sjsFS: A FTP-transparent User-Level File System

Shane E. Cruz, Jang S. Kim, Steven B. Lustbader
{scruz, jangkim, sbl}@ mit.edu

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper describes sjsFS, a new user-level file system that provides transparent access to FTP servers on the Internet. The current use of FTP servers is limited to using special software on a client machine to access data that is stored on these remote server machines. Unfortunately, this method of accessing files requires the user to explicitly download each file from these servers before he/she is able to use them. This unnecessary file-transfer forces the client machine to jump through an extra hoop before being able to make use of remote files. We have developed a new file system that allows the client to access files on any FTP server with the whole transfer process being masked to the user. By bringing remote files into the local file system namespace, we remove this extra hoop to create transparent read and write access to files on remote FTP servers. To achieve this goal, sjsFS makes use of a toolkit that helps extend the functionality of a basic NFS loopback server at the user level.

1 Introduction

As the Internet continues to expand and as high-speed network connections start making their way into the home, as well as the workplace, more opportunities arise to make use of this large array of networked computers. Even with the price of disk storage dropping rapidly, it is often much more convenient to make use of files that are located on remote machines in order to maintain consistency or to reduce local complexity. An ideal situation is one in which the user has the ability to make use of every remote file on the Internet with the appearance that these files are completely local to them.

The Sun Network File System [1] is a popular protocol that allows this form of transparent access to shared files that are connected together through a network. This protocol is extremely useful to enhance local area networks by allowing common files to be stored on file servers so that all clients that are connected to the LAN can make use of them transparently. Unfortunately, NFS is not suitable for the Internet as a whole because it is not easy for remote machines to work as NFS servers for the entire Internet. This is true because the Network File System does not have adequate key management in order to ensure security at a global scale [6].

This is where the file transfer protocol [2] steps in. FTP is a widely used protocol that allows client machines anywhere on the Internet to connect to remote machines (running file transfer protocol server software) in order to send and receive files. This efficient and reliable protocol typically involves a client side application connecting to the remote server and then allowing the user to log into that machine, often anonymously. Once the client is connected to the server, the user can browse the files (almost as if the files are local) that are viewable to that user and then transfer files between the two machines using FTP commands. Popular FTP clients, such as ones that come standard with most operating systems, attempt to make the user feel as if the files are local, after the connection is established, by making use of several UNIX-like commands such as *ls*, *cd*, and *pwd*.

Unfortunately, the limiting factor with the FTP protocol is that the file access is not transparent to the user and therefore it can be very inconvenient in some situations. Before these files can appear local to the user, the file transfer protocol requires that a user connect and login to the remote server. Even after being connected, the user must explicitly download the files desired before being able to view or make use of them. A much better solution allows users to access these files on the remote FTP servers and have them appear as if they exist in their local file system tree. This paper presents a new file system that removes this added inconvenience and makes all files located on remote FTP servers to transparently appear to exist on a local client machine.

1.1 Motivation

As high-speed connections become more common and as good FTP server software becomes available (and standard) on most computers, the number of FTP servers connected to the Internet continues to increase. As seen with the explosive use of file-sharing utilities such as Napster [3], Freenet [4], and many other similar tools, users are willing to share the files stored on their local disks with the world in exchange for the use of files located on remote machines. With this continual increase in the number of FTP servers and the community's desire to make use of these machines, it seems necessary to make these connections as seamless as possible.

In addition to this, many FTP servers continue to allow anonymous access to their machines and therefore allow any client to have at least read access to a large number of files on their system. This fact makes it even more reasonable to make this connection transparent to the user because the login process is unnecessary and therefore should be removed.

In building sjsFS, we hope to demonstrate the many ways in which transparent FTP access can be a useful feature to present to all users with a connection to the Internet. By removing both the unnecessary login process and the burdensome file-transfer procedure, we hope to present a system that extends the basic functionality of the UNIX operating system in order to make remote file-sharing a much easier task. More specifically, we hope to present a system that will allow a client to listen to an audio file on a remote machine or to share family photos with the appearance that all the files are local to each client.

