Design and Implementation of STFS: SFS/Thor File System

Sidney Chang and Yan Zhang
MIT Laboratory for Computer Science
6.894 Distributed Operating System Engineering
{sidchang,xyz}@lcs.mit.edu
December, 2000

Abstract

File systems and databases each have their own benefits and particular usage. File systems are generally appropriate for per−file operations on local disks whereas databases are more appropriate for operations that touch several different pieces of data where transactions and ACID semantics are important. The result of combining a file system and a database is a system that is able to leverage the benefits of both systems into a single system where the interface is familiar yet hides the complexity and many advantages of using a database.

We investigated the implementation of a file system in a database. Specifically we combined SFS, a secure global file system, with Thor, a persistent object−oriented database in order to provide access to files stored as objects in a database via NFS v.3 protocol. This combined system which we call STFS (for SFS/Thor File System) provides the transactional semantics of Thor to SFS clients while exhibiting comparible performance to SFS over ext2 and offering an easier implementation to programmers over kernel−based file systems.

1. Introduction

File systems and databases are often considered two distinct systems. File systems are commonly implemented in the kernel and are used to manage data on a local disk. Databases are commonly considered separate from file systems and usually have their own storage management. Operations on files in a file system are on a per−file basis whereas operations in a database can touch many different objects in the database and are wrapped inside transactions.

Even though file systems have a simple and widely used interface, they do not in general provide any means for transactional semantics that are provided by database systems, although this may be desirable in certain cases. For example, if the machine crashes during a "make" process, the programmer may not want to have her source or build tree change in any ways that leave some files in a "middle" state. In addition, a database system provides concurrency control, crash recovery and data replication, among other nice features.

A natural solution is to combine a file system with a database system as its back end and thus reap the benefits of both worlds. However, the overhead associated with database operations seems to be a major concern when people consider this idea. Some work has been done in this area (see related work in Section 6) but we have seen very few performance analyses.

With the advancements in database technologies in recent years, optimizations of various aspects of object−oriented databases have led to much better performance [Thor99]. As a result, the performance argument against using a database system with a file system may no longer hold. Our project serves as an evaluation of this argument.

This project combines the Self−certifying File System (SFS), with a database, Thor, into one system, SFS/Thor File System (STFS). In addition to providing access to files stored as Thor objects via SFS, we also provide serializability, atomicity and other database semantics to file system objects.

The main challenge of this project is designing a sensible interface to objects stored in an object−oriented database and exposing the transactional semantics of a database to the file system.

Through this project, we demonstrate that building a file system on top of a database is easier to implement than in the kernel and that the performance of our system is comparable to that of SFS.

2. Background

Several different systems come together to form STFS − TFS, Thor, and STFS. The following sections describe each of the systems and their usage in this project.

2.1 TFS

Previous work by Salvatore Valente implemented the Thor File System (TFS) [TFS], an NFS v.2 server on top of Thor. While this work is very similar to what we have done with STFS, there are several issues that we wished to address that were not addressed in TFS:

- The implementation of Thor classes for TFS was tailored for NFS v.2. In order to support NFS v.3, we implemented new Thor classes and data structures. TFS is also incomplete and inefficient in some aspects. For example, symbolic link is not supported, Thor's object–orientedness is not taken advantage of, and directories are implemented by arrays, making file lookup linear. We believe that our implementation has improved upon these issues (see Section 4.1).
- TFS was implemented using an older version of Thor where objects are defined in the Theta programming language. Our implementation uses a version of Thor where objects are defined in Java.
- TFS supports three modes of commit: *Standard NFS mode* (where each request is followed by a commit), *Optimized NFS mode* (where batches of requests are committed at once) and *Transactioanl mode* (where clients explicitly indicate commit or abort). However, it does not allow clients to switch among the different modes. In addition, transactional mode for the file system uses an awkward interface – users write to a "commander" file to indicate the desire to commit. We believe that we have provided a better model (Section 3).

2.2 Thor

Thor has a client–server architecture where servers are object repositories or OR's and clients are made up of front ends or FE's and application code. The database schema code is written in Java for the version of Thor we are using. They define the persistent objects that are stored in the database. This code is converted to C by the FE compiler and built along with the rest of the FE code. The application code, written in C++, may use this generated C code.

FE's in Thor cache objects from the OR and has a transaction manager to manage committing of transactions at the OR. It uses a hybrid adaptive caching scheme (HAC) to optimize performance.

