

Designing and Implementing Kiwi:
A Secure Distributed File System over HTTPS

Austin Che
austin@cs.stanford.edu

Computer Science Department
Stanford University

May 2001

Designing and Implementing Kiwi: A Secure Distributed File System over HTTPS

Austin Che
austin@cs.stanford.edu

Abstract

Kiwi is a distributed file system designed to allow for secure access to files globally. Kiwi uses HTTP over SSL, HTTPS, the industry standard protocol for secure transactions over the World Wide Web, and the WebDAV extensions to HTTP. By building on existing code and protocols, we were able to develop quickly the foundations necessary for a simple to use yet secure file system. The choice of HTTPS as a protocol gives several additional benefits including the ability to access files securely from behind firewalls and to access files with nothing more than a standard web browser. The current implementation includes a file system module for Linux with most of the standard file system functionality implemented. Additional features include setting up personalized views of a global namespace and working with encrypted files. The current status and possible future directions for Kiwi are discussed.

Thesis submitted for the Degree of Bachelor of Science in Computer Science with Honors at Stanford University, and advised by Professor Monica Lam. Additional thanks go to Constantine Sapuntzakis and Eric Ketchum for their help and advice.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Current Status	6
1.3	Thesis Organization	6
2	Background	7
2.1	File Systems Background	7
2.1.1	Implementation	7
2.1.2	Mounting	8
2.2	Types of File Systems	8
2.2.1	Local File Systems	8
2.2.2	Distributed File Systems	9
2.2.3	Cryptographic File Systems	10
2.3	File System Issues	12
2.3.1	Sharing	12
2.3.2	Caching	13
2.3.3	Fault Tolerance	14
2.3.4	Scalability	15
2.4	Existing File Systems	15
2.4.1	Local File Systems	15
2.4.2	Distributed File Systems	16
2.4.3	Cryptographic File Systems	22
3	Design	25
3.1	Protocols	25
3.1.1	HTTP	25
3.1.2	WebDAV	27
3.1.3	SSL	27

3.2	Security	29
3.2.1	Authentication	29
3.2.2	Access Control	30
3.2.3	File Encryption	30
3.2.4	Trust Model	31
3.2.5	Firewalls	31
3.3	Simplicity	32
3.4	Scalability	32
3.5	Summary	33
4	Implementation	34
4.1	Kiwi Server	34
4.1.1	Apache	34
4.1.2	Additional Modules	35
4.1.3	Dynamic Generation of Certificates	36
4.1.4	File Access With Web Browser	38
4.2	File System Client	39
4.2.1	Kiwi protocol	39
4.2.2	Usage	40
4.3	Kernel Module	40
4.3.1	VFS	41
4.3.2	Reading Directories	41
4.3.3	Requests	42
4.4	Kiwid	42
4.4.1	File Encryption	44
4.4.2	Path Translations	45
4.4.3	Caching	47
4.5	User Client	48
4.5.1	add	48
4.5.2	cat, cp, ls	49
4.5.3	encrypt/decrypt	49
4.5.4	list	49
4.5.5	shell	49
4.5.6	shutdown	50
4.5.7	sign	50

5	Results	51
5.1	Security	51
5.2	Simplicity	52
5.3	Scalability	53
5.4	Performance	53
5.4.1	Benchmark Tests	54
5.4.2	Potential Optimizations	55
5.4.3	Other Considerations	57
6	Conclusion	58
6.1	Further Work	58
6.1.1	File Sharing	58
6.1.2	File Migration	60
6.1.3	Caching Strategies	61
6.1.4	File Encryption	62
6.1.5	Key Management	62
6.1.6	Certificate Revocation	63
6.1.7	Mobile Computing	63
6.1.8	Ubiquitous Access	63
6.2	Final Thoughts	64

Chapter 1

Introduction

1.1 Motivation

Kiwi is a file system designed with the goals of security, simplicity, and scalability. In addition, we wanted to have files globally and widely accessible. Chapter 2 presents an overview of many existing file systems. However, none of these existing file systems were readily available and could suit our needs for a globally accessible and secure file system. Existing distributed file systems often do not have any security features or have security added as an afterthought. Some cryptographic file systems do have security as a design goal, but usually work on top of other non-secure file systems and may not be very easy to use.

For Kiwi, we chose protocols standard on the Web, using HTTP, WebDAV and SSL. HTTP is an extensible and simple protocol, WebDAV extends HTTP, allowing us to implement a read/write file system, and SSL gives us the desired network security. By using existing protocols and code, we found development much simpler and quicker than it might otherwise have been had we used a custom designed protocol. As further motivation, here are some sample scenarios where Kiwi may be useful in practice.

- *Web Browser Access to Files*

When traveling away from one's personal computer, a need arises to access important files. The nearest machine with a web browser is found and one points the browser to the file server. After appropriate authentication, secure read and write access to all files becomes available.

Nothing needed to be carried around, except maybe a pass phrase in one's head and a smart card in one's wallet.

- *File System Interface*

A web browser is a reasonable means of file access when away from one's personal computer, but is too cumbersome for dealing with files on a day-to-day basis. A distributed file system to securely access files is desirable. Authentication is done using certificates.

- *File Access Behind Firewall*

Even with a firewall between the user and the file server, files should be accessible without modifying the firewall or opening up a security hole.

- *Ease of Administration*

For the system administrator of a large organization or group, an easy way to manage files and users in a scalable manner is strongly needed. Users should be able to authenticate via a central source and gain access to files on all servers in the system.

1.2 Current Status

The core Kiwi system, including some extra features such as file encryption, have been implemented. A working, usable, and freely available implementation exists for the Linux operating system. Compared with similar file systems, the performance of Kiwi is still relatively sluggish. However, many optimizations have yet to be done, and, in addition, Kiwi has useful features not found in other file systems.

1.3 Thesis Organization

This thesis is divided into six chapters. Chapter 1 contains this short introduction with the motivation behind the Kiwi file system. Chapter 2 includes background about existing file systems. Chapter 3 gives an overview of the design of Kiwi and Chapter 4 discusses the details of implementation. Chapter 5 presents results and the comparative performance of Kiwi. Finally, Chapter 6 concludes with a discussion of areas for further work.

Chapter 2

Background

2.1 File Systems Background

File systems serve several main purposes:

- Storing information
- Retrieving information
- Sharing information

File systems are useful for long-term and persistent storage of information and are a way to organize information, usually in the form of files and a file hierarchy. A *file* consists of mapping a name to a block of information accessed as a logical group. Many ways exist to implement file systems and many file systems have been implemented, but relatively few of these are widely used. This chapter will give some general background about existing file systems.

2.1.1 Implementation

Normally, file systems are implemented as part of the operating system's functionality, in the kernel. Because they are implemented at a low level, applications should work with various file systems with no modifications on their part. Applications use a common interface provided by the operating system for working with files, while the operating system controls the semantics of this file access, whether it may involve reading a local disk block

or contacting a remote server to retrieve a file. The sections to follow will discuss mostly UNIX-based operating systems and file systems, because understanding how these work will form the basis for understanding the Kiwi file system.

2.1.2 Mounting

Many different types of file systems can co-exist on one machine. Telling an operating system the file systems to be used is done through the process of *mounting*. On UNIX, the user can request a file system like foofs be mounted under a directory like `/foo`. The directory `/foo` is called the *mount point*. All accesses to files under `/foo` will then be interpreted as a request for a file on the foofs file system. Note the operating system must first have the necessary code to work with a foofs file system to successfully mount a file system of type foofs. Mounting allows for accessing a variety of file systems from a single file hierarchy. Only a *superuser*, someone with system administration privileges, can perform mount operations, but, once mounted, all users on the system can use the file system.

2.2 Types of File Systems

Several broad distinctions can be made among file systems. Below is a general overview of how local file systems, distributed file systems, and cryptographic file systems are designed to work, and this overview will serve as a basis for the remainder of the work.

2.2.1 Local File Systems

Many file systems exist to access files on raw disks. They all have the common purpose of mapping file names to a collection of bytes on a disk. From the disk interface that allows access to blocks, local file systems need a way to arrange the blocks so they appear in a meaningful file structure. Below is a short description of how an idealized UNIX file system may be organized as many file systems follow the original model [23].

UNIX File System

The file system structure recording information relevant to the entire file system is called the *superblock*. The superblock contains information such as how many times the disk has been mounted and the location of the inode table. The inode table contains a list of the inodes on the system, where an *inode* is the meta-information stored along with each file. Inside an inode are pointers to the blocks on disks containing the data for the file. The inode also includes information such as the file's owner, file permissions, and last modification time. Inodes are numbered, but users would prefer to use symbolic names instead of inode numbers in referring to files. The function of directories is to map names to inodes. Multiple file names can map to the same inode, allowing for hard links, where the same data can be accessed through different file names.

2.2.2 Distributed File Systems

Local file systems are limited to files located on a single machine. Hence, users have no mobility to move between machines and maintain access to one's files. Distributed file systems work over a network and can therefore allow access to files located on separate machines. In a distributed file system, there is usually the notion of a server and a client. The server is a machine providing a service to others, while the client uses this service. Note the same machine can act both as a server and a client in some file systems.

Ideally, a network file system should be distributed over many computers, but behaves much as a local file system would. This is called *network transparency*, meaning the client interface should be identical for remote files and for local files [15]. UNIX handles this with a common mount interface for both local and distributed file systems.

One way to characterize network file systems is the granularity of data access on the server. At one extreme, the server provides only a raw disk interface. In this case, the network file system can work very similarly to a local file system, but instead of reading blocks from a local raw disk, blocks are read from a network raw disk. At the other extreme, the file server presents a file interface, and clients send requests for whole files and not specific blocks of data.

Having the file server work with blocks means the file server can be very dumb. Most of the work with maintaining file information, such as the

information commonly found in inode structures, is maintained by the clients. This is appropriate for file systems where the server does not know about the contents or structure of files, as in some cryptographic file systems. However, transmitting all this information over the network can increase the network traffic overhead and slow the system down. As blocks from the same file are often accessed together, having the file server return entire files makes sense. But this method can also be inefficient, especially with large files and when only small chunks of the file is being accessed. Thus, many existing distributed file systems work with something in the middle, allowing for retrieving blocks within files.

2.2.3 Cryptographic File Systems

Both local and distributed file systems do not usually provide cryptographic services. One cryptographic service is to encrypt the link between the server and client, thus preventing unauthorized people from listening in on the network communication. Although encrypting the network data stream protects against sniffers on the network, clients must still trust the server, as the server sees the unencrypted file. In many environments where the user does not control the server or many people have control of the file server, this trust is problematic. Even if the system administrators are trusted, these centralized file servers become an attractive target for crackers who gain access to many files if the system is compromised.

Encrypting the file on the clients prevents the server from gaining much knowledge about the information stored in the files. Although possible to perform manual file encryption, e.g., with the UNIX `crypt` command, this type of approach to encryption is cumbersome and error-prone. One may forget to delete the unencrypted file or forget to re-encrypt after editing the file. Putting the cryptographic routines directly into the file system allows for transparent and easier to use systems.

Several existing systems can encrypt entire disk partitions. Encrypting the entire disk has the advantage of working with any file system and everything, including file metadata, is encrypted. Disk encryption can be convenient when the entire disk needs protection. However, these systems do not allow for fine control over files to encrypt, and, thus, the focus here will be on systems that encrypt at the file level.

If files are encrypted, users no longer need to worry a system administrator will either accidentally or not so accidentally read private files. Note

the administrator always has the ability to delete files as they control access to the physical disk. With file encryption, all the work in encrypting and decrypting is pushed to the clients. The server only needs to authenticate the clients. Often, not only file contents need protection but also the meta-information such as file names, file size, or last modification time. The problem with encrypting the file size or other information normally stored in the inodes is that system programs needing to work with the file system would no longer work without decrypting the inode information. For example, automatic backups of the disk would no longer be possible if the inode information were encrypted. Thus, meta-information is often not encrypted.

Some issues come up in the design of any cryptographic system involving both security and ease of use. Many of these areas will be discussed in later sections.

