MFS: The Meta−Filesystem

Omprakash Gnawali, Kalpak Kothari, John Leffingwell
{om_p, kalpak, jkleff}@mit.edu
Massachusetts Institute of Technology
December 12, 2000

Abstract

The goal of the Meta−Filesystem is to provide a unified free space across remote Internet storage servers that enables users to store and retrieve files using a standard filesystem interface. Our solution incorporates a top−level filesystem interface built on the NFS protocol, a file transfer module to access remote storage services, and a database which stores file metadata similar to that found in a standard filesystem. In practice, our system has allowed storage and retrieval of large files with greater ease and speed than manual upload and download.

1 Introduction

1.1 Hypothesis

The Meta−Filesystem (MFS) will provide easy accessibility of disjoint free space by providing transparent, unified access to all the available space. In addition, the service will increase reliability and security in comparison to storage on a single remote server. We believe the performance will increase for storage of large files.

1.2 Motivation

We believe the idea of being able to get a unified free space is very appealing. Being able to use a normal file interface to store and retrieve large files will become seamless with MFS. A user will no longer need to manually split and store large files across various storage resources (eg. public FTP accounts, freedrive.com, local hard drives). A simple application of MFS could be one in which individuals may have limited space available (Athena quotas) but require access to much larger free space to store large files. We believe the ability to split and encrypt files and replicate the data across various resources will provide added security and reliability. Our system has a number of interesting applications. One direct application of MFS is to cater to people who wish to share and download MPEG Layer 3 music files. Our system will provide users the ability to upload their music files so they can be made available to all other users for download. This system can be compared to Napster, although the similarity ends in the ability to share and download files and maintain a central, global listing of files. Our system can provide more reliable and faster service since the data will reside on several dedicated remote servers as opposed to one individual server, thereby providing the ability to download independent fragments of a file in parallel.

1.3 Module Hierarchy

The MFS prototype has been implemented in three modules:

- **Filesystem Module**: provides a secure network file system interface to the user

- **File Transfer Module**: encapsulates the functionality to store and retrieve files from remote services (FTP servers, Web storage, etc.)
- **Metadata Module**: maintains the state of users, files, and remote services in MFS and presents an API to update the state.

The File Transfer Module has a dependency only on the Metadata Module, while the Filesystem Module has a dependency on both the File Transfer and Metadata Modules. The Filesystem Module is described in Section 2. Section 3 describes the File Transfer Module, and Section 4 presents the design of the Metadata Module. Section 5 describes the various algorithms used for storage and replication. In section 6 we discuss the performance of our prototype system and section 7 describes several failure recovery procedures. Finally, we discuss some related work in Section 8, and outline a few future improvements in Section 9.

2 Filesystem Module

2.1 Overview

We chose to build the client interface to MFS on top of SFS, the Self–certifying Filesystem [1] developed at MIT. Our system is implemented as a set of modifications to the user–level NFS3 server SFS uses for file transfer operations. We could have implemented MFS as an extension to any NFS server; the choice to use SFS was merely one of convenience since we had ready access to all SFS source code (we can also conceive of a client–only version consisting of a specialized shell which overloads the normal filesystem access commands "ls", "cp", etc.). The NFS3 protocol [2] provides transparent remote access to shared filesystems through a client/server architecture structured around RPC calls. A machine–independent XDR (external data representation) wraps primitive data types during communication between client and server.

Filesystem calls referencing an NFS–mounted directory are intercepted by the NFS client daemon and proxied to the NFS server via RPC. A standard NFS server implementation accesses the referenced file in the export directory in its local filesystem and returns the requested information to the client via RPC. The server in our system was modified to perform the functions specific to MFS: authenticating MFS users, splitting files into fragments, performing read/write/delete operations on remote storage servers, reconstructing a file from its fragments, and updating the MFS metadata database to reflect MFS transactions. We briefly explain the authentication procedure, followed by a description of how our system handles the principal NFS3 RPC calls.

One of our design goals was to make MFS look somewhat like a normal filesystem from the client's point of view. Thus every MFS user has a unique user id, and every file stored in MFS has attributes such as creation data, last read time, and last write time. A user doing "ls –l" in his MFS–mounted directory has access to this information just as in a standard UNIX filesystem.