1.2 The Goals of This Work

In this project, we design a solution that creates this seamless and transparent access to remote FTP servers. We present a new user-level file system that simplifies the file-sharing process by giving the user a system that provides:

- *Increased Usability*: sjsFS allows clients to access files and information stored on remote FTP servers with the connection and transfer happening behind the scenes. By masking these unnecessary actions, the user has less steps to complete and therefore this process becomes much more convenient. By simplifying the FTP process, we increase the usability of the system by extending the current UNIX file system.

- *Similar Functionality*: In order to make a convincing argument that sjsFS is worthy to adopt, we must create this extended file system in a manner that maintains similar functionality to the current system. Our chosen implementation just extends the current functionality of the UNIX file system and the functionality remains the same throughout the new parts of the system. File system accesses outside of the special intercepted directory are just passed on and handled by the standard UNIX file system, while new accesses within this directory mimic the behavior of the rest of the system.

- *Matching Reliability*: Another important goal of our design is to ensure that our new file system does not weaken the reliability of the file system in any way. As described in the design and implementation sections of this paper, we make use of a simple NFS loopback server toolkit that provides very high reliability that is not used in other FTP file system implementations.

At first glance, these goals seem like they are difficult to achieve and would require a great amount of complexity and coding. Again, by building sjsFS on top of the basic toolkit for user-level file systems, we demonstrate that these goals can indeed be reached in a system that is both non-complex and fairly simple to implement.

1.3 The Approach

We decided to build our system as a user-level file system on top of the toolkit suggested by David Mazières [5]. By creating a NFS loopback server, we are able to intercept calls to specific directories and transparently create and close connections to FTP servers. A background ftp daemon class creates these connections and transfers these files as necessary in order to mask the unnecessary FTP procedures. In using this toolkit, we successfully eliminated many of the other problems encountered in creating user-level file systems such as low-performance, deadlock, restrictions on file system structure, and the need to reboot after software errors [5]. The details of the design and implementation will be discussed in the next section of the paper.

1.4 The Rest of This Paper

In the following sections of this paper, we will give an overview of the design of the system and how it works. We will also present a brief overview of some of the implementation details and decisions. Then, we will give an analysis of the sjsFS system and discuss how well we met our goals and mention possible work that could extend this project in the future. Near the end of the paper, we will discuss the several other projects that have set out to create this transparent FTP access and analyze what makes these systems different from what we have done. Finally, we will conclude with a brief look at what we have achieved and what we have learned in the process.

2 Design Overview

sjsFS was designed to provided transparent FTP file access and transfer for any client/ftp-server connection. The system brings any file on any remote ftp server into the local file system namespace, assuming the client has the permissions to access that file. With seamless transfers and overall transparency in mind, sjsFS was designed to enhance file-sharing capabilities by removing the unnecessary steps taken by FTP client software. The two main components of the design are the user's interaction with the local file system and the architecture of the NFS loopback server and FTP server communication.

2.1 File System Interaction

In order to create a useful new approach to retrieving data on FTP servers, we needed a tight integration with the current UNIX file system in order to make sjsFS easy to use. As seen in the next section we implement this transparency and usability by having a user-level loopback server intercept all calls to a particular directory and then do all the FTP connections and transfers in the background. Done in a similar manner as the SFS secure file system [6], we take these intercepted calls and determine what steps need to be taken in order to mask the FTP procedures from the user. This provides UNIX-like access to files on FTP servers around the globe as if the files exist on the client's local machine. A typical read of a remote FTP file in sjsFS can be seen in Figure 1.

```
% cd /classfs/ftp/ftp.my.com/pub/
% ls
readme.txt     sinatra.mp3
% emacs readme.txt &
```

OR

```
 % emacs /classfs/ftp/ftp.my.com
/pub/readme.txt &
```

Figure 1: Opening a remote FTP file in sjsFS. As shown, it is possible to anonymously connect to the remote FTP server, transfer the file to the local disk, and execute a system `read` all with one standard UNIX-like `read` call.