- *Privacy.* Unauthorized users should not be able to gain access to an unencrypted file.
- *Integrity.* Unauthorized tampering with files should be detected.
- *Unintrusive.* A key or passphrase must be obtained from the user at some point, but this should only be done once per session if possible.
- *Transparency.* Existing applications should transparently work with encrypted files.
- *Control.* Selective encryption of files should be possible to allow for public files.
- *Key Management.* Managing all the keys in the system should be simple for the user and administrators. In addition, if passwords or keys are lost, will recovering encrypted files be possible?
- *Sharing.* Sharing encrypted files among users should be possible and easy to do.
- *Performance.* Performance must not be degraded to the point where people will not wish to use the encryption features.

2.3 File System Issues

Some issues, common to most file systems, need addressing in the design of any new file system.

2.3.1 Sharing

Sharing files among users is important in any file system. Several ways exist to define the sharing semantics in a file system. UNIX semantics for file sharing dictates that users should immediately see the effects of all writes to a file. Thus if both $user_1$ and $user_2$ open the same file, if $user_1$ writes to the file and $user_2$ later reads from the file, $user_2$ should see what $user_1$ had written to the file.

Another method to implement file sharing is called session semantics. These semantics guarantee changes to files are seen by other users when the file is closed. If multiple users have opened the same file, they could hold different or stale copies of the files when one of them writes to the file. If an application needs stronger sharing semantics than these, something outside the file system is needed to enforce the necessary semantics.

Another approach is to treat every file as read-only and immutable. Every change to a file essentially creates a new immutable file with a new file name to refer to the new copy. Because all files are read-only, caching and consistency become much simpler to implement.

Studies of usage patterns on UNIX systems have generally shown that the amount of sharing is minimal and that reads are much more common than writes. Thus, write sharing and, therefore, conflicts occur infrequently. A file system could potentially take advantage of this low sharing pattern. On the other hand, a possible explanation for this low degree of sharing is that people would like to share more but are hindered by the difficulty of sharing files. In this case, a file system should aim to make sharing easier, increasing the frequency of sharing among users.

Access Control

The manner access control is done greatly determines how easy and thus how much file sharing between users will occur. Access control in a file system is making sure users only access file resources they are allowed to access, including read access to some files and full write access to other files.

UNIX access control mechanisms do not allow for easy sharing. In UNIX, file permissions can be set to any combination of read, write, and execute. Each file is associated with a single owner and a single group, and only the owner can set the permissions. Three groups of permissions exist: one for the owner, one for the group, and one for everyone else. Although the UNIX model has the concept of groups, groups are not commonly used for sharing and are difficult to maintain, requiring an administrator to create the groups and add users to groups.

2.3.2 Caching

Different media have different costs of access. Cache memory near processors are very fast, local disks are slower, and remote disks can be even slower to access. The basic idea of caching is to keep copies of recently accessed data on faster medium to speed up repeated access to the same data. One copy somewhere is usually considered the master copy and every other copy is a secondary copy.

Caching is needed for performance and also for practical implementation reasons. All types of file systems need some form of caching. Local file systems cache disk blocks in memory to reduce the time needed to fetch blocks from a disk. Distributed file systems need to cache remote blocks or files locally to reduce the amount of network traffic needed. Cryptographic file systems need to cache decrypted blocks to operate on them.

The size of the smallest unit of cached data can vary from a single byte to an entire file. The purpose of caching is to retrieve a larger block than initially needed and to hope for locality in the data and for future cache hits. The larger the size of a cache unit, the greater the chance of a future cache hit. On the other hand, the time required to retrieve a larger chunk is increased.

Consistency

Changes made to a cache copy must eventually be propagated back to the master copy, and a cache copy can become out of date if the master copy is changed by another client. The general problem of keeping the cache copy consistent with the master copy is called *cache consistency*. To maintain consistency, a system needs a caching policy to determine what data is cached and when data is removed from the cache.

The job of maintaining consistent caches can be given to either the clients or the server. The clients can be responsible for periodically checking the validity of their caches. This has the disadvantage of frequent checks that may not be necessary. Another approach is to have the server maintain information about what each client has cached. When the server detects a client has something invalid in its cache, the server contacts the client and tells the client to invalidate that particular file. One disadvantage of the server-centered approach is extra complexity in both the client and server code.

Write Policy

When a cache block is modified, the changes can be pushed back to disk or a remote server at different times, corresponding to different cache write policies. One method, called *write-through*, is to immediately send all writes through the cache directly to the master copy. Another method is *delayed-write*, postponing writes until a future time. Delayed writes have a higher performance than write-through policies, as writes do not have to wait for the extra time required to write through to the master copy. However, write-through policies allow for better recovery and reliability if the machine crashes or the cache is lost somehow. A type of delayed write, called *write-on-close*, writes out the cache when the file is closed. The cache policy used is related to the sharing semantics. UNIX file sharing semantics are much easier with a write-through cache policy, while write-on-close is suitable for session semantics.

2.3.3 Fault Tolerance

A system is fault tolerant if the system can work properly even if problems arise in parts of the system. File systems can be characterized as either stateful or stateless. The less implicit state contained in each request, the more fault tolerant a system can be. For example, if a server needs to keep track of all active clients, the server is maintaining extra state. If for some reason a client crashes, will problems arise when the server later relies on its out of date information? Not keeping the state information about active clients would allow for clients to go up and down and not affect the server at all, providing an extra cushion when something does go wrong.

Availability is how readily accessible files are despite possible problems such as servers becoming unavailable or communication problems. A fault-tolerant system should aim to provide high availability. One common way of providing fault tolerance and increasing availability is to replicate files either on multiple disks on the same machine or on multiple machines. Replication greatly complicates the system in maintaining consistency among the copies. Complete consistency sometimes may be sacrificed for performance.

2.3.4 Scalability

Once we leave local file systems and work with a network of machines, we start to care about the number of machines we can work with. Scalability is the ability for the system to handle a large number of active clients and servers and still be able to provide high performance for all involved. Centralized machines do not scale well as the number of clients and, consequently, the load increases indefinitely. The other extreme would be a fully distributed peer-to-peer network where every computer can talk with any other computer on a network. On the other hand, the cost of management and system administration is also important for large systems, and centralized system administration is usually much easier than distributed system administration.

2.4 Existing File Systems

In this section, several existing file systems will be described as a basis for comparison and motivation in the design and implementation of our own file system.

2.4.1 Local File Systems

FFS

The fast file system (FFS) [17] was an attempt to replace the original UNIX file system with a higher performance file system. A central problem with the traditional file system, according to the authors, was that data blocks for files become scattered across a disk, leading to a disk seek operation before every block access.

Several changes in FFS were made to improve performance. The block size was increased from 1024-byte blocks to 4096-byte blocks. Larger blocks allow for transferring more information on each data access, but also increases the amount of wasted space. To reduce the percentage of wasted space for small files, FFS fragmented a single block into smaller pieces. With this method, FFS had about the same amount of wasted space as the traditional file system and also saved space on larger files by needing fewer block pointers.

Another modification to improve performance was to place blocks normally accessed together near each other physically on the disk. As a common operation is to list a directory, the inodes for the files in a single directory were clustered near each other in a group of cylinders. In addition, all the blocks for a file were put close together to speed accesses to files.

Optimizing performance in a local file system will also come up when we move to distributed file systems and our own file system. Other than optimizing performance, other additions were made to the file system, features now common in most UNIX variants. For instance, a file locking mechanism to allow for synchronization among processes and symbolic links to allow links across file systems were two features implemented in FFS.

2.4.2 Distributed File Systems

NFS

The most widespread distributed file system in use is Sun's Network File System (NFS) [32]. NFS is extremely mature, commercially supported, and widely available.

A machine in NFS can both be a server and a client, as NFS does not have the notion of a dedicated server. The server maintains an exports list, in `/etc/exports`, listing the directories accessible to clients and, in addition, the names of the client machines permitted to access these exported directories. NFS does not have a global namespace. The client chooses a directory to make remote files available through the process of mounting. Because each client machine chooses the directories to mount, each machine can have a different view of the namespace. No clear semantics are defined for file sharing and consistency. A file created on one machine may not be visible on another machine for 30 seconds.

The NFS protocol [19] is based on Sun's Remote Procedure Call (RPC). The NFS protocol is as stateless as possible to simplify server implemen-

tations. One consequence is that the NFS protocol does not contain open and close operations. To encourage implementations of NFS on different operating systems, the protocol is built on top of the platform-independent External Data Representation (XDR). In addition, files are identified not by path names or inodes, but by file handles. The goal of the WebNFS [35] project is to replace the HTTP protocol on the Web with the NFS protocol, claiming that much speedup is possible due to the amount of tuning that has gone into the NFS protocol.

AFS

Another file system in use is AFS [10], originally developed at Carnegie Mellon as the Andrew file system, and now commercialized. The original goal of AFS was to improve the scalability of file servers. AFS presents a common namespace to all clients, and with the observation that server processing is usually the system's bottleneck, they pushed much of the work into the client.

In AFS, the dedicated server machines are collectively called *Vice*. A user-level process called *Venus* is used by the operating system on each client to handle system calls on AFS files. Venus caches files from Vice when the files are opened and stores them back when the files are closed. Venus stores the client cache on the local disk of the client, and assumes cached files are valid unless told otherwise. There exists a callback mechanism where the server notifies clients with cached copies when another user is trying to modify the same file. Partly due to this callback mechanism, the system is not stateless, and could have fault tolerance issues. For example, if a server goes down, what the client and what the server know can be out of sync.

Originally, the Andrew file system used whole file caching, although caching in chunks is now also used. Whole file caching implies Venus only needs to contact the server on file open or close. Reading and writing blocks from files are performed directly on the cache copy. This method of caching means changes to a file are not visible to other machines until a file is closed.

Security was a concern in developing AFS [24]. There exists an authentication server to issue *tokens* for users. Tokens establish the identity of the user to file servers. Servers and clients mutually authenticate each other and, after authentication, they can communicate with encrypted messages, although this encryption is usually not used. In AFS, the servers are completely trusted and are assumed to be physically secure. Another assumption

in their design is a secure communication channel exists between the Vice servers, for the administrative functions that need to occur among the servers.

To protect files, AFS contains access control lists at the granularity of directories. A mechanism exists for creating groups of users and assigning permissions to groups. Both positive and negative permissions are allowed, so one can specify that everyone in a certain group except some user has a set of privileges.

AFS puts related files into *volumes*. Using a mechanism similar to UNIX mount, AFS can join together different volumes. Files in AFS are identified by an identifier called a *fid* that includes the volume number, a vnode identifier, and a uniquifier. Fids are independent of the physical location of a file, so they are not invalidated when files move from server to server. To find the location of a volume, the client queries a volume location database replicated on each server. Entire volumes can be moved transparently among servers to balance the load. Read-only volumes can be replicated on multiple servers.

AFS has added the notion of a *cell* to correspond to an autonomous group. Cells allow for the job of system administration to be split, while at the same time allowing for one namespace. With the many cells in existence today, AFS has become a global file system, demonstrating the original goal of scalability has in many respects been achieved.

Coda

Another distributed file system is Coda [13], also developed at Carnegie Mellon. Coda was built on top of AFS, so much of the design from AFS can be seen in Coda. For example, the namespace is arranged in volumes, the user-level cache manager is called Venus, and callbacks are used as in AFS. What is new in Coda is the idea of *disconnected operation*. In AFS, and in most network file systems, when a file server goes down, the files on that server are no longer available. For those with laptops and those who cannot be continuously connected to the web, sharing files over the network can prove troublesome. While connected to the network, through the process of *hoarding*, Coda caches files that may be needed when the system later becomes disconnected. Users work transparently on remote files even when disconnected from the network. Once the network is reconnected, the local file changes are played back from a replay log, and changes are merged if possible. This process may still require human intervention because some changes result in conflicts that cannot be automatically merged. The Coda

project has shown caching can be used to support disconnected operation. Coda uses whole file caching both to simplify the implementation and to make the disconnected operation more transparent.

Sprite

Sprite [18] is a distributed file system developed at Berkeley, built on the assumption of diskless machines with large main memories. Sprite relies heavily on caching, with caching on both the servers and clients, and with caches stored in local memory. Caches are organized around fixed size blocks, with each block identified by a unique file identifier and a block number. Sprite uses a delayed write policy. Dirty blocks are written back to the server after 30 seconds. A unique feature of the Sprite cache is that it can dynamically change in size. The file system negotiates with the virtual memory system to allocate the physical memory between the two systems as the relative needs of the two systems change.