The OR maintains a root object for all objects in Thor much like the root of a file system. All objects reachable from the root object are considered persistent. An FE may commit objects not reachable from the root, however they will eventually be garbage collected at the OR.

Application code operates on cached objects inside transactions. The applications can then request the FE to commit transactions and the OR will either commit or abort the transaction. An abort will occur if the transaction includes any operations on an object which has been modified by another client since the time the object was cached. If a transaction is aborted, all modifications caused by the aborted transaction will rollback and the state at the sfsusrv/FE will be as if the transaction had not happened. The application can be notified of an abort. Other handling of an abort, such as a retry of the failed operations, is left entirely up to the application.

2.3 SFS

SFS is a distributed file system that provides secure global access to files. SFS clients connect to SFS servers and make remote procedure calls (RPC's) to invoke operations on files. Clients can also cache data. This project makes use of a user–level SFS server which processes RPC calls from clients using the NFS v.3 protocol. RPC's are processed by making calls to an underlying file system which could be ext2, NFS, or the Thor database!

3. Design

The design of our system, STFS, was guided by the following goals:

- Incorporate transactional semantics of a database seamlessly into a file system. This is achieved by merging SFS and Thor FE.
- Provide a clean abstraction of a file system. This is provided by our design and implementation of the file system schema.
- Minimize the overhead by reducing the layers of indirection in our system. This is reflected in our system architecture.

The following sections describe the details of the design of STFS beginning with an overview of the system architecture, followed by discussion of the file system schema design in the database, and ending with a discussion of the front end of STFS which provides the client interface to the file system.

3.1 System Overview

STFS is organized into a client/server architecture (see Figure 1) where clients are unmodified SFS clients and servers are a combination of a modified SFS user–level server (*sfsusrv*), Thor front end (FE), and Thor object repository (OR).

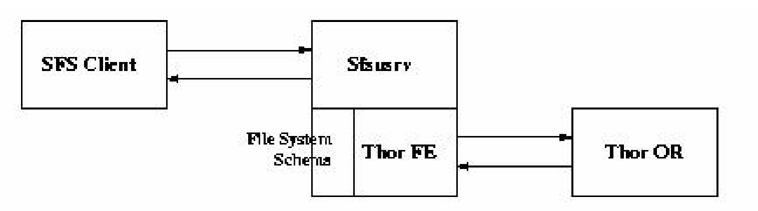

Figure 1: STFS System Architecture

SFS clients make file systems calls using NFS v.3 to the combined user–level SFS server and Thor front end (sfsusrv/FE). The sfsusrv/FE processes the procedure calls by invoking methods on Thor objects. Sfsusrv/FE wraps operations on Thor objects within transactions and commit them periodically to the OR. STFS provides flexible commit semantics to suit the needs of the user. In particular, the user can set the *mode* of operation by specifying the number of operations between commits(Section 3.4). Based upon the STFS's current mode of operation and the procedure call, it decides when to commit a transaction. (if it is COMMIT or stable WRITE, STFS must still commit immediately to preserve the NFS semantics).

To commit a transaction, sfsusrv/FE sends a commit request to the Thor OR. The OR will then either respond with success or an abort. Currently our design does not provide any means for notifying the client of an abort. Notifying a client of an abort when the commit request was caused by a stable WRITE or COMMIT request from the SFS client falls naturally into the NFS v.3 semantics by returning the correct error codes. However, it is not entirely obvious how a client can be notified of an abort on the periodic commits that the Thor FE can be set to do. To accommodate this, modifications to the protocol and the SFS client are necessary.

An alternative architecture that we considered is to separate sfsusrv and the Thor FE and have them communicate via a TCP connection. This would have the advantage of requiring minimal changes to sfsusrv and also bypassing the headache of merging the sfsusrv and Thor FE code bases. The disadvantage of this approach is that it woud both add yet another layer of indirection to processing a client request and duplicates marshaling and unmarshaling code. For this reason, we chose our current design.

3.2 File System Schema

We implemented a file system based on NFS v.3 protocol. NFS v.3 was chosen for its ease of implementation and also because sfsusrv has an NFS v.3 interface. A more important reason for our choice over NFS v.2 [NFS2] is NFS v.3's inclusion of the COMMIT procedure, which acts as a natural commit point in our transactional semantics. The file system schema described in this section implements the NFS v.3 protocol.