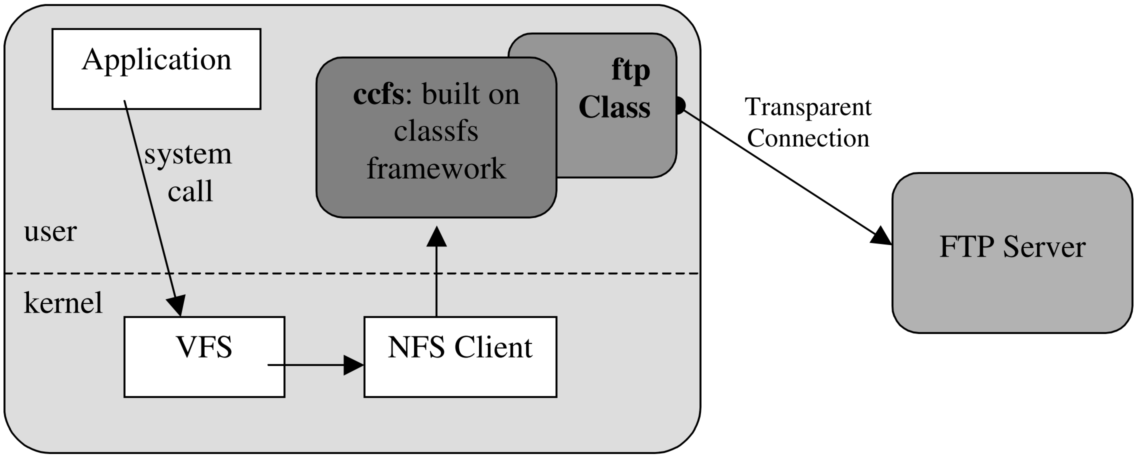

Figure 2: The NFS loopback server architecture of sjsFS

As seen in this figure, the anonymous FTP connection to the `ftp.my.com` server is masked to the client and it appears to be part of his local file system namespace. Once this connection is established in the background, the user is able to interact with the remote FTP file system in the same manner as the rest of the local file system. Upon opening the file (as seen above with Emacs), sjsFS transparently downloads the file to a temporary directory on the client's local machine and then allows the client to view the file. A detailed description of the read and write access allowed by sjsFS is described in *section 2.3* of this paper.

2.2 Architectural Overview

Figure 2 illustrates the overall structure of the sjsFS system. Implemented as an NFS loopback server, sjsFS sits in the user level and intercepts all calls for the `/classfs/ftp` directory and correctly makes the necessary FTP connections and file transfers transparently. This hidden connection and file transfer is done in the background by the `ftp` class daemon. As shown in the next section, this daemon creates a reliable TCP connection to the specified FTP server and acts as a middle-man between the client and the server, relaying all necessary file system commands. From these daemon responses, the `classfs` loopback server can reply to the system call RPCs to achieve the behavior desired.

As stated in our goals, we intended to design the user-level file system in a way that did not significantly increase the overall complexity of the system. The architecture of sjsFS is a simple extension of the basic SFS toolkit [5] with an extra daemon to interact with the FTP connection. By keeping the complexity of the system to a minimum, we help to ensure that we can avoid all the loopback server issues that are reviewed in [5].

3 Implementation

The intentionally simplistic design mentioned in *Section 2* leads to a fairly simple implementation. By making basic extensions to the `classfs` framework, we leave the majority of the code for sjsFS to the process of interacting and parsing the requests and responses from the FTP servers. The two major components of this system are the `classfs` loopback server and the `ftp` daemon class.

3.1 classfs – The NFS Loopback Server

In [5], David Mazières presents a toolkit for creating asynchronous user-level file systems in the C++ language. He presents the skeleton of a basic user-level file system that we chose to build upon. We decided to take advantage of the flexibility provided by bringing the file system to the user level in order to easily interface with the `ftp` daemon.