Sprite implements UNIX semantics for file sharing, in what the designers call concurrent write sharing. Files have a version number associated with them and this version number is incremented whenever the file is opened for writing. Servers are notified on all file opens and closes. When a client opens a file, the client compares the file's current version number with the version number for the file in the cache. If the version numbers are different, the client discards the information in its cache. To maintain cache consistency of writes, the server is able to detect a concurrent write sharing condition and disable caching on the clients, forcing all reads and writes to go through the server. As all operations can involve the server, this increases the load on servers and decreases the overall performance.

State is stored on the server and client side and this state is not stored on persistent storage. This may be an issue for fault tolerance. The kernel was also completely re-implemented for Sprite, so compatibility and portability may be an issue. However, the extra work apparently paid off as faster performance over both AFS and NFS was reported.

WebFS

The WebOS [30] project at UC Berkeley aimed to provide OS services for distributed applications. Part of WebOS was a global file system working on top of HTTP called WebFS [31]. Their system worked with their custom

WebFS server, but also could work in a limited way with standard HTTP servers. With their WebFS server, they enabled authenticated read/write access and cache consistency by extending the HTTP protocol. WebFS included access control lists, and users were identified by public keys. Another feature was its support for IP multicast, allowing for files and cache invalidation notices to be broadcast to interested clients. WebFS implemented the last writer wins cache consistency protocol.

xFS

Traditionally, distributed systems rely on central servers to provide all of the services in a file system. These central servers become a single point of failure, so many file systems have the notion of replication. However, another paradigm for file system design is a serverless network file system [2]. In this paradigm, all machines work as cooperating peers, and as any machine can take over if another machine goes down, this can provide high availability to file services.

The xFS system is an example of a serverless network file system. It is a log-based system, allowing for checkpoints and the ability to rollback. xFS eliminates central server caching and uses client memory as a large and cooperative cache. A problem is that the xFS protocol depends on a level of trust among machines unreasonable on a global scale.

UFO

UFO [1] provides access to a read-only HTTP name space, while providing read/write access through the FTP protocol. UFO is implemented entirely at the user-level by intercepting system calls in the same way a debugger would work. Consequently, UFO does not need any kernel code or kernel recompilation, providing the advantage of not requiring superuser privileges to install or run. A disadvantage, however, of being implemented completely at user-level is limited control and restricted data access, being only able to trap system calls and not having access to internal kernel structures.

OceanStore

OceanStore [14], a part of the Endeavor project at Berkeley, aims to create a utility infrastructure, providing secure and highly available access to persistent objects. A prototype system is being developed, although many

components have yet to be implemented. A distributed file system could be built on top of the completed OceanStore infrastructure. Although there is a separate application programming interface (API) applications can use to take full advantage of OceanStore's capabilities, a translation from a UNIX file system API to the OceanStore API will be provided.

Unlike related systems, OceanStore has a model of an untrusted infrastructure, relying on redundancy and providing cryptographic security to protect data. In OceanStore, by separating information from location, and allowing information to freely migrate, data should be able to survive major disasters or regional problems. The designers of OceanStore call their caching policy *promiscuous caching* as data can be cached anywhere. Objects can be replicated to any server around the world. This provides high availability and locality of data, but consistency in such a system becomes a difficult problem.

Locating objects on such a decentralized system is addressed also. All objects are tagged with a globally unique identifier. To locate an object, OceanStore uses a routing protocol to find an object on the network based on an identifier. Each update to an object effectively creates a new version of the object. A conflict resolution model allows for write sharing. But as the servers are untrusted, the servers need to resolve the conflicts while working with encrypted data. They discuss how to do operations such as search, insert, and delete on encrypted blocks. Directories map human readable file names to the global identifiers. The use of self-certifying path names [16] ensures file names are resistant to attacks. There is no single global root for the file system as a user of the system can choose several directories as roots.

OceanStore has the idea of introspection. A program can monitor the system, figure out how to optimize it, and then carry out the optimization. For example, one should be able to detect different usages on different machines during different times of the day, and compensate the load across those machines accordingly. By a continuous process of observing and optimizing the system, the system will be self-tuning and should achieve higher performance and reliability.

2.4.3 Cryptographic File Systems

CFS

The first system to allow transparent access to encrypted files is the Cryptographic File System (CFS) [5]. CFS can work on top of any other file system to put file encryption services into the underlying file system. By moving the cryptographic tools into the file system, CFS allows users to work with encrypted files easily. CFS is implemented using a NFS loopback server, and with no kernel code, CFS is very portable. On the other hand, a kernel implementation can be more efficient.

Every directory has a cryptographic key used to encrypt all file names and files inside the directory. From users' point of view, before they can use an encrypted directory, they need to *attach* the directory and enter in a password. After attaching the directory, the contents become visible under the CFS mount point. If a user has multiple unrelated directories they wish to attach, it may be necessary to remember several different passwords and remember to attach each directory separately.

Access permissions for all files in a directory are specified as a group. CFS protects not only the file data, but also encrypts file names and symbolic links. Information in inodes, such as file sizes, are not encrypted, allowing utilities that work with the file system such as `fsck` or automatic backup systems to continue to work without modification. CFS provides no easy way to share data between users, as the users will need to find a way to share the keys. No protection is given against unauthorized tampering with files.

TCFS

The Transparent Cryptographic File System (TCFS) [29] aims to improve on CFS by improving transparency. Whereas in CFS, the user needs to attach directories manually and enter in passwords, in TCFS no password, other than the login password, is required. Another change from CFS is TCFS moves all encryption into the kernel. With all code in the kernel, extra context switches are avoided, leading to higher performance. TCFS adds an encryption bit to the standard UNIX file attributes, and when the encryption bit is set, TCFS encrypts the file.

TCFS also implements threshold group sharing of files. Only when a certain threshold number of users in a group are logged in simultaneously on a machine can files be opened. This is implemented by giving each user in

the group a part of the group key, and guaranteeing if and only if enough parts are available, the group key can be reconstructed from the parts.

ECFS

The Extended Cryptographic File System (ECFS) [4] builds on top of CFS with several additional features. Whereas CFS provides for privacy, ECFS ensures data integrity using keyed hashes. Each block in a file is signed using the block's index in the file as a salt for the signature algorithm. The name of the file is used in verifying the integrity of the file to prevent a substitution attack, where the server substitutes a different file than the one requested by the user. ECFS also allows for the selective encryption of files. This allows for public files within an encrypted directory, whereas in CFS, all files in a directory had to be encrypted. File names can be encrypted selectively, but to avoid name conflicts, ECFS requires any given directory must have all encrypted or all unencrypted file names.

Cryptfs

Cryptfs [36] does encryption at the virtual inode (vnode) layer, can work on top of any other file system, and is implemented in kernel code. Files are encrypted by blocks so the entire file is not decrypted when changing one byte. File names and symbolic links are also encrypted. One initialization vector is used per mount, thus similar plaintext files encrypted with the same key will produce similar ciphertexts. Only one active key is associated with a session at a time, so the user is required to switch the effective key manually to access files encrypted with different keys.

SFS

Several different file systems have the name Secure File System (SFS). We will discuss the SFS system done between the University of Minnesota and StorageTek [12]. SFS provides encryption for users accessing files using normal networking protocols such as HTTP and FTP. SFS builds on top of the UFO file system. Files are decrypted on file open and encrypted again when the file is closed and put back out to the server. As the read and write system calls are not caught, partial encryption or decryption of files is not supported. SFS uses smart cards for authentication and key management. The processor on a smart card is used to perform key computations, such as

decrypting file keys. In the SFS architecture, a trusted group server enforces the access control lists and provides for an audit trail.

CSFS

The Cryptographic Storage File System [9], also known as Cepheus, provides for confidentiality, integrity, and availability of data. As in CFS, file data and file names are encrypted, but information within inodes is not encrypted. To allow for random access in files, each file block is encrypted separately using different initialization vectors (IV) so same plaintext blocks will not encrypt to the same ciphertext. In addition, cryptographic hashes are provided for integrity.

CSFS promotes group sharing. Most other cryptographic file systems leave key management up to the user and sharing involves disclosing a password to another user. CSFS uses a separate trusted group server to maintain group information and generates a key for each group. When someone leaves a group, the group key must be changed, meaning all files encrypted using that group key must be re-encrypted. To re-encrypt all files immediately would cause delays, so CSFS uses a delayed re-encryption method. Files are not re-encrypted until someone makes a change to the file.

CSFS stores extra information in the inodes, such as encryption keys. Because they made modifications to the standard file structures, they needed to write their own fsck program to handle crash recovery. The entire CSFS partition is stored as a virtual disk, that is as a file on the server's disk. This means accessing files on a CSFS partition outside a CSFS client system is not possible. CSFS uses a loopback NFS server on the client machine, avoiding the need for writing any kernel code. The file server implements many of the RPC calls available in NFS, but also adds an interface for directly accessing the raw disk.

CSFS uses a delayed write strategy for the cache write policy. Blocks are written to the file server when the block is ejected by the replacement policy. A write-on-close policy would have been more desirable, but the other policy was chosen for implementation reasons. Because the NFS loopback server used by CSFS is stateless and cannot tell if a file is open or closed, detecting a file close operation is not possible. Authentication is entrusted to the RPC library.

Chapter 3

Design

The primary goal in designing Kiwi was security, followed by the goals of simplicity and scalability. This chapter will discuss the protocols chosen for the Kiwi file system and how we addressed these issues of security, simplicity, and scalability.

3.1 Protocols

From the beginning, we wanted to try to build a file system on top of the HTTP protocol, as we did not wish to invent a new protocol of our own. We secured the HTTP protocol with SSL and extended it with WebDAV to allow for all file system operations. We will now look in more detail at the various protocols chosen for the Kiwi system.

3.1.1 HTTP

The hypertext transfer protocol (HTTP) has been widely deployed with the spread of the World Wide Web. HTTP [11] is a very extensible protocol and was chosen partly for this ability to extend it into a complete file system protocol. The protocol consists of a text header followed by binary data.

This is an example of a HTTP request:

```
GET /path/to/file HTTP/1.1
Host: www.somewhere.com
Some-Random-Header: Some-Random-Value
Content-Length: 0
```

<empty line>

The first line, called the request line, specifies the HTTP *request method*, followed by the resource being accessed, in the form of a Uniform Resource Identifier (URI), and ends with the HTTP protocol version. HTTP defines several basic HTTP methods including GET, HEAD, POST, PUT and DELETE. GET is what is used above and is what a web browser usually uses to download web pages. HEAD tells the server not to actually return the body for the response but only to return the headers. POST is used to send arbitrary data such as often found in HTML forms. PUT and DELETE allow for uploading or deleting a resource on the server and are rarely used, because most web servers only deal with read-only web pages.

After the first request line, any number of HTTP headers may follow in the format of a header name and value separated by a colon. Several predefined headers, such as Content-Length, specify the size of the data to follow the header. The headers are ended with a blank line, followed by a message body, and in the above case, no body is needed.

The HTTP response to a request uses a similar format as above. Here is a possible HTTP response to the above request for a file:

```
HTTP/1.1 200 OK
Date: Sun, 01 Apr 2001 12:34:56 GMT
Server: Apache/1.3.19 (Unix)
Last-Modified: Thu, 01 Mar 2001 01:23:45 GMT
ETag: "981cd-ce5-3acc0e14"
Content-Type: text/html
Content-Length: 9999
<empty line>
<9999 bytes of data>
```

The first line now specifies the result of the request. The rest specifies HTTP headers similar to the request. In this response, we see a non-zero Content-Length to tell the client that data follows the headers.

To perform the operations needed in a file system, such as retrieving file information and creating directories, we needed to extend the basic HTTP protocol. The two primary ways to extend HTTP are by adding new HTTP methods and by adding new HTTP headers. Unrecognized methods and headers are ignored by HTTP servers and clients, so extending the protocol

should not break older software. Initially, we extended the HTTP protocol ourselves with new methods and headers, but this approach was abandoned after we found WebDAV.

3.1.2 WebDAV

Web Distributed Authoring and Versioning (WebDAV) extends the HTTP protocol to allow for more operations on files by adding new methods and headers [33]. WebDAV is an open standard and we can take advantage of existing code. WebDAV defines several new methods and implements the PUT and DELETE methods originally defined in HTTP. The new methods added by WebDAV are:

MKCOL	COPY	MOVE	
LOCK	UNLOCK	PROPFIND	PROPPATCH

WebDAV has the notion of a collection that is essentially a directory. MKCOL allows for creating new collections or directories. COPY and MOVE perform their respective operations on files. LOCK and UNLOCK allow for a basic locking mechanism on files to facilitate sharing of files. Finally, PROPFIND allows for finding file attributes and file properties and PROPPATCH allows for setting file properties.