To represent a file system in an object–oriented database we define schema in the form of object–oriented classes where the persistent data in the database are instances of these classes and queries on data in the database are translated into method invocations on the persistent objects. Sfsusrv/FE processes SFS client requests by invoking methods on instances of file system classes. Figure 2 shows the main file system class hierarchy.

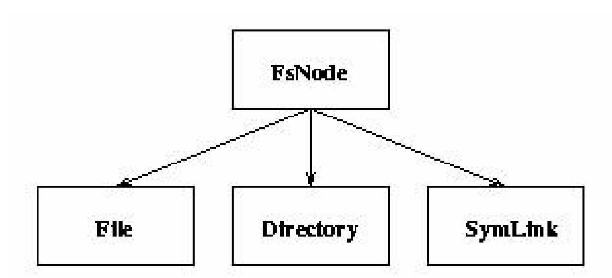

Figure 2: File system schema class hierarchy

There are mainly three different types of file system objects: file, directory and symbolic link. We repressent them as three classes: *File*, *Directory* and *SymLink*. All of these classes directly inherit from a base class *FsNode*, which contains attributes shared by the three classes. This structure mirrors the inode structure in a real file system. Below, we give a description of each of these classes:

FsNode FsNode maintains attributes for the node analogous to the fattr3 data structure in the NFS v.3 protocol. It also has methods for setting the attributes and checking access permissions.

File File has methods for reading and writing data in the file and for changing the size of the file (this is required by the SETATTR server procedure in NFS v.3). The File class stores file data in a variable number of fixed−sized blocks (Section 4.1).

Directory Directory has methods for looking up, creating and deleting file system objects. It also has methods corresponding to other NFS v.3 procedure calls, such as RENAME, READDIR, etc.

SymLink SymLink simply maintains a path name of the symbolic link.

Furthermore, we defined a *FileSystem* class to act as a container of all the above file system objects and also as a dispatcher of procedure calls. For each RPC call that sfsusrv/FE processes, the FileSystem class will be queried with an analogous method (Section 3.3). The difference between the FileSystem class and the original implementation of the RPC handler in sfsusrv (the *client* class) is that the FileSystem class makes calls on Thor objects as opposed to system calls to the machine's underlying file system. In addition to keeping all the file system objects created and dispatching methods on them, the FileSystem class is also responsible for creating unique file handles.

Rather than providing an interface to the file system through the FileSystem class, sfsusrv/FE could have directly manipulated the other file system classes such as File and Directory and filled in its own reply blocks. However, the use of the FileSystem class minimizes the number of points at which the sfsusrv/FE module needs to query the database and provides better abstraction. Minimizing the number of queries to the database increases the performance since the overhead of method dispatching from non−schema code to the schema code is higher than that associated with method dispatching from within.

Additional classes were defined to represent the replies of NFS v.3 procedure calls, file attributes, etc. Most of them mirror the XDR data structures defined in [NFS3].

3.3 SFS user−level server and Thor FE

Thor FE is merged with the SFS user−level server by adding a new option to the FE to run the modified sfsusrv at the start up of the FE. FE is still available to run other applications besides sfsusrv.

At the design level, the main change that we need to make to SFS user−level server is to translate RPC calls from the client into method invocations on Thor objects rather than system calls as before. Note that not all procedure calls have their corresponding methods in FileSystem class. COMMIT and stable WRITE are such examples. They are translated directly into a commit call to the OR. Overall, we try to minimize the changes to sfsusrv.

3.4 Modes of Operation

To deal with the notion of database transactions when processing requests from the SFS user−level server, we make use of NFS v.3 protocol semantics. Unlike TFS, STFS has a more natural interface for a client to request commit operations since it is based on NFS v.3 protocol and not NFS v.2 protocol. NFS v.3 protocol specifies a COMMIT operation on a specified file. This operation requests that the server ensure that any changes made to the file are committed to stable storage. In addition a stable WRITE request in NFS protocol specifies that the write be written to stable storage before returning to the client. Therefore COMMIT and stable WRITE requests are natural commit points for the sfsusrv/FE to commit at the Thor OR.

Additionally, the Thor FE periodically commits transactions every n (>= 0) operations and n can be configured by the user. The higher n, the fewer commits per set of transactions. However, lowering n will incur performance costs since a commit operation is expensive. When n = 0, STFS runs in a purely transactional mode which means that commits will only happen when the user explicitly requests it with a stable WRITE or COMMIT. A user could then use these points to wrap her operations inside a transaction.