2.2 Authentication

Our authentication procedure is simple. Until a user has successfully authenticated himself to the MFS server, RPC calls generated by the NFS client will be effectively ignored by the server. To authenticate himself, a user runs a short login script on his machine. This script prompts the user for his MFS username and password and tries to create a file named

"<username>:<password>" in the directory at the MFS server's local mountpoint. An NFS3 CREATE RPC is generated by the client and intercepted by the server, which parses the request and checks the username/password pair against credentials contained in the MFS metadata database. Following successful authentication, RPC calls on the part of the client are handled by the MFS server as described below. A user logs out of the system by executing a logout script which passes a predefined logout string ("mfslogout" in this case) to the server in a CREATE call; following logout, further RPC calls will be once again ignored by the NFS3 server.

2.3 NFS Server Modifications

The following NFS3 RPC calls are of importance to the MFS system: GETATTR (get file attributes), ACCESS (check access rights), LOOKUP (generate and return a file handle), READDIR (return listing of directory contents), CREATE (create a new file), WRITE (write content to a file), READ (read content from a file), and REMOVE (delete a file). The above have been implemented for operation with MFS.

GETATTR is specified to retrieve for the client standard file information including size, user id of owner and owner's group, number of links, time of last access, time of last modification, and time of last attribute modification. This information is used by the client to maintain consistency of its NFS3 file data cache and file attributes cache. In MFS this information comes from the MFS metadata database rather than the local filesystem. ACCESS gets the access rights for a specified file and relays that information to the client. In our system these permissions are irrelevant due to the facts that MFS user authentication is required and that the user does not interact directly with the local filesystem of the MFS server. The server responds by indicating full permissions to the client (read/write/delete/execute).

In a standard NFS3 implementation, a client executes the LOOKUP call in order to obtain a *filehandle* used for reference to a given file in the server's local filesystem. The filehandle consists of a 10–byte sequence generated by the server in response to a client request containing the name of the file for which to generate the handle. This mechanism is used for efficiency, so that the client need not specify the full path of a file after doing a LOOKUP. The client and server are responsible for keeping an internal mapping between each filename and its corresponding filehandle. In the MFS system, a unique file identifier suitable for use as the filehandle already exists: the MFS metadata database indexes each file in the MFS system by a unique integer id. We use a string representation of this unique file id as the filehandle for a file, thus avoiding the need to keep a separate mapping between files and filehandles in the server.

As MFS does not currently provide for subdirectories, the READDIR call has the simple function of getting a listing of all files belonging to a single MFS user (in response to "ls" on the part of the user, for example). This information is available in the metadata database, and is retrieved by a call to the metadata interface specifying the MFS user id of the current user. The client is able to interpret this information and generate a directory listing for inspection by the user.

The CREATE, READ, WRITE, and REMOVE RPC calls deal with both the Metadata Module and the File Transfer Module. Arguments to the FileTransferModule constructor include the type of operation requested ('put', 'get', 'del') and a callback function to be executed upon return of the File Transfer method requested (FileTransferModulePutFile(), FileTransferModuleGetFile(), FileTransferModuleDelFile()). Each callback function takes an integer argument signalling success or failure of the FileTransferModule operation which triggered the callback. Thus each of these callbacks can react to failures of the File Transfer operations:

FileTransferModule(mfs_user_id, filename, transfer_op, callback);
void callback(int success);

CREATE is handled by adding an entry in the files table in the metadata database and creating an empty file in the MFS base directory on the MFS server. REMOVE is handled by FileTransferModuleDelFile() as described in the specification of FTP submodule. Its callback checks for success of FileTransferModuleDelFile(), signalling failure to the NFS client if appropriate.

For the READ operation, the MFS server first checks to see whether the requested file exists locally. If not, the file is fetched to local disk by calling FileTransferModuleGetFile(), which locates the fragments for a file, retrieves them, and concatenates them into a single file in the MFS server's local filesystem. Once the single file has been constructed on the MFS server, the client can read from it as it would any standard exported file. Such reconstructed files in the MFS server's local cache are deleted after some reasonable period of time to limit the lifetime of stale data. Since NFS3 READ and WRITE operations are stateless, the MFS server has no way of knowing when the client has finished reading/writing a file. This has implications for the WRITE operation. When a the server receives a WRITE RPC from a client, it starts a timer set up to trigger a callback function. This callback checks the number of outstanding timers for that client. If there are no outstanding timers (it was the last callback triggered), the file is split up and written to remote storage servers (via FileTransferModulePutFile()). In addition, the file is purged from the local file cache to limit the lifetime of stale data as in the case of READ.