In WebDAV, properties are essentially name and value pairs. WebDAV has a flexible mechanism for dealing with properties using PROPFIND and PROPPATCH. *Live properties* are properties the server knows about and maintains. An example of a live property is whether a file is executable or not. Arbitrary properties are also allowed and these are called *dead properties*. For example, a client could set a property with the creator's contact information, if someone wanted to contact whoever created the file. The server stores dead properties as is without processing. The ability to set arbitrary properties can be used to store caching information or other meta-information to be stored along with a file.

3.1.3 SSL

The secure sockets layer (SSL) [28] was designed by Netscape and is also an open standard. Adding security to HTTP by using HTTP layered over SSL (HTTPS) was an easy choice, as this is the standard protocol used by sites

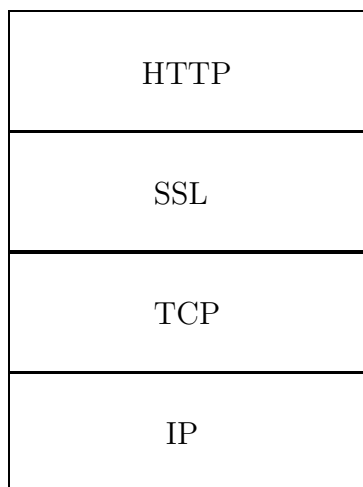


Figure 3.1: Secure Sockets Layer (SSL)

on the Web needing secure transactions. Note HTTPS is different from the Secure HTTP (S-HTTP) protocol [26] that adds security extensions to the HTTP protocol.

SSL works at a level above TCP/IP but below the application layer. Applications can continue to use HTTP or any other protocol unmodified, while the underlying SSL implementation will secure the socket and thus the connection. Thus, all the benefits in choosing HTTP still exist by going with HTTPS.

A key part of the SSL protocol is the handshake used to set up the connection. The handshake is used to determine the cipher for encryption, certificates are exchanged and the server and client can authenticate each other, and then a key exchange is performed using public key methods. Once the SSL handshake is successfully completed, the two communicating parties have verified that the other party is who they say they are and that they share a common secret key. Using this common secret, all further network traffic is encrypted using a symmetric cipher.

With e-commerce, credit card numbers, financial transactions, and other sensitive information going over the Internet using the HTTPS protocol, one would expect the underlying security protocol, SSL, to be secure. Especially due to the open nature of the protocol and the many people who have thought about the protocol, SSL is likely one of the most secure protocols available.

Another benefit of using SSL is the availability of free implementations of SSL, helping shorten the development time, and relieving us of developing a protocol from scratch.

3.2 Security

Unlike other cryptographic file systems designed to put cryptographic services into existing file systems, Kiwi was designed as a standalone cryptographic file system with strong security features. Security in a file system can be easy if the file system limits access to files. At the extreme, a file system could disallow access to files for anyone not sitting in front of one particular machine at a particular time of the day, and without passing a retinal scan, fingerprint scan, and maybe a voice signature for extra protection. Thus, security is not enough in a file system. Accessibility to files is also necessary. Secure access to files from anywhere and everywhere was a primary goal in developing Kiwi.

Different security problems can be addressed. There is the problem of unauthorized access to information. By using SSL, we eliminate much of this problem of attackers listening in on the network. There is also the issue of access control on specific files. Users should only be able to see files they have permission to access. Another security concern is denial of service. This issue was not addressed at all, partly due to its complexity. There is not much Kiwi can do if an attacker floods the network with bogus packets. However, the current solutions to prevent denial of service in network systems should also apply to this system.

3.2.1 Authentication

As in any file system, servers need to authenticate users. We chose to use certificates as certificates are what SSL inherently supports. Each server is set up to trust at least one certificate authority (CA). In simple systems, the CA could very well be located on the same machine as the server. When users obtain accounts on a machine, instead of user names and passwords, they would instead receive certificates and private keys. The certificate would contain the user's public key and their user name on the server's machine. Of course, a certificate authority needs to cryptographically sign a certificate for it to be valid.

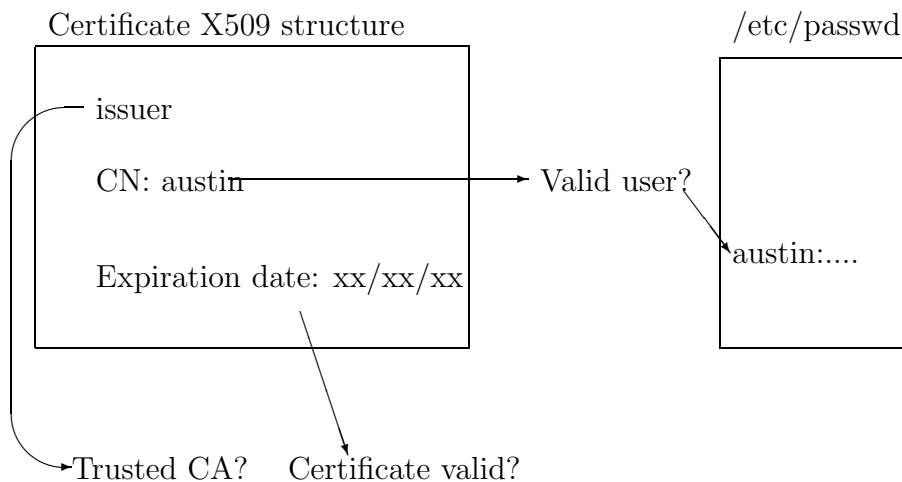


Figure 3.2: Authenticating users with certificates

The process of authentication on the server side should proceed as follows. The server first checks the certificate is valid and signed by a trusted CA. Then the server looks at the “Common Name (CN)” field of the certificate for the name of the user, e.g., *austin*. The server looks up that name in the local password file and makes sure the user has permission to access files on the system. From then on, the server should treat the user as authenticated as the user specified in the certificate.

3.2.2 Access Control

Once users are authenticated, a mechanism must exist to control the access to resources. UNIX access control is too difficult to use and does not promote sharing. We desired a better system for access control, preferring a system similar to AFS with access control lists, allowing normal users to create groups and set permissions on files. An access control system such as in AFS would promote greater sharing and ease of use.

3.2.3 File Encryption

SSL provides session encryption by protecting against people listening in on the network. The files, however, are still seen in unencrypted form by the

file server. In some situations, one may not want the system administrator of the file server to be able to read one's files. A user may also be concerned about someone breaking into the file server and gaining access to all of the files there. File encryption prevents the leak of information to someone who has access to the file server.

Several significant differences exist between file encryption and session encryption. Files are persistent, so long term security must be taken into account. Encrypting a file such that no one can break into the file now is not enough; it must not be possible, or the risk should be sufficiently low, that in the future the file will also remain safe. Another difference is that files can require random access, whereas most encryption techniques simply cannot work with anything other than a sequential stream of data.

Kiwi supports both file encryption and session encryption. Many other cryptographic file systems simply encrypt all files, making it unnecessary to also encrypt the network traffic. But, in the Kiwi system, not all files need to be encrypted. Not encrypting the network traffic would make unencrypted files vulnerable to network snooping. Having both forms of encryption implies that encrypted files are encrypted a second time when being sent over the network with SSL. These two forms of encryption work together to give a high level of security and cryptographic protection.

3.2.4 Trust Model

We assume the user completely trusts the local machine. Not trusting the local machine would face difficult problems as the local machine can observe everything the user accesses. Trusting the local machine implies the user trusts everyone who has superuser access on the local machine. In addition, the user believes the file server is reliable in storing files but the file server may not be trusted to not reveal the contents of files. The network is completely untrusted by the user. The file server trusts the certificate authority to only generate certificates for authenticated users. Note the certificate authority could be running on the same machine as the file server.

3.2.5 Firewalls

Having HTTPS as the network protocol has an important consequence related to firewalls. Companies or other organizations usually protect their internal computers from the Internet by setting up a firewall between the

company's computers and the rest of the Internet. The firewall filters out unwanted packets on all but selected ports. The HTTP and HTTPS ports are usually opened on firewalls to allow for those inside the firewall to browse the Web. Therefore, for most firewall setups, the Kiwi network traffic will pass right through the firewall along with the other HTTPS traffic. Allowing file access from behind existing firewalls greatly reduces the administrative hassle of managing the firewall. If another protocol and port were used, another port would need opening on the firewall, and many companies have a policy against opening up extra ports. Thus, Kiwi is one of the few file systems usable even if the client and file server are separated by a firewall. This access to files, even if one is behind a firewall, is another step towards secure access to files everywhere.

3.3 Simplicity

We wanted it to be simple to use and maintain the Kiwi system. For example, the AFS developers themselves said they found their prototype system difficult to operate and maintain. Other file systems like NFS are no easier to maintain or use, especially as the number of machines increases. In Kiwi, one of the goals was to provide users with a familiar environment and to reduce what they had to learn to use the system.

Another major goal was to simplify the development and implementation of the system. Because the time available for development was very short, we wanted a functional system as soon as possible. We tried to use existing code whenever possible, and choosing standard protocols allowed us to take advantage of the large amount of existing code. With freely available implementations of all the protocols used, we could focus on building the file system itself, dramatically shortening the amount of time needed for coding.

3.4 Scalability

Another design concern was scalability. We can view scalability in several different ways. Traditionally, people have looked at how many clients file servers can handle simultaneously. We are more concerned about handling a large number of servers and clients rather than the individual load one server can handle, because we are looking for a file system to work on a global level.

HTTP has proven itself as a scalable protocol with the growth of the Web. The number of transactions done over the Web continues to increase with the number of pages and information available also increasing at a fast rate.

In several file systems, server processing power was found as the bottleneck. For example, in AFS, substantial performance gains were achieved by pushing as much of the work into the clients. Increased performance and lower server load implies higher scalability as more concurrent clients can connect at a time. In Kiwi, the same type of approach was taken to push the work into the clients. We wanted to modify the existing server as little as possible while still maintaining reasonable usability. Thus the server was made as simple as possible, and the clients were made smart enough to do most of the work.

3.5 Summary

The major design choices made for Kiwi include:

- Choosing standard protocols where possible
- Using HTTP as a simple, extensible protocol
- Extending HTTP with WebDAV
- Securing network connections with SSL
- Using certificates for authentication
- Allowing for transparent file encryption
- Reusing code to simplify implementation
- Providing for scalability and global access to files

The next chapter will discuss how we turned this design into a working implementation.

Chapter 4

Implementation

Due to the standard protocols chosen, many of the steps involved in implementing the Kiwi server and client were relatively easy. This chapter will discuss implementation issues and discuss how the file system can be used on a practical basis.

4.1 Kiwi Server

For a file server, we decided to build on top of a standard web server, Apache. Apache is well supported and maintained, and, as a result, we found the Kiwi file server extremely easy to implement, needing only a couple hundred lines of extra code.

4.1.1 Apache

The Apache web server [3] was ideal for several reasons. First, the source is freely available and its design and architecture allows for the easy creation of modules to extend the functionality of the basic Apache server. Secondly, since the Apache group was formed in 1995, dozens of people have put in significant effort into making Apache stable. Furthermore, Apache has proven itself with about 60% of all existing web servers running some form of the Apache server.

Because Apache was not originally designed for what we needed to do, we needed to modify configuration files and patch Apache in a few places. Apache was configured to share the entire file system. Unlike a normal web

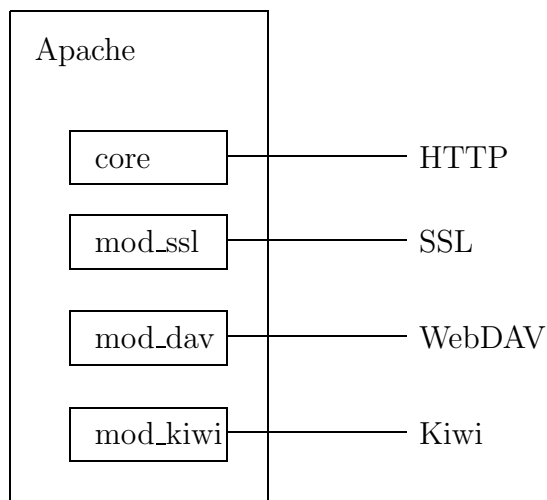


Figure 4.1: Apache modules

server, we wanted to make everything on the system available, for example, even system files located under `/etc`. To share the entire file structure, we needed to run Apache as the superuser.