It is also possible to support dynamically switching between different values of n while using STFS by providing a special file name that the user can write to with the value of n. STFS can then intercept this WRITE call, get the value of n and change the commit count.

4. Implementation

The implementation of STFS consists of two main components:

1) representing file system objects as Thor objects
2) integrating the SFS user−level server with the Thor FE

In the following sections, we discuss the implementation issues in each of these components.

4.1 Representing File System Objects

As mentioned in Section 3, we implemented the various types of file system objects in classes in Java. These classes are then compiled into C code (as described in Section 2.2), built with the Thor FE and used by application programs written in C++. We chose to implement our file system classes in Java as opposed to Theta, another object−oriented programming language supported by Thor. This is because currently we have more support for Java Thor and also because the Theta version does not correctly implement inheritance.

The rest of this section will go into more detail on the implementation of file system objects and will be followed by a discussion of the limitations imposed on STFS by the Java implementation for Thor.

File Handles In NFS v.3 protocol, file handles can be variable in length. However, to simplify the implementation, we implemented file handles as 64−bit long integers generated by the FileSystem class. The *fileid* attribute of a file system object is set to be the same value as its file handle. The FileSystem class guarantees the uniqueness of the file handle by maintaining a monotonically increasing counter which is incremented each time to get a new file handle. We are able to use such small file id's since the complexity of identifying a file is simpler in our object−oriented database. In a traditional file system, the large file handle is often used to store information that uniquely identifies the series of inodes that finally lead to the file's location in the file system.

When a file is deleted from the file system, its file handle is never reused. This implementation is not very efficient and limits the amount of objects that can be stored in the file system if we perform a lot of creations and deletions. Although 2^{64} is probably a large enough namespace for most file systems.

FileSystem Class Besides creating new file handles, the FileSystem class also maintains a hashtable mapping file handles to existing file system objects. This table is used for quick lookup of objects based on a file handle given by the SFS client. New entries are added to the hashtable whenever a new file system object is created and removed when it is deleted from the file system. We chose a hashtable implementation mainly because it has good performance when the number of entries is large, giving a constant lookup, insertion and deletion time. When a FileSystem object is first created, it contains an empty root directory with file handle equal to 0. Finally, the FileSystem class has dispatch methods for most of the NFS v.3 procedure calls.

FsNode Class In addition to file attributes, FsNode also contains a pointer to the FileSystem object that created it so it can call methods on the FileSystem object to insert and delete objects.

Directory Class A Directory object maintains a hashtable mapping the names of entries (as a String) in that directory to their corresponding file system objects. In addition to fields and methods inherited from FsNode, a Directory object also contains a pointer to its containing parent directory. This pointer points to itself for the root directory. When a Directory object is first created, it already contains two entries: "." and "..", corresponding to the current and parent directories.

A Directory is responsible for implementing a number of NFS v.3 calls including LOOKUP, CREATE, LINK, etc. Among these, the implementation of READDIR is worth pointing out. First of all, we use the *mtime* of the object as the *cookieverf* to guarantee the freshness of the *cookie*. Secondly, we generate a Vector of all entries in the directory from the hashtable on the fly whenever one requests a READDIR for the first time (as identified by a 0 *cookie* and 0 *cookieverf*). Finally, to simplify the implementation, we only return one entry at a time, ignoring the *count* argument of the READDIR request. This is the correct implementation as long as *count* is greater than the size of the entry returned, which is the common case.

File Class Files are implemented as vectors of fixed sized data blocks. Each block is a byte array of 1024 elements. Vectors are used so that files can grow easily. Fixed sized data blocks are used to quickly locate the position for read and write in a file. Another reason for using Vector of fixed sized data blocks as opposed to one single String object is due to limitations in Thor discussed below.

Limitations of Thor In Thor, an object size can not be greater than the size of a page, which is usually 8192 bytes. This imposes a constraint on some of the data structures used by the file system classes. For example, as mentioned, the maximum file size would be about the size of a page if we chose a String representation of the data in the file since a String object cannot be bigger than a page.

Even so, the size of a file is still limited. The vector that holds a file's blocks can not grow larger than a page. If a page is 8192 bytes and the vector's elements are 4 byte references to the blocks, a file object is only able to house references to roughly 2048 blocks. Therefore with 1024 byte blocks, the size of a file is limited to about 2MB.