3 File Transfer Module

3.1 Overview

The File Transfer Module encapsulates the functionality to store, retrieve, and delete files from remote services such as FTP servers, web storage, and local hard disk drives. The various protocols to interact with different remote services are included in submodules. In our prototype implementation, we restricted the kind of remote service to FTP servers and created an FTP submodule. Using our prototype as an example, we describe the sample submodule and then present the design of the File Transfer Module.

3.2 FTP Submodule

The FTP Submodule provides an API that enables all basic FTP operations, including the login procedure, file download or upload, and file deletion. File transfer operations are done using asynchronous I/O that allow multiple simultaneous uploads or downloads. The submodule interacts with the Metadata Module to update the state of the system using commits and aborts.

3.3 Transfer Module Design

The File Transfer Module uses the Metadata Module's API to get information regarding a particular file (be it for storage, retrieval or deletion). The information reveals a list of fragments and their corresponding server IDs. Based on the server ID, further information about that particular server is obtained. The server information, among other parameters, reveals the service type and login parameters. For each fragment, the File Transfer Module calls the relevant submodule and initiates the desired procedure (upload, download or delete). If the procedure for a fragment fails, retry attempts can be initiated after a suitable wait period, as is described in section 7.

The high–level interaction between the File System Module and the File Transfer Module occurs via a callback procedure that is passed along by the former as described in section 2.3. Upon completion of the desired action, the latter triggers the callback and returns a success or failure.

4 Metadata Module

4.1 Overview

There is a set of data structures to keep track of storage locations of the file. State information is maintained in persistent storage so that the system can recover from errors. In our experimental system, this set of data structures is implemented as tables in a mySQL database. There is no particular reason why we chose mySQL. Any database server can be used with the current implementation without much change. We chose a standard database package to store file system information because that gives us atomicity, reliability, and all the metadata manipulation functions in abstracted Structured Query Language (SQL) form. In case we need to expand the system, it is easy to get a bigger Oracle database or other commercial database systems. All the metadata manipulation was done using C++ wrapper functions that called mySQL C library. Moreover, a mySQL database system can be easily replicated across multiple servers (using simple shell scripts) that can act as backups in case the primary Metadata Server goes down.

4.2 Metadata Structures

User information is stored in the table called "users." This table has biographical information about the user as well as login username and password to be used to authenticate with MFS. New users can signup using a web–based interface. Files are represented as a row in the table called "files." A file is uniquely identified by a file identifier (fid). The column name stores the name of the file which is unique per user (uid). Each row contains additional information about a file, such as creation date, last write date, last read date, and size.

A file is composed of fragments. Each fragment is uniquely identified by the fragment identifier (fragid). Each row in the "fragments" table represents a fragment and stores the size of the fragment along with its ordering key. The ordering key is a sequential number used to combine the fragments in ascending order of this field when a user issues a read request.

Information about the servers and FTP accounts available to our system is kept in the table called "servers." Each server has a server identifier (sid), hostname, service_type, total_space, rating, and five customizable text parameters and information parameters have meaning that is associated with server_type. Information about a new remote server is inserted manually. In case of FTP, we have used the first column to be the name of an FTP account in that server and the second column to be the corresponding password. The rating for a server is given manually when a server is added. This rating can go from a minimum of 0 to a maximum of 10. There is also a column that keeps track of current simultaneous connections to a server. The value in this column is smaller than the maximum simultaneous connections allowed in this server by the number of active simultaneous connections. A value of 0 means no more connections are to be made to the server before one of the active connections becomes inactive.

There is a table that maintains a mapping between a fragment and servers. When we do replication, the requirement is that we store more than one copy of the entire content of the file. Since a file is composed of fragments, we have to be able to keep multiple server mappings for each fragment.