4.1.2 Additional Modules

In addition to using Apache, we also used the `mod_ssl` module for Apache that provides an interface to the OpenSSL library and `mod_dav` to incorporate WebDAV support into Apache. We then created an additional module called `mod_kiwi`. Therefore, the entire Kiwi server consisted of Apache, `mod_ssl`, `mod_dav`, and `mod_kiwi`, with most of this code already existing and being maintained by others. Using these existing components as the foundation for a file server meets our goal of extreme simplicity in development.

`mod_kiwi`

The primary purpose of `mod_kiwi` was to perform authentication. As we mentioned, the Apache process is run as the superuser, yet we cannot allow everyone access to all files on the system. Our module checks the name in the certificate presented by the client and authenticates access to files based on that name. Enforcing access checks on files is done by making a `seteuid`

call to effectively change the permissions of the Apache thread. Then all further requests from the same client will be performed by that server thread with the permissions of the user. Thus if the user attempts to access a file without sufficient permissions, the server thread will be blocked by the local file system for not having enough permissions. This simple method allowed us not to worry about race conditions nor to handle the access checks ourselves.

However, Apache and the other modules were not designed with the idea its effective permissions would be changing all the time. Some files, as in log or lock files, are opened and written by all threads, but each thread could be running with the permissions of a different user. Solving this required adding additional `seteuid` calls in several places in `mod_ssl` and `mod_dav` to temporarily switch back to the superuser.

4.1.3 Dynamic Generation of Certificates

We want to allow for mobility and access to files anywhere, but carrying around a personal certificate to use for authentication is not practical in today's environment. In the future, smart cards, or other similar devices, may allow for carrying around certificates wherever one goes. For now, most people are comfortable carrying around some password to access system resources. Kiwi allows using a password to access a dynamically generated certificate. These certificates are generated using a Netscape browser and generally have a short life time.

Certificates can be added to the client certificate cache in the Netscape browser. During the SSL handshake phase, the server requests the client to authenticate itself, and if the browser presents a valid certificate, the server allows access to files on the server. However, when a valid certificate is not available, the server redirects the browser to another page. Normally there would be a page saying the client is forbidden to access the page. In our case, the browser is redirected to a server script protected with HTTP Basic Authentication, causing the browser to pop up a dialog asking for a user name and password. The password is still sent over SSL, making it immune to network snooping.

After the user is authenticated, the user is shown an HTML form that asks for information to go into generating a certificate, such as the life time for the certificate and the user's email. The important part of the form is a HTML tag Netscape invented called `KEYGEN`. When the user submits a form with the `KEYGEN` tag, the Netscape browser automatically generates

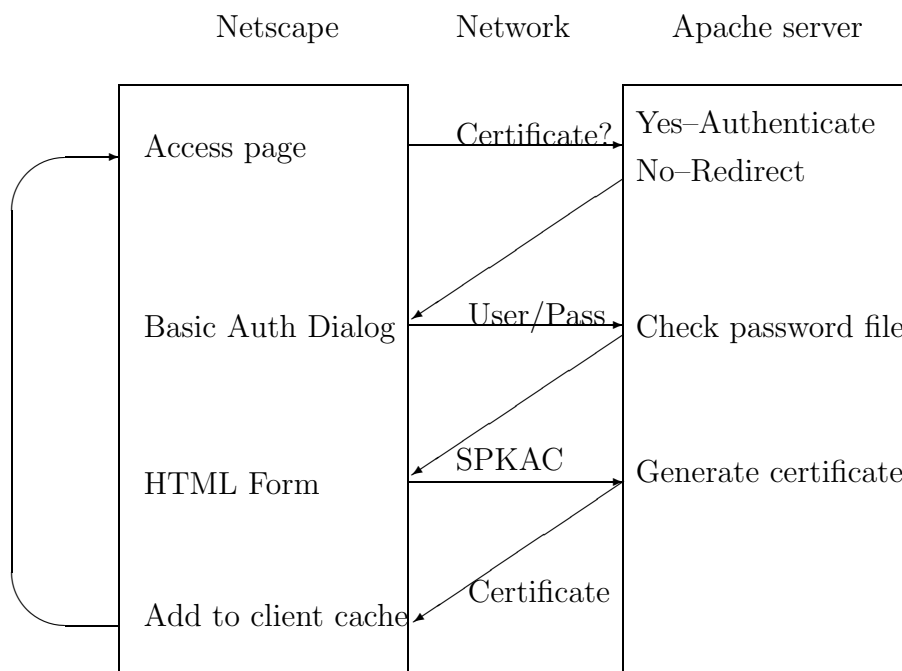


Figure 4.2: Generating certificates with Netscape browser

a private and public key pair. The Netscape browser keeps the private key in its cache, signs the public key, and sends the public key to the server in a structure called a Signed Public Key And Challenge (SPKAC).

When the server receives the SPKAC, the `kiwi sign` command, described on page 50, is used to generate a new certificate. The server passes the newly generated certificate back to the client. By passing back the certificate with the appropriate MIME type, the browser automatically notices the certificate, finds the matching private key in its cache, and adds both the private key and the certificate into its client certificate cache. Now if a user tries to access the server again, as there is now a certificate, access should be granted for the life time of the certificate.

If the user is using a browser on a public computer, then having a very short life time for the certificate is important in preventing other users of the computer from being able to access private files. Unfortunately, there is no easy way to tell the Netscape browser to automatically remove certificates. Users need to manually remove their certificate from the web browser once they have finished using the computer.

This method of generating certificates is not limited to when one needs to generate a certificate quickly to access a couple files. Even using a file system interface, a user needs access to valid certificates. For the sake of simplicity, we used this same method of generating certificates with a browser, except certificates can now be allowed to have a longer life time. Once the certificate is downloaded, the user exports the certificate from the browser, and then can pass the exported file to the `kiwi add` command, described on page 48.

Another issue is where the password for authenticating users comes from. One possibility is to use the local password file, `/etc/passwd`, already on the system. Thus, users can use the same password both to log in to the machine in the standard ways and also for generating certificates. Alternatively, the administrator can set up a separate password file with passwords generated specifically for Kiwi.

4.1.4 File Access With Web Browser

With the server set up as above, we immediately gain all the benefits of having a web server and its functionality. Using any web browser, users can securely authenticate, download a certificate, and have secure access to files. File upload capabilities are possible through CGI scripts. Thus, with only a browser, one can download and upload files securely using HTTPS. So

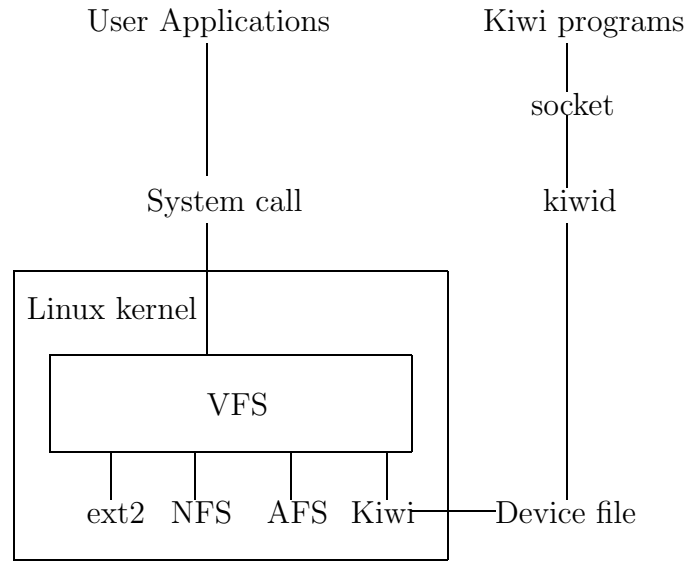


Figure 4.3: Linux File System

without much work, we have a simple web-based interface for our file system. This partially fulfills our goals of secure file access from anywhere, or at least anywhere there is a web browser and a network connection.

4.2 File System Client

Although accessing files from a browser is a useful feature if one is traveling or otherwise not able to access one's computer, for daily use, a browser interface is much too cumbersome and limited. To attain the full functionality of a file system, we developed a Linux file system, allowing the mounting of Kiwi on a directory like `/kiwi`. The file system can be divided into three parts: the Linux kernel module, the user-level program, `kiwid`, to actually handle requests, and the programs users use to interact with the Kiwi system.

4.2.1 Kiwi protocol

A protocol was needed for communication between the kernel and `kiwid` and between user programs and `kiwid`. The protocol developed was fairly simple. One side issues a request and the other side carries out the request and

returns a reply. The request and reply both follow the same format. The request, or reply, first consists of a header. The header includes, among other things, the type of the request and the number of parts in the request. Each request header is then followed by a number of parts, where each part also consists of a header followed by the part data. Much of the common code between the user kiwi programs and kiwid, including the functionality to communicate using the Kiwi protocol, was put into a library, *libkiwi*.

4.2.2 Usage

Using the client is fairly simple. The kernel module is installed under Linux using the `insmod` command. Once the Kiwi kernel module has been installed, Kiwi can be mounted on some directory such as `/kiwi`. All Kiwi files under here use a naming scheme very similar to how pages are named on the Web. To refer to the file `/path/to/file` on server `machine`, one would access the file `/kiwi/machine/path/to/file`. This is essentially a URI-based namespace and is similar to other systems like WebFS. With the user programs, users authenticate themselves to Kiwi, and can then access the files under `/kiwi`. Kiwid is automatically invoked as necessary and runs in the background to handle requests from the user.

4.3 Kernel Module

The kernel code for the file system was implemented on Linux because of the available documentation, the great support, and, most importantly, the free source code. In implementing Kiwi, we could look at and learn from the implementations of other network file systems on Linux. The Linux kernel also allows for dynamic loading and unloading of kernel modules, easing the development of the file system. Unlike Cryptfs with all of the code in the kernel, as much as possible of the Kiwi code was put into the user-level daemon, kiwid, instead of the kernel. This was to allow for easier and safer development, as each line of kernel code has the possibility of crashing the machine, whereas the worst a user program such as kiwid can do is crash itself.

4.3.1 VFS

Linux has a virtual file system layer (VFS) to make programming a file system easier by combining many of the common file system tasks into the common VFS layer. Most of the time spent in developing the Linux kernel module was put into understanding how the Linux VFS layer worked. The VFS layer provides a common interface for all file systems, whether they are local or remote or have other special characteristics. This layer uses virtual inodes (vnodes) to identify files, where a vnode is a unique number across file systems to represent a file. Vnodes correspond to inodes except inode numbers are not unique across file systems.

When applications make system calls on files, they pass to the kernel the name of a file. These system calls are passed to the VFS layer and the VFS layer determines the file system being accessed by that filename. Once the file system is determined, VFS calls the appropriate routine to handle the call. For instance, the VFS passes system calls made on files under `/kiwi`, or wherever else Kiwi was mounted, to the Kiwi kernel module to handle the request. The Kiwi kernel module in turn passes the request to `kiwid` to actually handle the request. The response coming back from `kiwid` goes through the Kiwi kernel module, back up to the VFS layer, and out to the application.

4.3.2 Reading Directories

Most of the system calls were very easy to handle, simply being passed to `kiwid`. However, the `readdir` system call was more difficult to implement, requiring the kernel to maintain state. Applications repeatedly call `readdir` to return entries within a directory, until there are no more entries to return. Each successive call to `readdir` needs to return the directory entry after the last entry returned in the previous call, and so, the kernel must keep track of the position in the directory. The initial implementation had the kernel caching directory entries in kernel memory between `readdir` calls, but, partially due to the desire to move code out of the kernel, the current implementation uses a file on the local file system as a cache of directory entries. The directory entries are written by `kiwid` into the file, and the kernel reads from the file, keeping track of the file position between calls.

One special directory is the root of the Kiwi tree. Obviously, when we list the root, we cannot and do not want to list all the servers in the world.