On the other hand, the limitation mentioned above can be overcome by changing the internal representation of the data. For example, instead of using an array structure, a tree structure can be used to store references to other objects thus not consuming a contiguous page for that single object. This limitation is purely implementational and is orthogonal to the use of an object−oriented database.

4.2 Integrating SFS User−level Server and Thor FE

Integration of the SFS user−level server with the Thor FE consists of several parts: modifying SFS user−level NFS v.3 procedure calls, handling mount protocol, and modifying Thor FE. The rest of this section gives the details of implementing these parts.

Modifying NFS v.3 Procedure Calls We took the code for sfsusrv and modified it to translate the NFS v.3 procedure calls to method calls on Thor objects rather than a series of system calls to the machine's file system in the kernel.

Mounting the File System When the Thor FE starts, it initializes the Thor root or retrieves an existing persistent Thor root. Using this root, different applications can reach their own roots. Sfsusrv gets its root (a FileSystem object) in the same way. We still need to give some form of reply to mount requests from the client.

The unmodified SFS user−level server returns a file handle of the exported directory by making system calls. Similarly in our system, we can query our database with the requested path and call the FileSystem object to retrieve the file handle of the exported directory. However for the current implementation, we simply return the root file handle, which is always 0.

Modifying Thor FE The Thor FE is modified to have an option to run SFS, taking arguments for it (such as the configuration file) from the command line. It then gives control over to the modified sfsusrv code passing the root handle of Thor.

5. Evaluation

For evaluation, we looked at the performance and programming implications of our system.

5.1 Performance

The performance of STFS was compared to the SFS user−level server running on Linux with the ext2 file system We would have liked to also compare the performance of our system on other operating systems such as OpenBSD or OSF1 and with other file systems such as FFS but could not due to time constraints.

5.1.1 Experimental Setup

Our experiment involves a single client/server pair. The client machine is a 200 Mhz Pentium Pro running OpenBSD 2.8. The server machine is a 700 Mhz Pentium III running Linux Redhat distribution 6.2 (with the NFS v.3 kernel patch), with ext2 as its file system. They are connected by a 100Mbit/sec Ethernet. The client runs SFS client software and the server runs the merged SFS user−level server and the Thor FE along with the Thor OR.

To evaluate the performance, we ran a modified Andrew Benchmark over TCP connection between the client and server using NFS v.3. We compare

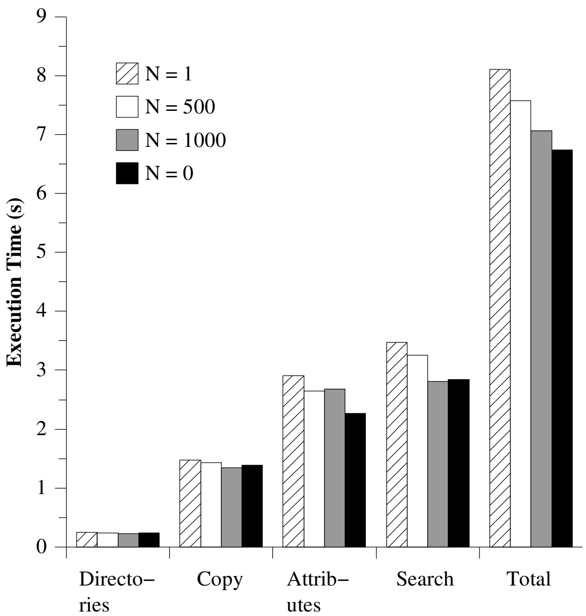

Figure 3: Andrew Benchmark for STFS with different values of n

	Directories	Copy	Attributes	Search	Total
N = 1	0.247	1.477	2.909	3.473	8.106
N = 500	0.239	1.427	2.646	3.261	7.573
N = 1000	0.231	1.350	2.676	2.807	7.064
N = 0	0.240	1.392	2.268	2.844	6.744
sfsusrv/ext2	0.195	1.442	2.020	2.551	6.208

Table 1: Data for Figure 3 & 4

the results of the benchmark when the server was STFS and when it was the unmodified SFS user–level server. The Andrew benchmark consists of five different phases:

1) Phase I: many subdirectories are recursively created.
2) Phase II: stresses the file system's ability to transfer large amounts of data by copying files.
3) Phase III: recursively examines the status of every file in fscript, without actually examining the data in the files.
4) Phase IV: examines every byte of every file in fscript.
5) Phase V: is computationally intensive

Our modified Andrew benchmark removes phase 5 because of compilation errors inherent to the benchmark. It is also modified to contain 41 as opposed to the original 70 files.