MFS Metadata Module is used by File Transfer and Filesystem Modules using the MFSMetadata API. Several C++ structs are used to return information about files and servers from the API functions. The structure "mfsfile", for example, has all the standard information about a file required by an NFS server while serving a file. The structure "server" has necessary information about the server to be able to connect to the server and request the files. To implement file transactions there are tables keeping track of pending and committed transactions. The table "file_transactions" records a unique file transaction in each row. To implement a highly granular fault−tolerant system, each fragment transaction is stored in "fragment_transactions." This enables us to retry an aborted transaction at the granularity of a file or fragment. Server throughput is used as a metric in the file splitting algorithm. This is also stored in a table called "server_throughput", also managed by the MFS Metadata Module.

4.3 Metadata API

Here we describe some of the high−level functions defined in the MFS Metadata API:

Authentication−related Functions:
authenticate(username, password) returns a positive uid if username and password match.

Filesystem−related Functions:
listfiles(uid) returns a list of file structs
listfile(fid) returns a struct for given fid
getfid(uid, filename) returns a fid for given uid and filename
fragments(fid) returns a list of fragments for the file specified by fid
add_file(uid, filename, filesize) adds the specified file of the specified size to the Metadata database. The splitting algorithm is used to split the files into several fragments. Those fragments are then assigned to different servers.
reassign_frag(fragmentid) reassigns a given fragment identified by fragid to a different server. This is used when the server associated with the fragment becomes unreachable. In addition, it can be used to "transport" popular fragments from a lousy server to faster servers.

Semaphore Functions:
get_ftp_handle(sid) returns a positive handle id if the all the connections to a server is not exhausted.
surrender_ftp_handle(handle) returns the handle after the connection is terminated so that another connection request can cave in.

Transaction−related Functions:
insert_fragment_transaction(fragid, serverid, transaction_type) inserts a fragment transaction of a given type and marks it as an uncommitted transaction.
abort_fragment_transaction(transaction id) aborts the fragment transaction identified by the

transaction id.
commit_fragment_transaction(transaction id) commits the fragment transaction identified by the transaction id.
insert_file_transaction(fid, transaction_type) inserts a file transaction of a given type and marks it as an uncommitted transaction.
abort_file_transaction(transaction id) aborts the file transaction identified by the transaction id.
commit_file_transaction(transaction id) commits the file transaction identified by the transaction id.

Service Quality−related Functions:
insert_throughput(server id, size, transaction_type, time_taken) is used to insert a transaction to the table 'server_throughput'.
get_throughput(server id) returns the average server throughput for given sid.
slap_server(sid, nslaps) decrements the server rating by nslaps.
pat_server(sid, npats) increments the server rating by npats

5 MFS Algorithms

5.1 File Split Algorithm

The algorithm that splits a file into smaller fragments must take several performance and space metrics into account. The two main criteria are the reliability and data throughput of a remote server. The free space available on a given remote server must also be considered. Based on the aforementioned criteria, we designed a simple algorithm that gives preference to servers with higher reliability rating and data throughput. We set a soft limit for the maximum size of a fragment in order to ensure fair distribution of the fragments of a given file. For files larger than 7.5 MB we chose a split size of 2 MB. For smaller files we believe some performance may be gained by splitting the file into even smaller fragments and we present a detailed analysis for the choice of split sizes in section 7. The algorithm first orders all the available storage servers by their reliability rating and by data throughput. It then allocates fragments to the servers depending on the amount of free space available on each server. For instance, if the space available is larger than the fragment soft limit, the fragment size is either the file size or the soft limit, whichever one is smaller. The pseudocode for the basic splitting algorithm is given in Figure 1.

```
1.    for each server,
        server.quality = (wt1 * server.rating)+(wt2 * server.throughput)
2.    sort servers by quality
3.    sizeleft = filesize
4.    while(sizeleft > 0) {
       for(i = 0; i < number_of_servers; i++) {
         if(!sizeleft)
             break;
         if(!server[i].freespace)
             skip current server;
         if(sizeleft <= server[i].freespace) {
            if(sizeleft < MAXSPLITSIZE) {
              allocate fragment of size <sizeleft> to server[i]
              sizeleft -= sizeleft;
            }
            else {
              allocate fragment of size <MAXSPLITSIZE> to server[i]
```

```
                sizeleft -= MAXSPLITSIZE;
              }
            }
            else {
             if(server[i].freespace < MAXSPLITSIZE) {
              allocate fragment of size <server[i].freespace> to server[i]
              sizeleft -= server[i].freespace;
             }
             else {
              allocate fragment of size <MAXSPLITSIZE> to server[i]
              sizeleft -= MAXSPLITSIZE;
             }
           }
         }
       }
5. return list of (fragment_name, fragment_size, server_id) tuples
```

Figure 1: Basic File Splitting Algorithm

In order to sort the servers, we converts the reliability rating and data throughput for each server into a single final metric using a weighting process. The throughput used is actually the server's average throughput, since the Metadata Module maintains throughput information for all past file transfers corresponding to each server. A brief analysis for choosing the right weights for the rating and throughput are described in the "Performance" section.