Initially, doing a listing on the root Kiwi directory will return nothing. When a server is referenced, that server name is cached, and a directory is created for the server under the root directory. Thus, `ls /kiwi` will give the user a list of all servers previously accessed by that user. Currently, the list of servers is persistent and will continue to grow indefinitely as more and more servers are accessed. In the future, removing older entries to reduce the length of the list may become necessary.

4.3.3 Requests

The Kiwi kernel module communicates with user processes (kiwid) through a device file. Currently, all requests going through the device file must be initiated by the kernel. User processes can send requests into the kernel module using `ioctl` calls. The kernel separates requests by user, as there is no single kiwid process handling requests for all users on a machine. A fixed maximum number of concurrent requests is allowed for each user and is currently set at 10.

Because requests have multiple parts, a kiwid process needs to call `read` multiple times on the device file to completely read a single request. The module guarantees that successive reads will return successive parts of a single request. To prevent concurrent reads by different processes conflicting with each other, the module needs to keep requests separate. The request being handled, at any point, is stored in the internal file structure that is unique per open of the device file. This request stored internally is then used for successive reads and writes until the request is completed. Although this goes against the traditional semantics of reading and writing, we need no locking or other synchronization to guarantee that a full request is handled by a single process. Multi-threaded processes accessing the device file concurrently need to either perform locking themselves or open up the device file multiple times, allowing the kernel to manage the requests.

4.4 Kiwid

Following on our goals of ease of development, we tried to reuse as much existing code as possible when implementing the user-level daemon, kiwid. We used the freely available OpenSSL [20] library for all of the cryptographic functionality we needed, and the neon library [21] to handle both the HTTP

and WebDAV protocol over SSL connections.

Kiwid has several main functions, shown below:

- Read requests from the kernel and from the user
- Communicate with Apache server
- Maintain user credentials
- Manage the local cache
- Encrypt and decrypt files
- Perform path name translations

Its primary job is to carry out the requests by talking to the Apache file server. Kiwid listens for requests from the kernel through the local device file and also listens for requests from user programs from a local domain socket. Requests received from either the device file or local socket are handled in exactly the same way. Kiwid is multi-threaded, creating new threads as necessary to handle the incoming requests.

Unlike many other daemons that run as the superuser on a machine, one kiwid process is run per user, running with the permissions of that user, because we found no need for kiwid to run privileged to perform the above tasks. With the bulk of the code in a user-level and unprivileged process, security is much less of an issue within kiwid itself. The only part needing to be made secure is the interface between the kernel and kiwid. This is easier as the size and the complexity of the kernel code is much easier to handle than the large amount of code in kiwid. Also, theoretically, users can have their own customized versions of kiwid to do what they want it to do.

Several problems arise with having one kiwid running per user. The main issue is the difficulty of sharing between users on the same machine because no communication is possible among kiwid processes for different users. For example, if one user accesses a common program like emacs, and another user also accesses emacs, they will each download their own copy of the file. Another possible issue is the use of resources. With a single kiwid process, more efficient sharing of processor resources can exist among users. With the current design, if users are not accessing any Kiwi files, their kiwid processes are also idle, taking up memory and some processor time. These concerns were part of the reason the initial implementation of kiwid used a single

privileged daemon. However, we changed the implementation to have one kiwid process per user, finding multiple kiwid processes simpler to use.

A benefit of having a per-user kiwid process is that the only part of the system requiring privileged access to the machine is the installation of the kernel module. Even without the kernel module, kiwid can still perform requests received from the local socket. Users can still manually perform file requests, such as retrieving and uploading files, without the kernel module. There may be systems where a user does not have superuser privileges and wants access to Kiwi files. A user could have a ftp-like or shell-like interface to the Kiwi namespace for accessing files. Also, as kiwid should be mostly portable, one can run the client portion of Kiwi on platforms other than Linux that do not currently have a kernel file system module. For instance, kiwid is known to work on Solaris, allowing access to Kiwi files on these systems. By increasing portability, we allow for more universal and secure access to files.

4.4.1 File Encryption

File encryption is performed similar to other cryptographic file systems. Encryption and decryption is done transparently by kiwid on the client side. When an encrypted file is read, kiwid decrypts the file and stores the decrypted file in the client cache. All file operations by applications work on the decrypted file, so they all work as they normally would. When the file is put back to the server, kiwid re-encrypts the file, and sends the encrypted file over to the server. Thus the server never has access to the decrypted file.

Users have full control over encrypting specific files and not others. The files to encrypt are specified in a configuration file, with each line consisting of a regular expression matching files to encrypt. Thus one can easily encrypt all files under a directory or all file names ending with a certain string. This is in contrast to other systems like CFS with the granularity of a directory, where all files in a directory are encrypted or not. Having the granularity of a directory can create problems. For example, one may want to encrypt all files in one's home directory but still have some public files, such as a .forward file. This is one of the problems ECFS addresses.

Currently, the file encryption key and initialization vector (IV) is stored at the beginning of the file itself. A hash of the file's contents is appended at the end to detect tampering with the file. Kiwi uses standard cryptographic algorithms with SHA-1 for its message digest algorithm and Blowfish for file

encryption. Kiwi uses the Cipher Block Chaining (CBC) mode of Blowfish allowing for the encryption of byte sequences of arbitrary length. The algorithms themselves can be changed easily to any algorithm OpenSSL supports.

In many situations, one can learn a great deal of useful information from the metadata associated with a file, such as its name and size. For file names, the obvious solution, and what many other cryptographic systems do, is to encrypt the file names. Kiwi can also encrypt file names and then, makes it into a legal filename by doing base64 encoding on the encrypted name. The base64 encoding expands the length of the file name by about one-third.

4.4.2 Path Translations

UNIX machines have the notion of a symbolic link, also known as a symlink. A symbolic link from *file_a* to *file_b* means every access of *file_a* should be transparently redirected to the real file at *file_b*. In the Kiwi system, a similar but more powerful notion of path name translations is used. In the simplest case, path name translations can work similar to symlinks. Kiwid can be set up so all accesses to one path are automatically redirected to another path.

The way to specify these translations is through a configuration file. Each line of this file is similar to the ‘s’ command in `sed` that substitutes a regular expression with a replacement. Lines beginning with a # are ignored and can be treated as comments. The first character in a line is used as the delimiter character. The easiest delimiter to use is probably the space character. Figure 4.4 is an example of how a possible configuration file could look. Note the caret character (^) matches the beginning of the line and the backslash numbers are back-references, referring to the portion of the input string that matched the given group in the first regular expression.

Translations can be used for the same purpose as symlinks, by making path names shorter and easier to remember and allowing one to group files logically. For example, in the example file, a shorter name is used to refer to my home directory. Another use is to merge directories on different servers into one directory. For instance, suppose one wanted to have access to all of one’s binaries under `/kiwi/bin` regardless of where the files are actually located. Some binaries may be located on one server and some other files may be located on another server. One could have `/kiwi/bin/emacs` point to a file on one server and have `/kiwi/bin/vi` point to a file on a different server. Referring to programs by short names chosen by the user is much easier than remembering the server specific locations for files.

```
# map my home directory
~/home /server1/home/austin

# Link applications under one directory
~/bin/emacs /server1/usr/bin/emacs
~/bin/vi /server2/usr/bin/vi

# All files under myfiles are located on server1
# If server1 is down, try server2
~/myfiles/(.*) /server1/\1 /server2/\1
```

Figure 4.4: Sample translations file

Some benefits with this translation do not exist with simple symlinks. With the use of regular expressions and back-references, a user can specify an entire class of links that is not possible if one wanted to specify symlinks manually. In addition, multiple translations can be specified per regular expression. For example, suppose some file is replicated and located on several different machines. One would like access to this file even if one of those machines goes down. Users may not care whether they retrieve a copy of `emacs` from `server1` or from `server2`, as any accessible version will suffice.

By using this path name translation mechanism, we obtain, in a very simple manner, some of the same advantages of other file systems that replicate files and balance the load over multiple servers. However, this scheme is not quite the same as the traditional methods, because the list of translations is tried in order until one that works is found. True load balancing would work much better by randomly picking a file server. On the other hand, priorities can be assigned to servers. For example, suppose `server1` is the primary server and if `server1` is up, this is the server to contact for the files. Only if `server1` is not available should the backup servers be tried. The backup servers could have files a bit out of date, but nevertheless still have reasonably fresh copies of the files. Note this currently proves problematic for file writing. File consistency becomes a challenge if clients could be writing to several different servers.

4.4.3 Caching

Caching is very important for high performance. Several things need to be cached, including the file contents, file attributes, and directory listings. Kiwi caches entire files. On a file open, the file is retrieved by kiwid and placed in a local cache file. The local cache file is used for all file operations, until a close or flush operation is received to write out the file. This is a write-on-close cache writing policy. The cache file is validated on every file open by contacting the server. If a connection with the server cannot be established, the local cache file, if it exists, is always used as being the most recent copy of the file. No method exists to meaningfully handle reconnection with the server if the cache file had been modified while disconnected. To support merging changes transparently would involve implementing some of the ideas of disconnected operation, as seen in Coda [13].

Directory listings and file attributes are retrieved using the PROPFIND method defined in WebDAV. PROPFIND takes a header called depth that can be either 0, 1, or infinity. A depth of 0 indicates the file or directory itself, 1 indicates the directory and all of its direct descendents, and infinity indicates the entire tree rooted at the directory. By giving a depth of 1 when we do a listing of a directory, we are returned the file attributes for all the files in the directory. On successive calls for file information on those files, with the system call `stat`, this file attribute cache is first consulted. Executing a `ls -l` or `stating` every file in a directory is about as efficient as retrieving only a directory listing. This caching reduces the load on the server, as fewer connections are needed. The cache information is currently set to expire after 15 seconds.

All cache information in Kiwi is stored in files on disk and, thus, is persistent. This means kiwid can recover easily from a crash, as no information will be lost between sessions. Because few of a user's files are likely to have changed between reboots or sessions, making the cache persistent can save time in refilling the cache on startup.

Consistency

Our system only requires clients to establish connections to the servers. Some other file systems including AFS have the server contact clients, usually as part of the cache consistency protocol. We assume clients manage the cache consistency themselves. Clients try to validate their cache copy upon opening

a file. If users need stronger consistency guarantees, they can use external systems, such as version control systems like CVS, to keep track of changes from different users. By pushing all the work into the clients, we make the server as simple as possible. In addition, this allows the client to be behind firewalls, and the protocol still works. Whereas many other systems focus on how to implement cache consistency, we took what we saw as the easiest approach that would work reasonably well. In addition, we did not try to emulate the same semantics as a local file system, such as Sprite. Performance and simplicity were deemed more important than full compatibility with UNIX semantics.

4.5 User Client

The user interacts with kiwid through another program simply called `kiwi`. The `kiwi` program includes a collection of commands to handle various tasks, such as authentication and manually downloading files from a server. If kiwid is not already running, kiwid is automatically started in the background by the Kiwi commands needing to communicate with it. Thus, a user does not need to worry about manually starting the kiwid program. A description of some of the commands implemented for use with Kiwi follows. All these are accessed as `kiwi [command] [OPTIONS]`.

4.5.1 add

This command adds a file containing the user's credentials into kiwid's credentials cache. The user's credentials consists of their certificate and private key and in our system is mainly stored in a standard PKCS12 file. The file can and should be encrypted to prevent accidental exposure of the user's credentials if the local machine is compromised.

Suppose a user has the credentials necessary for connecting to machine *server* in file `server.p12`. Before accessing files, a user would type `kiwi add server.p12` and the command would ask the user for the password to decrypt the key. From that point on, the user will have access to all files under `/kiwi/server`.

Kiwid can keep track of multiple credentials for different servers or groups of servers. Kiwid chooses the appropriate set of credentials for a server based on a simple set of rules. The user can specify mappings from server names,

using regular expressions, to credentials. If credentials are not found for a server through these mappings, a certificate signed by a CA with the same common name as the server is used if available. As a final resort, kiwid uses a default certificate.

4.5.2 `cat`, `cp`, `ls`

These commands are analogous to the corresponding UNIX programs except they also work for files located in the Kiwi namespace. We can use the `kiwi cat` command to view files, the `kiwi cp` command can copy files from and to a remote Kiwi server, and the `kiwi ls` command allows for listing directories on remote Kiwi servers. For example, a command like `kiwi cp /kiwi/machine/my/file /tmp/localfile` would download a copy of the remote Kiwi file into a local file.