5.1.2 Performance Results

The performance of STFS was not expected to outperform that of sfsusrv because of the overhead of object creation and commit for Thor. For example, the FE of Thor communicates with the OR via TCP connection.

As mentioned in Section 3.4, we expect higher values of n to give better performance, where n is the average number of operations between commits. To simulate transactional semantics, we have the server commit only when the client sends a COMMIT request or *stable* WRITE (n = 0). We compare it with the cases where commit happens after every operation (n = 1), after approximately 500 operations (n = 500), and after 1000 operations. Note that we adhere to the NFS v.3 semantics even for n > 0 in that we still commit in all cases where we would in the n = 0 case. Figure 3. shows the results.

As indicated by Figure 3, n = 0 gives the best overall performance in terms of total execution time. On the other hand, n = 1 gives the worst performance in all cases. In between, performance when n = 1000 is better than n = 500. In particular, when n = 1000, the performance is approximately the same as n = 0. Discrepancies might come from our statistically insignificant sampling.

Next, we compared the performance of sfsusrv with that of n = 0. The result is shown in Figure 4. As expected, sfsusrv outperforms STFS in almost all phases. The worst case is the first phase, where STFS has a 23.1% degradation over sfsusrv. This is probably due to the inefficient way Thor initializes arrays, which is being used in our Directory data structure. Rather than initializing all entries to 0 (e.g., using memcpy), Thor writes 1's (Thor's representation of NULL in Java) into each entry individually to guarantee Java semantics. We could potentially speed up the initialization of arrays and hence the performance of this phase by memcpying 1's to arrays. In Phase II, STFS actually performs 3.4% better than sfsusrv. In this phase, STFS and sfsusrv commit approximately the same number of times, however, a STFS commit only involves a write to the log at the OR, whereas a sfsusrv commit requires writing to disk, which is longer. Therefore, even with network overhead, STFS outperforms sfsusrv for copying files. Reading of metadata and data of the files is slower for STFS than sfsusrv. In phase III and IV, STFS takes 12.2% and 11.5% longer time than sfsusrv.

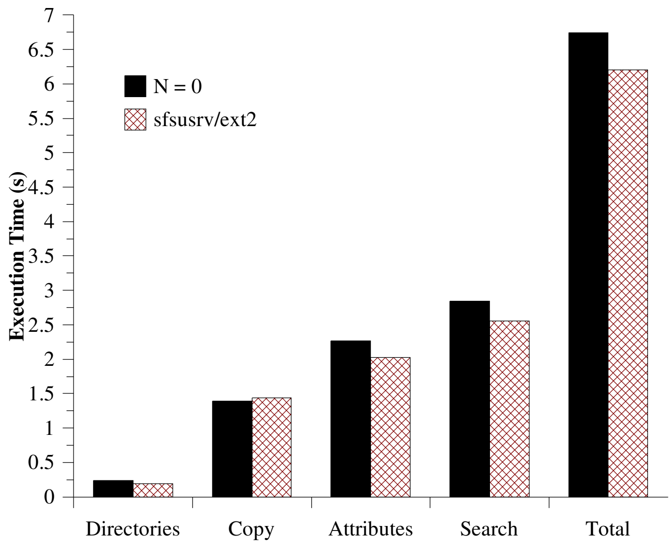

Figure 4: Andrew Benchmark comparison of STFS (n = 0) and sfsusrv/ext2

This can be explained by the network overhead in fetching data from the OR. This might be improved by increasing the cache size of the FE.

Overall, the performance of STFS is not as bad as we expected and there are places where we think further optimizations could be made.

5.2 Programming Implications

Implementing a file system in an object–oriented database is much simpler and straightforward than hacking the kernel. This in turn eases the task of debugging and maintaining the code. Changing the implementation of the file system is also easier when it is object–oriented. For example, we can change the lookup function for directories or the representation of a file.

Code size also decreases. Table 2 shows the line count for different components we are considering. We can see that the ext2 implementation and sfsusrv code together has a code size 6 times greater than that of the STFS user–level and schema code combined. While one could argue that we should also take into consideration the size of the Thor database when comparing the two, it could also be argued that since the database is a general tool, it is not solely used by the file system and can be used by a much broader range of applications such as a calendar.

