of these files in the FSCASE layer could serve this purpose. These identifiers would only be a part of the local view of the file system presented by SFS/NT, and would not be stored on the SFS server. One approach is to append a number that represents the lexicographic order of the file names. For example, a directory containing the files `thesis`, `Thesis`, `ThesiS`, `problemset`, `ProblemSet`, and `notes` would appear to contain, when listed:

```
thesis~1          problemset~1
Thesis~2          ProblemSet~2
ThesiS~3          notes
```

Furthermore, attempts to access such a file name would be mapped by FSCASE to the actual name on the server.

Such an implementation would suffer from a problem, however. If one user was to list the directory and another user was to then delete the file `Thesis`, `ThesiS` would change from `ThesiS~3` to `ThesiS~2`. If the original user then attempted to delete `Thesis~2`, without knowing of the second user's action, he would delete the file `ThesiS` instead of the intended file `Thesis`.

An approach which fixes this problem, at the expense of longer identifiers, is to directly encode the capitalization in the identifier. Using hexadecimal identifiers in which the most significant bit corresponds to the first character of the file name, a 1 bit corresponds to a capital letter, and a 0 bit corresponds to a lower case letter, number or symbol, the above example becomes:

```
thesis~0          problemset~0
Thesis~20         ProblemSet~204
ThesiS~21         notes
```

This does not resolve a second problem, though, which is shared by any disambiguation scheme that attaches additional information to the file names. Suppose a user creates a reference, in a configuration file or some other location, to the file `notes`.

If a second file named `Notes` were to be created in the same directory, possibly without the knowledge of the user, the use of identifiers would be required to access either of the files and the pre-existing reference to `notes` would become broken. Short of always requiring the use of identifiers even when there is no current ambiguity, which would be unacceptable to most users, it seems that any identifier-based method will have this limitation.

8 Conclusion

SFS/NT has demonstrated the usefulness of the FIFS framework. It was certainly easier to develop and debug the file system as a user mode application than it would have been as a kernel mode device driver. The file system dispatch interface which it exposes is sufficient for developing a usable file system, and its stackable driver architecture made it relatively straight-forward to develop support for symbolic links and case-sensitive file names. The addition of caching to the FIFS-implemented file system also proved to be fairly simple. However, although additional work could be done to improve the caching in FSSFS, building support instead for a cache mechanism into the framework would permit all FIFS file system drivers to benefit without each driver having to implement its own cache.

SFS/NT has succeeded in bringing Secure File System access to the Windows NT platform. We hope that, with further work on caching, additional performance improvements will be obtained over the current results, making the performance of SFS/NT more competitive with that of other available file systems.

9 References

[1] Danilo Almeida. Framework for Implementing File Systems in Windows NT. Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science Master's Thesis, May 1998.

[2] Andrea J. Borr. SecureShare: Safe UNIX/Windows File Sharing through Multiprotocol Locking. In *Proceedings of the 2nd USENIX Windows NT Symposium*, USENIX, August 3-4, 1998.

[3] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, Network Working Group, June 1995.

[4] J. Bradley Chen, et al. The Measured Performance of Personal Computer Operating Systems. *ACM Transactions on Computer Systems.* 14(1):3-40, February 1996.

[5] Martin F. Gergeleit. *ONC RPC for Windows NT.* Manual comes with software distribution.

[6] John S. Heidemann and Gerald J. Popek. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems.* 12(1): 58-89, February 1994.

[7] Galen Hunt. Proxy Driver Home Page. From `http://www.research.microsoft.com/os/galenh/proxy/`, November 1998.

[8] Michael L. Kazar, et al. DEcorum File System Architecture Overview. In *Proceedings of the Summer 1990 USENIX*, USENIX, June 11-15, 1990.

[9] Yousef A. Khalidi and Michael N. Nelson. Extensible File Systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, ACM, December 1993.

[10] Paul Leach and Dilip Naik. A Common Internet File System (CIFS/1.0) Protocol. Internet-Draft, IETF, December 1997.

[11] David Mazières. Security and Decentralized Control in the SFS Global File System. Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science Master's Thesis, August 1997.

[12] Microsoft Corporation. Microsoft Windows NT IFS Kit. From `http://www.microsoft.com/hwdev/ntifskit/default.htm`, November 1998.

[13] Rajeev Nagar. Windows NT File System Internals: A Developer's Guide. O'Reilly and Associates, Sebastopol, CA, 1997.

[14] Erik Reidel, Catharine van Ingen, and Jim Gray. A Performance Study of Sequential I/O on Windows NT(TM) 4. In *Proceedings of the 2nd USENIX Windows NT Symposium*, USENIX, August 3-4, 1998.

[15] Erik Reidel, Catharine van Ingen, and Jim Gray. Sequential I/O on Windows NT^{TM} 4.0 – Achieving Top Performance. From `http://research.microsoft.com/barc/Sequential_IO/SEQIO.doc`, November 1998.

[16] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, ACM, October 1991.

[17] David A. Solomon. Inside Windows NT, Second Edition. Microsoft Press, Redmond, WA, 1998.

[18] Mirjana Spasojevic and M. Satyanarayanan. An Empirical Study of a Wide-Area Distributed File System. *ACM Transactions on Computer Systems.* 14(2): 200-222, May 1996.

[19] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Network Working Group, August 1995.

[20] R. Srinivasan. XDR: External Data Representation Standard. RFC 1832, Network Working Group, August 1995.