As the kernel module providing a file system has only been implemented for Linux, people on other operating systems would be out of luck if they wanted to access their files on non-Linux systems. But using these programs, they can manually download Kiwi files from a Solaris system or from some other system, work with the file locally, and upload the file again to the server. This is certainly not as convenient as using a transparent file system interface, but it at least makes the system usable on other operating systems.

4.5.3 `encrypt/decrypt`

These commands allow for the manual encryption or decryption of files. Even if the Kiwi file system is not up and running, one would hope to have access to one's encrypted files. These programs can decrypt an encrypted Kiwi file or encrypt a normal file.

4.5.4 `list`

This command lists the credentials Kiwid has in its credentials cache and that were added previously with the `add` command.

4.5.5 `shell`

This is a shell designed to work in a way similar to command line ftp programs. This simple shell allows for using any of the kiwi commands, changing

the current directory, and issuing other commands to access Kiwi files manually.

4.5.6 shutdown

This command is used to shutdown kiwid cleanly, releasing all of the user's credentials from memory, and to perform all necessary cleanup routines.

4.5.7 sign

This is a helper program to create and sign certificates. The default programs in OpenSSL have no easy way to create a certificate by signing with a custom certificate authority. This program can take a certificate signing request (CSR), usually sent to certificate issuers like VeriSign, and creates a certificate from them using the certificate and private key of a local certificate authority. This program can also create certificates from a Netscape SPKAC structure, discussed in section 4.1.3.

Chapter 5

Results

The Kiwi file system works well enough to be usable. One can do most normal things one expects to be able to do, such as reading and writing files. Applications for the most part work as expected with Kiwi files and running programs located on Kiwi servers also works as expected. Development work on Kiwi continues to use Kiwi itself. We will discuss how well Kiwi does in the areas of security, simplicity, and scalability, the areas that we initially listed as design objectives. In addition, we will see how well the file system performs in practice.

5.1 Security

The security of Kiwi depends on SSL, a protocol designed for security. However, even if SSL is secure itself, there may be unknown problems with the OpenSSL implementation or in the way Kiwi uses the library. A code audit is definitely needed, and, as all source code is available, we hope people will begin to look at and poke at the code.

A security issue is that we run the Apache web server as root, something it was not designed for. In fact, running Apache as root required recompiling Apache with a special flag (-DBIG_SECURITY_HOLE). However, we do not think this is a security hole in the Kiwi system. As soon as the connection is made and the client presents a valid certificate, the `seteuid` call is made to drop the root permissions. There should be nothing the user can do through the Kiwi server that could not be done when the user is logged in locally.

In comparison to NFS, the most popular distributed file system, Kiwi

provides an extraordinarily high amount of security. For example, NFS has very weak security, relying on trusting machines based on IP addresses. Besides being able to spoof IP addresses by compromising a trusted machine, an attacker can gain access to all the files on the server available to users from the trusted machine. Kiwi, on the other hand, doesn't trust machines. Even if a client machine is compromised, an attacker cannot access files on servers without valid user certificates. In addition, the network traffic for NFS is unencrypted, so anyone can listen on the network and gather data about accessed files. All network traffic in Kiwi is encrypted using SSL. Even if there may be attacks on SSL, attacking SSL certainly is much more difficult than attacking the NFS protocol. In addition, NFS has no support for encrypting files like Kiwi does.

5.2 Simplicity

Historically, file systems have taken a long time to put together. By using HTTPS as our underlying protocol, we got off the ground almost immediately. With almost no work, we had a working file server. The client side was a bit more work, but we still could reuse much existing code. Although the file system is not yet complete, with many improvements still to be made, tremendous progress has been made considering the relatively small amount of time and the small number of people working on the system.

We were also striving for simplicity in use. Setting up other file systems is sometimes very non-trivial. Setting up a globally accessible AFS system requires the creation of a new cell and registration of the cell with a central source. A file listing all cells is updated and periodically copied over to each AFS client. Under Kiwi, setting up a domain is as easy as setting up a web site, with use of the well-established and mostly decentralized domain naming system (DNS). In addition, setting up a Kiwi system is about as easy as setting up Apache. Many more people have experience setting up a web server such as Apache than other file servers like NFS or AFS servers. The use of a technology people are familiar with should make Kiwi simpler to set up and to use than other file systems.

5.3 Scalability

Our goal was to have a globally accessible file system, and so scalability was certainly a concern. However, the scalability of the system has been largely untested. But the emergence of the Web and the growth of web servers is a sign of the scalability of HTTP/HTTPS as a network protocol. One problem is that the Web was designed for read-only data and not for a full read-write file system, so Kiwi may not be as scalable as the Web has been. Moreover, HTTP is by far more common than HTTPS on the Web, and whether the computationally expensive HTTPS protocol can scale well is unknown.

The scalability and potential growth of a system can be measured partly by how easy the job of the administrator is. For example, maintaining and administering a large number of NFS systems is complex and difficult. If a NFS server is added to a system or a new client should gain access to a server's files, potentially many configuration files need modification. NFS requires the explicit mounting of every server to be accessed, causing problems when servers go down or files change locations. Clients will often hang waiting for NFS servers that may never come up. In addition, mounting new servers and editing configuration files require someone with superuser privileges. Always having a system administrator around to perform these tasks is not scalable and is inappropriate for a global file system. Besides system administration, NFS was simply not meant to work in a large distributed environment, and there has been empirical data supporting this lack of scalability in NFS [10].

In Kiwi, servers can go up and down, just as web servers go up and down, and clients won't be impacted much. The Web could not possibly have grown to its size today if someone had to centrally manage it when servers went up or down. The number of distributed web servers already in existence, managed by different people everywhere, shows the potential for a system such as Kiwi to scale globally.

5.4 Performance

An unresponsive filesystem can become unusable, so we need the performance of Kiwi to be reasonable. With the overhead of cryptographic routines, Kiwi is expected to have worse performance than a non-cryptographic file system, but how much worse is it?

	Kiwi (no enc)	Kiwi (enc)	NFS
MakeDir	2	2	0
Copy	13	15	1
ScanDir	4	6	1
ReadAll	5	8	1
Make	20	22	9

Table 5.1: Andrew benchmark running on single machine (results in seconds)

	Kiwi (no enc)	Kiwi (enc)	NFS	AFS
MakeDir	3	3	2	1
Copy	18	18	8	3
ScanDir	6	6	3	2
ReadAll	7	8	4	2
Make	30	27	9	9

Table 5.2: Andrew benchmark running over network (results in seconds)

5.4.1 Benchmark Tests

We ran some tests with the Andrew benchmark [10]. The benchmark consists of five phases. The first phase creates some directories, the second phase copies a bunch of files, the third phase does a `stat` on every file, the next phase reads every single file, and the last phase starts a `make` process. The benchmark was run both on a single system, with the server and client on the same machine, and also on separate systems connected by a local network. Tests were run both with file encryption turned off (no enc) and on (enc). Note that network traffic is always encrypted using SSL, so the file will be encrypted twice when file encryption is on. For the tests, we used an Intel Pentium III at 700MHz and 128MB of RAM. All tests were run multiple times and averaged. The numbers given in Table 5.1 and 5.2 show the elapsed time in seconds required to execute that phase of the benchmark.

Interestingly, turning on file encryption added relatively little overhead. Overall, the Kiwi system is indeed slow compared with NFS or AFS. However, several things should be kept in mind. No tuning has gone into the system, unlike the other file systems, and the code has not been optimized in any way.

5.4.2 Potential Optimizations

Many components in the Kiwi system could be enhanced and many optimizations could be performed to increase performance.

SSL

The Apache with SSL setup is a very generic solution to the problem of serving files. More features are built into Apache than we currently use. Turning off all but the minimal set of functions needed could improve performance. Part of the slowness compared with the non-cryptographic file systems above is undoubtedly due to the SSL protocol itself being computationally intensive, with every connection requiring modular exponentiations as part of the public key cryptography routines. Inherent limitations exist to how many connections a secure web server can handle at once. Much of the work done to speed up secure web servers can also be applied to Kiwi. Hardware SSL accelerators and other methods such as batching [25] may be able to speed up the SSL handshake.

The HTTP protocol is not a bad protocol itself, and if SSL were completely eliminated, then there could be performance improvements. Eliminating SSL would require that the roles SSL currently plays can be satisfied by the other parts of the system. If all files are encrypted and checked for integrity, then we do not need SSL for privacy. On the other hand, we still need a secure authentication scheme. Eliminating SSL, however, may also remove much of the benefit of Kiwi in accessing files with standard web browsers, as encrypted files will no longer be readable without manual decryption.

Kiwid

Another place for optimization is kiwid. Moving some of the code from kiwid into the kernel would be more efficient by reducing the number of context switches. For example, kiwid, and not the kernel, does most of the caching in user space. So a lookup into a cache requires a context switch that could be avoided. Both the NFS client and server are implemented in the kernel, making them more efficient than the Kiwi client and server. Systems like TCFS have increased performance by moving code into the kernel. Thus, performance in Kiwi could be improved if all code in kiwid were moved into the kernel, although portability and the ease of development would suffer.

Caching

The caching strategy can also be improved. NFS clients cache blocks of files whereas Kiwi caches whole files. Therefore, Kiwi would be extremely inefficient for accessing small blocks from large files. On the other hand, by writing out whole files at a time, Kiwi can be more efficient in some cases than NFS. Because Kiwi only contacts the server on file open and close, file reads and writes are inexpensive compared with a file system that contacts a server over the network for every read or write. To initially retrieve a file, Kiwi can be about 15 times slower than a local file system, but once a file is in the local cache, Kiwi should have the same performance as a local file system. An area for improvement would involve moving all of Kiwi's caches from disk and into the much faster main memory, similar to what Sprite does.

Kiwi uses a write-on-close policy, causing the close operation to stall while the file is written out. As most files are usually open only for a very short time [22], this type of cache policy probably does not reduce network traffic significantly. Another approach would be to delay the write for even longer and to allow the close to complete immediately.

Name Lookups

Another issue may be the file name lookups. In contrast to file systems like NFS and AFS, files are not given abstract identifiers in Kiwi. Every request on a file passes the full path name to the file in the HTTP request line, in the form of the Uniform Resource Identifier (URI). Because all files are referred to by their full path, every access requires a name to inode translation call through the `namei` routine. As the AFS designers discovered, name resolution is a very common and time consuming operation [10]. They thus converted their original AFS prototype that worked with file names into one that only worked with inodes. However, as the UNIX system call interface cannot deal with inodes, AFS needed to add system calls to work with inodes, complicating the system. For Kiwi, a similar change would involve drastically changing the Apache web server in addition to changes in the client.

5.4.3 Other Considerations

When initially implementing Kiwi, we wanted to make minimum changes to the operating system and use existing code unmodified when possible. We did not want to redesign an entirely new operating system or reinvent new libraries, even though starting from scratch may have allowed for extra code optimizations. Thus, in many cases, we traded off performance for simplicity. Although the performance of the system is not terribly impressive at this point and can certainly be improved, the performance may never reach that of NFS or AFS. However, what Kiwi lacks in performance, Kiwi makes up in other areas, especially security.

Chapter 6

Conclusion

In this final chapter, we will discuss some areas for further work and finish with some concluding remarks.

6.1 Further Work

For the first implementation, our primary motive was to quickly develop a working file system. There is, however, still much room for improvement. Much code remains to be implemented, and performance certainly can be increased. Some of the following are areas that can be further developed and integrated with the current system.

6.1.1 File Sharing

To share files, we would prefer to rely almost entirely on client side access control to both push the work to the clients and to make the server as simple as possible. We want to avoid having a centrally trusted group server as used in other systems, so we propose to implement file sharing mainly with cryptographic techniques. All files can be encrypted with a file key. Encrypted files can be publicly readable by everyone. Therefore, to share files, we need a mechanism to share the file encryption key with everyone who needs to read the file.

Users on machines are named in the form of user@machine. Thus, giving permissions to me@mymachine means user “me” on machine “mymachine” should have access to the file. Groups are named similarly. Ev-

ery group must be created by a user and the group itself is named as `user.group@machine`. People can find the public keys of users and groups by having the keys stored in a well known place in users' home directories. Thus, to retrieve the public key for `me@mymachine`, a client would access `/kiwi/mymachine/~me/.kiwi/public/me.key` to obtain a certificate containing the user's public key.