	Codesize
Stfsclient + stfsusrv	1928
File System Schema	1956
FE – stfs	26308
OR	15793
Ext2	24076
Sfsusrv	1415

Table 2: Code sizes for STFS and ext2

6. Related Work

Combining file systems with database systems is not a new idea. Below, we give an overview of two systems most relevant to our work and compare them with ours.

OdeFs OdeFs[OdeFs94] is a file–like interface to Ode, an object–oriented database based on the C++ object model. Database objects are accessed and manipulated like files in a traditional file system using the standard Unix commands. Like our system, OdeFs is implemented as a user–level process – a network file server, but it uses the NFS v.2 protocol instead of NFS v.3. Unlike our system, OdeFs is built on top of a conventional file system: directories and files correspond to actual Unix directories and files in the underlying magnetic file server. Operations on directories and files are simply forwarded to this file server which does the actual work and other requests are satisfied by the Ode DB servers. In our system, we define our own directory and file data structures and treat them like any other objects in the database. In OdeFs, all objects in the database have to be modified to implement a file system interface, whereas in our system, no existing database objects need to change. In a sense, our system and OdeFs chose the opposite tradeoffs: our system does not support existing file system tools, such as disk partitioning, formatting, etc; while at the same time, we do not need to modify the existing objects in our database system. The opposite is true for OdeFs.

SHORE SHORE (Scalable Heterogeneous Object Repository) [Shore94] integrates file system and OODB concepts and services. On one hand, it provides Unix–like file system model including a tree–like object naming space and support for legacy Unix file–based tools. On the other hand, it provides OODB–like services. It shares similarities with Thor in some aspects, such as typed object store and two–phase commit protocol. There are also differences between the two OODBs; below, we list a few obvious ones:

- Objects remain inside Thor, which is critical for safe sharing. This is different from SHORE which does not guarantee type–safety.
- Thor uses hybrid adaptive client caching scheme, which reaps the benefits of both object and page caching; Shore does only page caching.
- Thor uses optimistic concurrency control which is shown [OCC95] to outperform that of adaptive callback locking, used in SHORE.
- SHORE uses a symmetric peer–to–peer architecture rather than the client–server architecture used by Thor to avoid the extra layer of indirection between a client and server helping to achieve scalability.

The primary goal of SHORE is to provide "holdout" applications to eventually move data out of files and into persistent object repositories. This is quite different from the goal of our project, which is to experiments with adding transactional semantics into file systems and evaluate the performance, without worrying about existing Unix files or database objects, Thor provides a lighter–weight, heavily optimized OODB which fits our purpose.

However, neither Shore nor OdeFs offered any performance evaluation, which was one of the goals of our project.

8. Conclusion

In this project we have merged two distinct systems – SFS, a secure global file system, and Thor, a persistent object–oriented database. By merging these two systems we were able to provide database transactional semantics to a file system fairly seamlessly via NFS v.3 protocol. Our results showed that STFS performs comparably to SFS over ext2 and that implementation of a file system in a database can be much simpler than in a kernel.

Acknowledgments

Thanks to many people for helping with getting things to run! Chuck – for helping us get SFS on Linux to build and most of all for helping us get past nasty link errors when merging SFS into Thor. Chandra – for helping us with Thor and for all the benchmarking advice. Jinyang and Frank for helping with SFS and random–beatings!

References

[NFS3] Callaghan, B, et al. "NFS Version 3 Protocol Specification", RFC 1813, June, 1995.

[NFS2] Sun Microsystems, Inc. "Network Filesystem Specification", RFC 1094.

[OCC95] Adya, A, et al. "Efficient optimistic concurrency control using loosely synchronized clocks", ACM SIGMOD, 1995.

[OdeFS94] N.GeHani, et al. "OdeFs: A file system interface to an object–oriented database", 20th VLDB, 1994.

[SFS99] Mazieres, D, et al. "Separating key management from file system security", SOSP, 1999.

[Shore94] Carey, M.J., et al. "Shoring up persistent applications", ACM SIGMOD, 1994.

[TFS] Valente, S. "The Thor Filesystem", 1998.

[Thor99] Liskov, B, et al. "Providing persistent objects in distributed systems", ECOOP, 1999.