5.2 Server Reliability Rating Algorithm: Pick the better server first

To allow for differentiating good servers from bad servers, a simple rating mechanism was implemented. Each server has a rating between 0 and 10. A 0 rating means that the server could not be reached in the past 5 attempts and hence MFS believes that this server may be down or that it was unable to open a socket to that server. On the other end of the reliability spectrum, a rating of 10 means that the MFS system believes that it can connect to the server in the first attempt. The algorithm currently implemented to update the reliability ratings is as follows:

1. After 1 retry attempts to connect or transfer a file to a server, decrement its rating by one.
2. After 5 retry attempts, bring down the rating to 0 or 1 (0 if the previous rating was 1, and 1 for all other previous ratings).
3. If the connection to a server succeeds in the first attempt, increment the rating by 1.
4. The rating cannot be lower than 0 or greater than 10.

Servers with rating 0 may not be used at all by the split algorithm until the free space on all other servers is utilized, and hence their rating may not get updated to reflect the latest conditions. Hence a script may be run periodically that tries to connect to all servers with rating 0 and update their ratings to a higher number if they can reached successfully.

6 Performance

6.1 Determining Fragment Size

In most cases, using a split size that is too small leads to worse performance because of the additional overhead of multiple connects and logins while one that is too large may degrade performance by not exploiting parallelism. For our system to yield a substantial performance benefit, determining the optimal fragment size for file upload is of central importance. To determine this optimal size, we ran an FTP client which opens a maximum of 5 connections to a single host (for the purposes of this test, we discovered that an average server will accept 5 simultaneous connections from one client). We measured the time taken to upload files of 1, 5, and 10 MB to two different servers using this FTP client. One server ("server 10" in Fig. 2) is a "fast" server on the MIT network (jkleff.mit.edu); the other server ("server 4" in Fig. 2) is a "slow" server on the Internet (ftp.50megs.com). For large file transfers, which is the intended domain of our system, the graphs show that while there is not much performance benefit for the "nearby" server, throughput to the other server is maximized in a definite narrow range. Given that the vast majority of file storage servers used in MFS will be of the latter type, there is a clear performance benefit to choosing fragment size approximately equal to 1/5 the total file size. (This is of course due to our assumption of 5 simultaneous connections. For n simultaneous connections, we would expect the optimum fragment size to be roughly 1/n times the file size.)