Access Control Lists

Access control lists (ACL) are stored in a separate file. The ACL includes the users and groups with access to the file. The ACL also contains the file encryption key encrypted with the public key of each user or group with read permission on the file. Most of the work can then be pushed to the client. For reading files, clients access the ACL, decrypt the file encryption key using their keys, and then gain access to the file.

Although nothing is lost if a random attacker sees an encrypted file, a random person should not be able to write over any file. Thus, we still require the server to control write access, as write permission cannot be managed by clients. To successfully write a file, clients need to prove they have access to a write key specific to the file. Each ACL contains a separate file write key encrypted in the same manner as the file encryption key used for reading. When an encrypted file is first written, the server is told the write key for the file. For clients to successfully write a file, they must prove they know this write key. It must also be possible to write the ACL file. Having write permission on the ACL file itself is equivalent to the admin permission in AFS and allows for changing the ACL. The ACL file contains another key used for modifying the ACL itself.

When a request is made for a protected file, the client first checks the ACL. The client is responsible for determining a way to access the file if possible. The client maintains a cache of the keys of all groups known to the client. Each group listed in the ACL is checked with the current key cache for a match. If the client finds a match, they can decrypt the appropriate key and then perform the operation requested.

Groups

The next issue is how groups should work. The job of managing groups should not be based on a separate trusted server, but should be done in

as decentralized a manner as possible. Each user should be able to define groups in a fashion similar to how groups work in AFS. A hierarchy of groups should be possible. If $group_1$ contains $group_2$, then adding a user to $group_2$ will add the user automatically to $group_1$. This is important for scalability but presents an implementation challenge. Without a central group server as in other systems, how does a user know the groups he is a part of?

Users need a means of discovering the groups they are a part of or of adding these groups manually. When a new group is created, a group key is generated. For every new member of that group, the group owner creates a file in a public group directory containing the group key encrypted with the new member's key. Checking if a user or group is part of another group becomes as simple as looking for the existence of an appropriately named file under the group directory `/kiwi/mymachine/~me/.kiwi/public/groups/` and using that to download the group key.

WebDAV

The WebDAV group has put out a specification for access control lists to control access to resources [34]. The goals are close to our own in promoting sharing and security, although the scheme proposed by the WebDAV group relies on the server to control access. As of this time, however, the specification has not been finalized and no known working implementations exist. However, this is still a possible direction to go that would follow our design philosophy of relying on existing code and standards whenever possible.

6.1.2 File Migration

In the current implementation, naming files works like a traditional distributed file system. The client needs to know the name of the server that contains a desired file to access that file. With some files such as application binaries, users do not care about the specific server used to access the files. If a client could request a file, such as `emacs`, without having to know the location of the file, there would be *location transparency*, where the name of a file does not correspond with the physical location of the file. *Location independence* is the idea of always using the same name to refer to `emacs`, even if the physical location of the file changes from one machine to another machine [15]. We must also consider that some files such as `/etc/passwd` are specific to each machine and cannot be migrated.

This ability to migrate files around without changing the names used to refer to the files can be a very useful step in a truly global and scalable system. File migration can also help in balancing load. Most load balancing done today with web servers relies on using the Domain Naming Service (DNS) to randomly distribute requests across a set of servers. Although DNS load balancing works for a file system, other ways to balance load would also help. Load balancing can be easy with file migration because files can be moved to and served from lightly loaded servers.

Other than simply migrating files, replicating files may be useful. Server mirroring would permit for a group of servers to stay in sync with each other, allowing all servers to appear identical from the outside. With many identical servers, these servers could be anonymous allowing the client to not care about the actual server providing the files. However, a way to find the names of servers may be needed. Currently, if users initially do a `ls /kiwi`, they will see nothing. They need to know the name of the server to connect to. Perhaps something similar to the search interface available on the Web would be helpful.

6.1.3 Caching Strategies

Caching and consistency issues still need further exploration. At this point, the Kiwi system only has a limited amount of caching and is not very smart about caching. Different types of data can have different caching strategies. Some data is public and mostly read-only, e.g., application binaries, whereas other data is changed very often. Are there benefits in making a distinction between different types of files? In the self-certifying file system, increased performance and availability were achieved given only read-only data [8]. If one could give hints about the type of a file, such as if the file is read-only, more intelligent caching schemes may be available.

Currently, cache files are not automatically removed, in most situations. Bounding the size of the cache is something to be added. We originally had the view of a system where a single user logs in, does whatever needs to be done, stores the local cache files, and everything would then be wiped clean for the next user. In addition, no sharing of cache files between users on the same machine is possible, so even if two users access the same file, they will each have their own cache copy.

Much research into methods for caching Internet services has been done. For example, Harvest [7] is a hierarchical cache system. Also many proxy

web servers have been implemented and are in use today. Although these systems were mainly designed for the static, read-only data on the Web, many of these ideas should be transferable to a file system, as much of the data on a typical file system is read-only. Exploring whether these existing caches can be used beneficially for this project would be an interesting direction.

6.1.4 File Encryption

Currently, no partial encryption and decryption of files is possible in Kiwi. Even if a user only wants to change one byte in a file, the entire file will be decrypted and re-encrypted. What is necessary to allow for encrypting and decrypting blocks in files instead of the whole file together? Finding arbitrary blocks in the encrypted file must be possible. The problem with prepending data such as the file encryption key and IV to the file is that file are pushed off block boundaries. One alternative is to store encryption information elsewhere. CFS stores an IV inside of the group id (GID) field of the inode. However, the group id can be changed outside of the system, so assuming its immutability is problematic.

6.1.5 Key Management

Because there are potentially many keys in the system, key management becomes a critical issue. One issue in key management for file systems involves lost keys and forgotten passwords. If a key is lost, an encrypted file can become useless. We could perhaps use a key escrow service, allowing a trusted entity to recover the file in an emergency. If no one entity is trusted, splitting the key up to multiple parties would ensure that only if all parties worked together would they be able to recover the key. Another possibility is to design a smarter escrow service using smart cards [6], logging all uses of the key on the smart card, so the user can determine, after the escrow period, how many times the card had been used.

We currently store the keys for users on the file server itself, but we assumed the untrustworthiness of the file server. A malicious file server could provide its own public key when a user requests the public key for some other user. Then the file server will be able to decrypt the file. If the certificate authority (CA) and the file server are not the same machine, unauthorized access would only occur if the CA and the file server collude. This whole issue of public key management is probably not a simple problem to solve.

One approach taken by the self-certifying file system [16] is to avoid the issue of key management in the file system itself. File names contain public keys of the server, but how those names are generated is handled somewhere outside the system.

6.1.6 Certificate Revocation

A related issue to key management is revoking rights. Certificate and key revocation is a difficult issue cryptographic systems have to deal with. What happens if users no longer should have access to files that they used to have access to? Re-encrypting these files with a new key is necessary, but finding the files to re-encrypt could be difficult. Another concern is that re-encrypting all the files at once would be inefficient. Certificate revocation is an important issue in a certificate infrastructure, but in Kiwi, as in many other systems, this problem is difficult to handle, and, as a consequence, is often ignored. However, adding an efficient method to revoke permissions is important for a practical file system.

6.1.7 Mobile Computing

How can the system handle mobile computing? Coda [13] allows for disconnected operation for users with laptops. PeStO [27] provides a model for ensuring consistency with a user-specified level of optimistic or pessimistic strategy for caching. As more and more computing resources become mobile, there may be ways to take advantage of the mobility in a file system, allowing for a more accessible and global file system.

6.1.8 Ubiquitous Access

One way to increase the global accessibility of files is to port the Kiwi system to non-Linux platforms. The server should be portable, with Apache already being used on a variety of platforms. Kiwid should be mostly portable also, so porting to other operating systems only requires writing a file system module to talk with kiwid. Because the design pushed much of the work into kiwid and made the kernel module as simple as possible, not too much code is necessary to port the file system to another operating system. Another way to increase accessibility is to allow web-access to files with browsers other than Netscape, e.g., with Microsoft Internet Explorer.

6.2 Final Thoughts

A file system like AFS provides much better file system services than the Web, yet it is not AFS but rather the Web that has grown exponentially and has spread around the world. Why would people choose protocols like HTTP over something that could be considered technologically better? A large part of the reason must be simplicity of use. Adding file system services to what people are familiar with should allow for faster adoption and acceptance.

In developing Kiwi, we thus focused on the technologies people are already using. File systems do not need to be created from scratch. Existing protocols like HTTP and SSL *can* be extended to work as a file system protocol. We put together Apache, `mod_ssl`, `mod_dav`, OpenSSL, neon, and, with a few additions, to build a mostly usable system in a short time. Kiwi has many advantages over other file systems by building on existing technology. Unlike any other file system, Kiwi can be used to access files securely using a standard web browser and from behind firewalls. And in comparison with other file systems, the time to develop Kiwi has been quite small, mainly due to the enormous starting code base.

Although the performance is still lacking, Kiwi runs fast enough for most purposes. The entire system, including all source code, is freely available for public use. Nevertheless, Kiwi is probably not stable enough for everyday use, and more work is necessary before Kiwi can reach high stability. Further issues for exploration and areas for improvement include caching, consistency, and sharing. Developing the Kiwi file system proved to be a valuable learning experience, and although taking a system from usable to practical can be a long and difficult road, it can also be quite rewarding.

Bibliography

- [1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user level: the UFO global file system. *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 77–90, January 1997.
- [2] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *15th ACM Symposium on Operating Systems Principles*, December 1995.
- [3] Apache http server project. <http://httpd.apache.org/>.
- [4] David Bindel, Monica Chew, and Chris Wells. Extended cryptographic file system, December 1999.
- [5] Matt Blaze. A cryptographic file system for unix. In *Proceedings of the First ACM Conference on Computer and Communications Security*, November 1993.
- [6] Matt Blaze. Key management in an encrypting file system. *USENIX*, June 1994.
- [7] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. *Proceedings of the 1996 USENIX Technical Conference*, Jan 1996.
- [8] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.

- [9] Kevin E. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, MIT, 1998.
- [10] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Robert N. Sidebotham M. Satyanarayanan, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [11] HTTP/1.1 protocol. RFC 2616.
- [12] James Hughes, Chris Feist, Steve Hawkinson, Benjamin Marzinski, Jeff Perrault, Matthew O'Keefe, and David Corcoran. A universal access, smart-card based, secure file system. <http://www.securefs.org/>, February 2000.
- [13] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10:3–25, Feb 1992.
- [14] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
- [15] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), December 1990.
- [16] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *17th ACM Symposium on Operating System Principles*, 1999.
- [17] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

- [18] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.
- [19] NFS protocol specification. RFC 1094 and RFC 1813.
- [20] OpenSSL. <http://www.openssl.org/>.
- [21] Joe Orton. neon. <http://www.webdav.org/neon/>.
- [22] John K. Ousterhout, Hervè Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *10th ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.
- [23] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.
- [24] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.
- [25] Hovav Shacham and Dan Boneh. Improving ssl handshake via batching. In *Proceedings RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 28–43. Springer-Verlag, 2001.
- [26] Secure hypertext transfer protocol. <http://www.terisa.com/shttp/>.
- [27] Michael G. Sørensen. A mobility-transparent model for consistency. Master’s thesis, University of Copenhagen DIKU, 2000. <http://www.garfield.dk/pesto/index.html>.
- [28] SSL 3.0 specification. <http://home.netscape.com/eng/ssl3/>.
- [29] Transparent cryptographic file system. <http://www.tcfs.it/>.
- [30] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. Webos: Operating system services for wide area applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, Chicago, Illinois, July 1998.

- [31] Amin M. Vahdat, Paul C. Eastham, and Thomas E. Anderson. Webfs: A global cache coherent file system.
<http://www.cs.duke.edu/vahdat/webfs/webfs.html>.
- [32] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the sun network file system. *Proceedings of the 1985 USENIX Winter Conference*, pages 117–124, Jan 1985.
- [33] WebDAV. <http://www.webdav.org/>.
- [34] WebDAV access control protocol. <http://www.webdav.org/acl/>.
- [35] WebNFS. <http://www.sun.com/software/webnfs/>.
- [36] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system.
<http://www.cs.columbia.edu/~ezk/research/cryptfs/index.html>.