Our changes to this class involved correctly intercepting certain RPC calls and passing them along to the `ftp` daemon in order to interact with the FTP server. When the user attempts to access any file located in a subdirectory of /classfs/ftp, the `classfs` system must create the `ftp` daemon object if necessary and then tell the daemon what it needs to execute from that point forward.

For example, when the user tries to do a `read` call on a file that is located in the `/classfs/ftp/ftp.my.com/pub` directory, the `classfs` object creates a new `ftp` daemon object to connect to the `ftp.my.com` server anonymously. The loopback server then notifies the daemon that it needs the requested file to be transferred to the local disk. In the background, the daemon will use the FTP protocol to correctly `get` the file and will place the file into a temporary directory on the client's local disk. After this is done, the daemon will return the value of 1 (indicating success) to the `classfs` object. Then, `classfs` correctly switches filehandles in the reply RPC in order to trick the user into thinking that the file actually exists in the virtual directory, when in reality it only exists in the temporary directory. More on this virtual directory mapping can be found in *Section 3.3* of this paper.

It is also the job of the `classfs` object to determine how long to keep around the `ftp` daemon object for each connection to a specific FTP server. We decided that it would increase performance to leave this FTP connection open and alive until the user has left the server's virtual directory for a few minutes.

The last major task left up to this loopback server object is to determine when the file that exists in the temporary directory should be removed from the local disk in order to ensure that stale data is not kept around on the local disk. As is all cache consistency problems, this is not always an easy problem to solve. We decided not to worry in too much detail about stale cache data, and therefore we currently just leave the files in the temporary directory until that temporary directory is cleaned out by the standard file system. As discussed in *Section 5* of the paper, it would be possible to run a cleanup daemon in the background to periodically remove files that have existed in the temporary directory for longer than a specified amount of time, but this has not yet been implemented.

3.2 FTP Daemon Class

The `ftp` daemon class was built in order to make the transparent interactions with the FTP servers. Its job is to immediately create the connection and login to the requested FTP server upon its creation. After creation, the `ftp` object sits in the background and waits for method calls from the parent `classfs` object in order to process transactions with the server machine. This daemon (and therefore the FTP connection) remains alive until the loopback server decides to delete the `ftp` object.

In order to ensure reliability, the `ftp` class creates a TCP connection to the FTP server and leaves this socket connection open until the object is deleted. A callback for a `keep_alive` method makes sure that the server does not disconnect from sjsFS because of an idle timeout by sending a NOOP command to the server every 300 seconds.

The major tasks of the `ftp` class in the current implementation consist of performing read-only access on the FTP server. The daemon class, upon request, has the ability to perform operations such as `ls`, `dirlookup`, and `getfile`.

In the `dirlookup` method, the daemon just returns a 0 or 1 to the `classfs` object to indicate whether the specified directory exists on the FTP server and whether or not the client has access to that directory.

The `ls` and `getfile` methods also return a 0 or a 1 to indicate success or failure. Unlike the `dirlookup` method, these methods actually create a new file on the client's local disk. As expected, the `getfile` operation just transfers the requested file to the temporary directory on the local disk. The `ls` operation also creates a file in this temporary directory. This file is just a text file that contains the directory listing of the specified directory. This file is always saved to the correct path in the temp directory under the filename of "`_listing_`".

In order to maintain asynchronous behavior across the entire system, this class does all file I/O and socket communication asynchronously. Making use of the asynchronous libraries also presented in [5], we use callbacks and reference counting to implement the new classes.

3.3 System Communication

The `classfs` object creates a new `ftp` object whenever a user attempts to change directories immediately under `/classfs/ftp`. For example, if the user changes directories to `/classfs/ftp/ftp.my.com`, the `classfs` object must create a new `ftp` daemon object in order to create a connection and login to the `ftp.my.com` server. Therefore, basic communication between the `ftp` daemon and the `classfs` object can be done by explicit C++ method calls. For example, when attempting to change to a new directory on the FTP server, the loopback server can call a *cd* method on the daemon, which can then return a 1 if successful or a 0 if unsuccessful. The loopback server can then generate a reply RPC to pass back to the kernel.