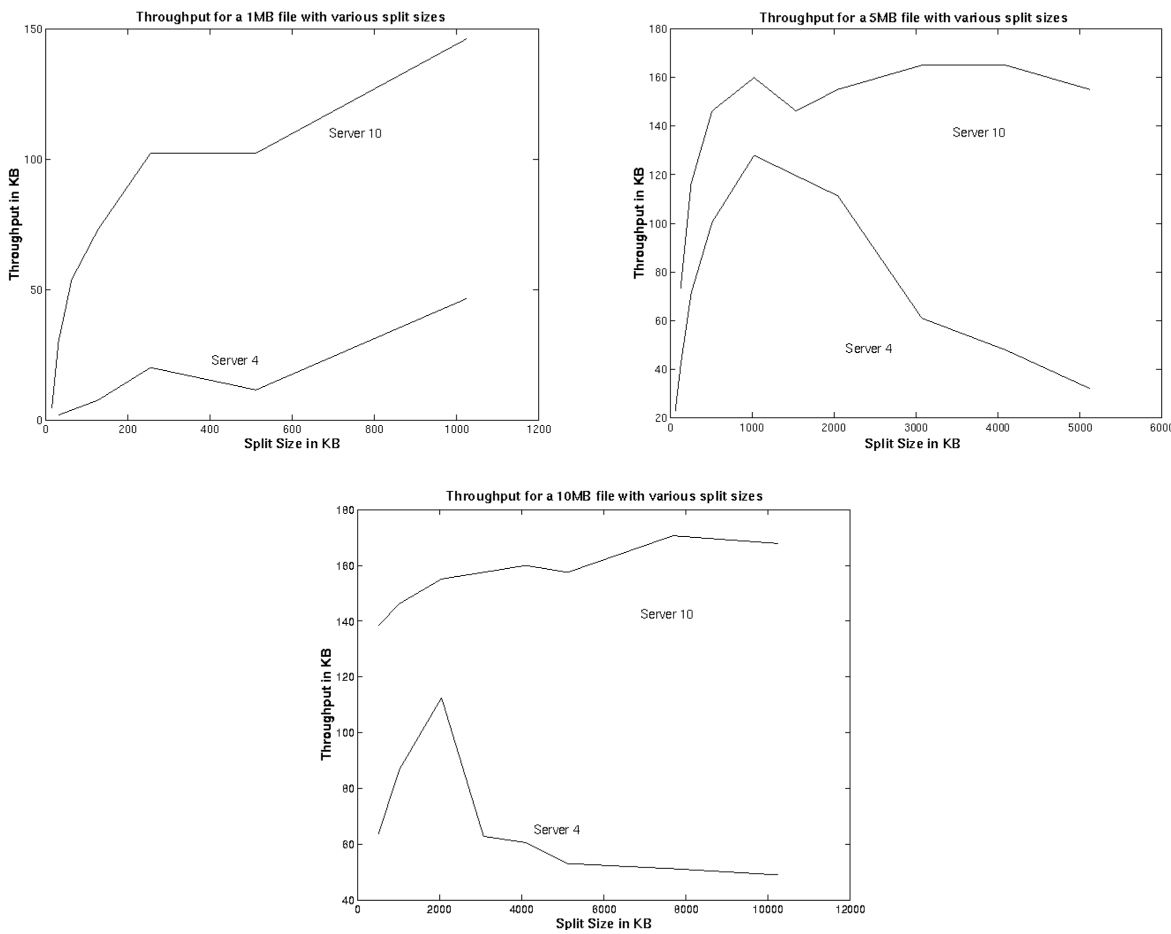

Figure 2: FTP throughput to a single host (max. 5 simultaneous connections)

We measured the upload and download performance of our system against that of a standard UNIX FTP client upload/download to a fast single server ("server 10" above) of files of various sizes. The MFS system used for this evaluation included 6 remote storage servers. We tested two cases of maximum fragment sizes. In the first case, we used a static 2 MB fragment size for our evaluation. For the second case, we chose a fragment size based on the size of the file being uploaded. For files smaller than 2.5 MB, we used a 512 KB split size; for files

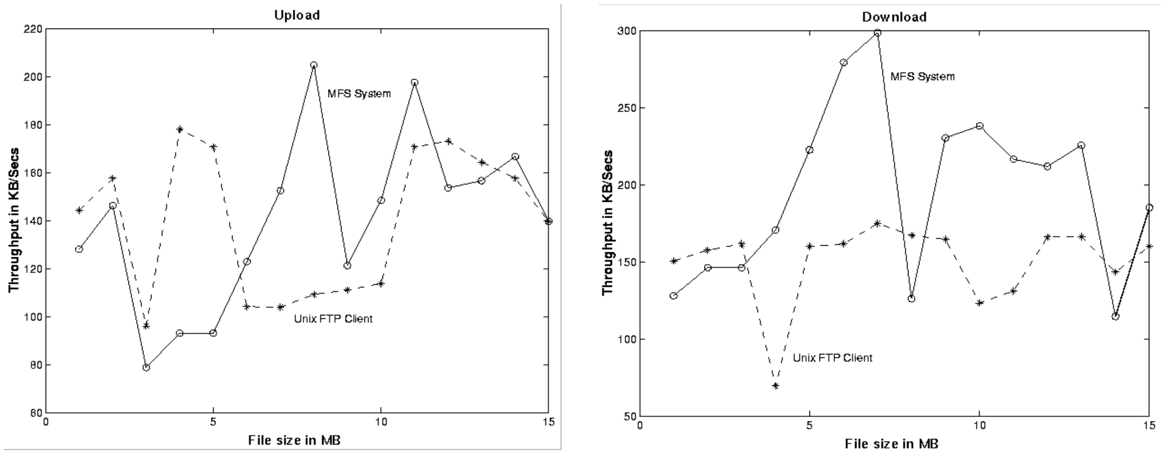

Figure 3: Comparison of standard FTP upload/download to MFS upload/download with fragment size = 2 MB

between 2.5 and 7.5 MB, we used a 1 MB split size; and for files larger than 7.5 MB, we chose a 2 MB split. Results of the tests are shown in Figures 3 and 4. In the static split size case, the upload and download throughputs between the standard UNIX FTP client and MFS are neck–and–neck for files smaller than 3 MB. This is mainly because the MFS client closely resembles the standard client because the MFS client is essentially transferring only one or two fragments. The upload throughput for files of sizes 4–5 MB remained lower than the standard client

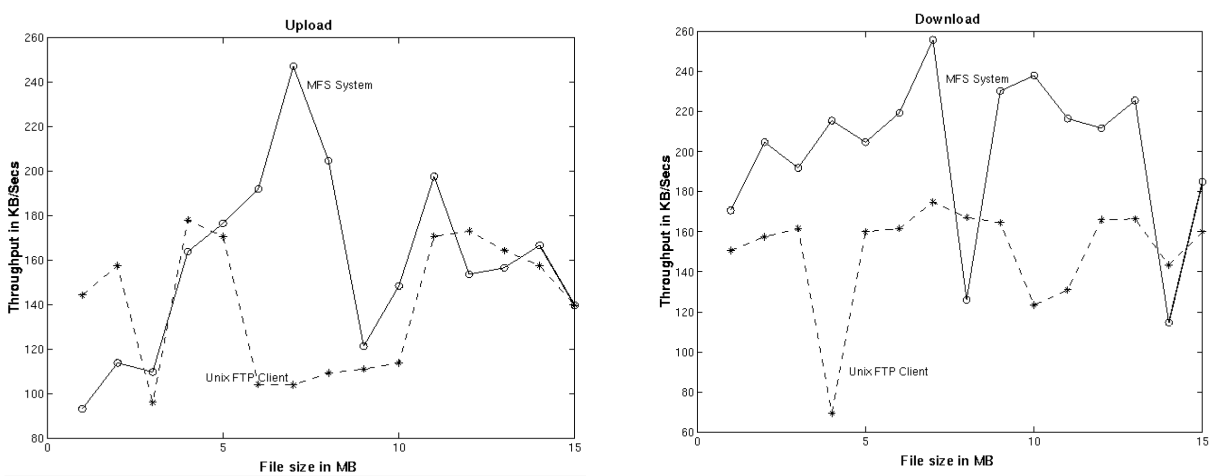

Figure 4: Comparison of standard FTP upload/download to MFS upload/download with dynamic fragment size

because there are still fewer fragments and the overhead of extra FTP logins brings down the performance. As the file size is made larger, the additional overhead is compensated for by the increase in parallel bandwidth, and MFS clearly performs better than the standard client. On a few occasions, the performance dropped below that of the standard client, which is primarily explained by changes in the dynamics (retry attempts) of the various servers being used by MFS.

For the second case, we used the same measurements obtained from the standard UNIX FTP client, and overlaid the results of MFS that used dynamic split sizes as mentioned above. Choosing a split size that is appropriate to a given file size ensured that the parallel bandwidth is maximized and smaller fragments are distributed across a larger number of servers. In both the upload and download throughputs, we observe a noticeable improvement in files smaller than 3 MB in comparison to the first case revealing the performance benefits of MFS.

7 Failure Recovery

7.1 Fragment Upload Failure

When a file is being uploaded, any of the given fragments may not be transferred due to remote server or connection failure. In order to ensure permanent storage of a fragment (and hence the file), an alternative remote server can be requested from the MFS Metadata Module. The file transfer module can then try storing the fragment on the new server. It is possible to repeat this mechanism a couple of times before reporting a general "file transfer" failure. This ensures that several attempts are made to store the fragments of a file, thereby increasing the probability of success.