When it is necessary for the loopback server to transfer a file from the remote FTP server to the local disk, the `ftp` class still just returns a 1 or 0 to indicate success or failure. Upon receiving a response of 1, the loopback server now knows that the file requested is now present on the local disk. The system maps files to a similar directory hierarchy, but places them into a temporary directory on the local disk. An example of the mapping from "virtual" directories to actual local directories can be seen in Figure 3.

Virtual Directory:	Actual Local Directory:
/classfs/ftp/<directory>	/tmp/ftp/<directory>

Figure 3: Mapping of Virtual Directories in sjsFS

Once the daemon relays the information to the loopback server that the file is now on the local disk, the loopback server can mask the virtual directory by swapping file handles in order to make it appear as if the file actually resides in the virtual directory. From this point, the client machine has received the remote FTP file and placed it into the local file system without any knowledge of the FTP login process and the file transfer procedure. The series of transparent steps taken in order to view a remote FTP file can be seen in Figure 4.

1. Application makes a call to read a remote file
2. `classfs` intercepts the system call
3. If the associated `ftp` object is not already created, `classfs` creates it in order to open the transparent connection to the FTP server
4. The `classfs` class notifies the `ftp` object that it needs the specified file
5. The `ftp` class daemon transfers the file to the local machine and places it in the correct directory as specified in Figure 3; tells `classfs` it is done
6. `classfs` switches the filehandles to make the actual file on the local disk appear to be in the virtual directory under `/classfs and sends a REPLY RPC to the kernel`

Figure 4: Steps to view a remote FTP file in sjsFS

3.4 Current sjsFS Properties

In the current implementation of the sjsFS file system, the following properties and restrictions apply. Each of these could be further expanded in order to make sjsFS even more usable, but we did not have the time necessary to implement this functionality.

- *Anonymous FTP access*: Currently, FTP server access is limited to machines in which anonymous FTP access is allowed. This seems to be the most significant case in the file-sharing situation, and therefore was the focus of our design. *Section 5* of this paper will describe a possible way to implement access to servers that do not provide anonymous FTP access.

- *FTP Server Software*: Due to the time-constrained nature of this project, we decided to focus our attention on correctly parsing and interacting with the software from one specific

FTP server. Fortunately, many FTP sites are based on the same, or very similar, set of commands so this constraint would be easy to eliminate in the future.

- *File System Commands*: Again, in order to correctly design and build a working system in the time we were allowed, we decided to limit the possible set of file system commands that could be used within the `/classfs/ftp` directory structure. In the current implementation of sjsFS, we focus on providing read-only access to anonymous FTP servers. We limit our set of file system commands to mainly directory navigation and system read functionality. By simply extending the `ftp` class daemon, the implementation could easily be extended to provide for more file system commands.

4 Analysis

In this section, we take a look at the sjsFS system we have built and determine if we have achieved what we set out to do. We also discuss the differences between using this user-level file system and using standard FTP client software to access files on remote FTP servers, presenting examples of where sjsFS could prove to be much more convenient than a typical FTP client.

4.1 Meeting the Goals

As described in *Section 1.2*, the major goals of this project were to create a user-level file system that provides transparent client access to FTP servers on the Internet. In doing so, we hoped to build a system that improved on standard FTP clients by increasing usability, providing similar functionality, and maintaining reliability.

Even though we only implemented this system to provide read-only access to anonymous FTP servers, we still believe that we achieved a better usability than that provided by the standard UNIX file system and a typical FTP client. Since a large number of FTP servers are used as file-sharing devices, it is often the case that the servers are only needed for anonymous read-only access. In this situation, it is fairly obvious that sjsFS makes viewing remote FTP files a much more seamless and easy process. By simply extending the functionality of the `ftp` daemon class and with minor changes to the `classfs` code, it would be possible to extend the system to provide access that is not anonymous and also provide write access to FTP servers that permit this.