7.2 Fragment Replication

Although not part of the current MFS implementation, we have designed a mechanism to increase the fault tolerance of the system by adding redundancy of the fragments. An improved version of the split algorithm mentioned earlier can be implemented so that when a file needs to be stored, the algorithm will try to allocate each fragment to several unique servers. However, it should be noted that the gain in redundancy is proportional to the number of storage servers available. Since the replication procedure is done by the Metadata Module, it is transparent to the File Transfer Module which transfers the fragments to the corresponding servers.

When a user requests a file, the file transfer module receives the list of fragment locations from the Metadata Module and tries to get all unique fragments. Since the list also contains redundant fragments, if any fragment download fails, it retries for a few times, and eventually, it will attempt to download the fragment from the next candidate server in the list. This ensures that several attempts are made to restore the fragments of a file, thereby increasing the probability of success. We can request a fragment from all servers storing it and pick the one that arrives first, thereby decreasing the fragment download latency.

7.3 Server Expiration

Various remote storage services have an expiration date, and often, commercial services terminate accounts if they remain idle beyond a certain time limit. In order to safeguard the fragments stored on such servers, the Metadata Module can maintain such expiration information in the servers table. A simple, daily cron job can be run that ensures that such servers are used periodically. In addition, the advantage of fragment replication discussed in section 7.2 provides us a redundant system that will not fail when any single remote service is inaccessible.

8 Related Work

[3] presents a persistent data management layer designed to simplify cluster–based Internet service construction. The system provides a self–managing layer called a distributed data structure which presents a conventional single–site data structure interface, but partitions and replicates the data across a cluster. The system however is geared towards a high–end, closely working cluster of computers and aims to provide a storage system that functionally lies somewhere in between a file system and an RDBMS. Our system is quite different in that it aims to provide transparent access to the free space on heterogeneous storage systems (FTP, database, SCP, NFS, local disks) located anywhere on the Internet by simulating a simple filesystem–like interface. MFS does not provide an application layer data structure, but rather it is an end service by itself. Our system has some similarities to Publius [4], which allows anonymous publishing of documents documents on the Internet. The publisher takes the key, K, that is used to encrypt the file and splits it into n shares, such that any k of them can reproduce the original K, but k−1 give no hints as to the key. Each server receives the encrypted Publius content and one of the shares. Our system, on the other hand, does not aim to provide anonymity but does provide security. A file can be encrypted before sending to MFS in such a way that when the file is split, individual fragments will not reveal any useful information (if, for example, the encryption scheme includes cipher–block chaining). Thus if an encrypted file is stored using MFS, the contents will remain unknown to servers.

9 Future Work

While building our prototype system, we encountered several real–life problems which led us to modify our design to incorporate remote service failures. Servers which we believed were really reliable often cannot be reached and the dynamics of the Internet can often lead to slower or faster connections. We were unable to conduct extensive benchmarking of the system, but it was sufficient to design a mechanism to order servers based on reliability in addition to throughput, and include retry attempts and the ability to reassign fragments to alternative servers.

Another aspect that we were unable to attend to in the prototype was adding security. However, it will not be too difficult to add a function in the client such that it encrypts the file before sending it to the MFS server. Some research might be required to ensure that the type of encryption used is such that individual fragments cannot be decoded.

10 Conclusion

The main idea behind the design of MFS is to enable transparent filesystem–like access to free space on disjoint remote services. This approach eliminates the need to manually split large files and transfer them to various storage servers. Further, MFS is much more efficient by making use of asynchronous I/O and effectively transferring files in parallel. We found that, for files sufficiently large, our system has allowed storage and retrieval with ease and greater speed than manual upload and download. The MFS prototype demonstrates the viability of building a system that is scalable to a large number of multiple, heterogeneous remote storage services, and its initial performance results illustrate the potential of this approach.

References

[1] David Mazieres, Michael Kaminsky, M. Frans Kaashoek, Emmet Witchel. Separating key management from file system security. *Operating Systems Review* 34(5):124−139, Dec 1999.

[2] B. Callaghan, B. Pawlowski, P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[3] Steven Gribble, Eric Brewer, J. Hellerstein, D. Culler. Scalable, Distributed Data Structures for Internet Service Construction, The University of California at Berkeley (OSDI 2000).

[4] M. Waldman, A. Rubin, L. Cranor. Publius: A robust, tamper−evident, censorship−resistant web publishing system. *http://publius.cdt.org/publius.html*, (Hyperlink visited December 5, 2000)