Following in a similar fashion, we feel that even our subset of commands that we enable in our implementation provide adequate functionality for community file-sharing. Since we focus mainly on the read-only access to these servers, we feel that directory navigation and system `read` capabilities are the two major components necessary for a fully functional system. Therefore, we have achieved our goal of providing similar functionality to that of the standard UNIX-like file system.

Lastly, we take a brief look at how sjsFS maintains the reliability that is provided by the standard method for FTP access. By limiting the complexity of the design, building on top of the SFS toolkit, and working over reliable protocols, we extend the usability of the system without reducing the reliability in any way. More details on the reliability of the asynchronous SFS toolkit can be found in [5], but are beyond the scope of this paper.

4.2 Simplicity

A major design decision that we faced was determining how complex we were willing to make the system in order to achieve better usability and extended functionality. One of the main advantages of sjsFS over other FTP file systems presented in *Section 6* is the basic design and implementation that we chose. By taking advantage of a powerful toolkit for creating user-level file systems, we were able to easily mask the FTP interactions without greatly increasing the overall complexity of the system. In any system design, it is almost always the case that reducing complexity often leads to a better working system that is much easier to debug. We are very happy with the design and implementation decisions that we have made.

4.3 Attractive Uses of sjsFS

In this section, we give a few quick examples of why sjsFS, even with its current implementation, could be more convenient and useful than the standard FTP client. In all cases, we see that the argument remains similar to those presented in systems such as [3] and [4]. File sharing is becoming increasingly popular and a seamless interaction is needed in order to continue this rapid growth.

1. *MP3 streaming*: Is it really necessary to download and organize every MP3 that you listen to? It is often the case that you listen to a song a few times and then that song is forgotten, permanently taking up valuable space on your local disk. With sjsFS, this file will be downloaded transparently and temporarily to your local disk, therefore saving you the extra space as well as simplifying the download and organizational procedures.

2. *Reading course notes*: In a similar manner, it is often the case that you may download some notes for a school class to your local machine only to eventually forget those files are there. Why require that you explicitly connect to the FTP server, download the files, and organize them on your local disk? As seen throughout this paper, sjsFS would eliminate these hassles, therefore making FTP servers much more convenient to use.

There are obviously several other examples and reasons why this FTP-transparent user-level file system could be very useful, but due to the length-requirement of this paper, these are left up to the reader's imagination.

5 Future Research

As we review the current implementation of sjsFS and take a look at the above analysis of our system, we see that there is much room for extending the functionality of the file system in the future. The strict time limitations didn't allow for us to explore far beyond the basic functionality that we were able to implement, but below are a few ideas for future work to the sjsFS system that would make it even more attractive to replace standard FTP client software:

1. *Stale-Cache Cleanup Daemon*: Implementing a daemon that periodically scanned the temporary directory for files that persist for an extended period of time could be one way to attack the cache consistency problem. Another option would be to have the `classfs` object create a callback to delete the files a specified amount of time after they are transferred from the server.

2. *Non-Anonymous FTP Access*: Several options exist in order to provide access to FTP servers that do not provide anonymous access. One option is for the username and password to be passed in the pathname to the `ftp` daemon object. The daemon could then successfully login to the non-anonymous server. Other ideas such as tickets, cookie-like authentication, and user-prompting could also serve as a suitable method for access to non-anonymous FTP servers.

3. *Write Access to FTP Servers*: This extension to sjsFS would simply require a few additional methods to be added to the *ftp* daemon class in order to correctly send files to the FTP server.

4. *Extended File System Functionality*: Extend the list of possible file system calls in the virtual directories to allow for better usability with operations such as `grep` and other neat utilities.

5. *Support for All FTP Server Software*: Alter the `ftp` daemon class to correctly interact not only with one specific FTP server software package, but with all popular packages used.

6 Related Work

There have been, and still are, many projects that attempt to provide transparent FTP file access in a user-level file system. Even though many of these systems have been built with similar goals, and therefore similar usability arguments, the design and implementation choices have been different than that of sjsFS and therefore we believe we can achieve a more stable system while maintaining high performance.

Older projects such as the Alex FTP File System [7] and the User File System [8] were designed in a slightly different fashion than our system. By replacing the NFS file server with a program that translates between the NFS and FTP protocols, these systems were made to be very portable, yet they were plagued with slower performance. Due to the slow performance, these projects have been dropped and replaced by newer systems that are much faster. Another older project was the SIGOPS FTP File System developed at the University of Illinois [9]. This project was similar to ALEX, but it was implemented in the kernel in hopes of better performance. It appears that this project has also been abandoned, probably due to the inflexible nature created by working in the kernel.

More recently, projects such as UserVFS [10] and FTP FS [11], ftp2nfsd [12], and the Internet File System [13] have had very similar goals to sjsFS and vary mostly in the chosen implementation of the system. We hope to better these systems by making sjsFS more stable, more flexible, and at least as fast. By building on top of the SFS toolkit, we are able to create a user-level file system that is both more flexible than several other designs and also that avoids several NFS loopback server issues faced in other implementations. This toolkit will also help us avoid the problems deadlock, file system restrictions, and the need to reboot after software errors. Since the limiting factor in our design should be the network, our system should match the performance of the other systems currently available.

7 Conclusion

sjsFS is a convenient user-level file system that provides transparent access to files on anonymous FTP servers. Being built with simplicity, usability, and reliability in mind, sjsFS makes use of an asynchronous toolkit for creating user-level file systems. Our design and implementation choices led to a system that is both less complex than other current systems and also one that is very reliable and stable.

Our analysis of the sjsFS file system demonstrated how this flexible file system in the user level could provide a much-needed convenience to those clients who engage in file sharing through the use of anonymous FTP servers. sjsFS masks the unnecessary connection, login, and file-transfer procedures from the user, therefore making file-sharing over FTP servers a much more seamless process.

The prototype implementation of sjsFS still leaves many new features to be explored, as seen in *Section 5* of the paper. Fortunately, by creating this file system in the user level, we have developed a highly flexible system that does not require much modification in order to expand the current functionality. The future of sjsFS could lead to a system that could expand beyond the file transfer protocol in order take further advantage of the large network of arrayed computers that we call the Internet.

8 Acknowledgements

We would like to thank Frank Dabek and Robert Morris for the help in setting up the SFS toolkit, debugging our SFS key management issues, and interacting with the FTP servers. We would also like to thank all those who helped review this paper and provided insightful comments.

References

[1] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119-130, Portland, OR, 1985. USENIX.

[2] J. Postel and J. Reynolds. *File Transfer Protocol*. Network Working Group, RFC 959, October 1985.

[3] The Napster File-sharing Community. http://www.napster.com

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. *Freenet: A Distributed Anonymous Information Storage and Retrieval System*. In Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, 2000. International Computer Science Institute.

[5] David Mazières. *A toolkit for user-level file systems*, Proceedings of the Summer 2001 USENIX, Boston, MA, June 2001, pp. 261-274.

[6] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. *Separating key management from file system security*. Operating Systems Review 34(5), pages 124-139, Dec. 1999.

[7] The Alex FTP File System.
http://www.ludd.luth.se/~kavli/alex.html

[8] The User File System.
http://www.goop.org/~jeremy/userfs

[9] SIGOPS FTP File System. University of Illinois, Urbana-Champaign, 1996. http://www.acm.uiuc.edu/sigops/ftpfiles.html

[10] UserVFS.
http://www.sourceforge.net/projects/uservfs

[11] FTP FS. http://ftpfs.sourceforge.net

[12] Deepak Gupta and Vikrant Sharma. ftp2nfsd - A Portable and Extensible FTP-to-NFS Gateway. Indian Institute of Technology. http://www.cse.iitk.ac.in/users/deepak/ftp2nfsd

[13] Adnan Ilik, *Internet File System*, Research Paper in 6.894, Massachusetts Institute of Technology. Dec. 2001.