StashFS : Generalized Disconnected Operation

Robert Mustacchi
Computer Science Department
Brown University
rm@cs.brown.edu

Abstract

With the rise of network file systems and cloud-based data storage an increasingly common problem is being able to access data if servers or communication channels fail. In this paper we introduce StashFS, a generalized method for disconnected file system operation and access across any two file systems while restricting software modifications only to the client. With StashFS, clients can disconnect and make changes to locally cached copies of files. They can detect conflicts and merge in their changes upon reconnection. In addition, we measure the performance and overhead of the system and conclude that the overhead while using it is acceptable.

1 Introduction

As distributed and cloud services become more commonplace, the expectations of users for easy integration and data synchronization become much higher. Thus suggesting to a user to do something such as reformat a hard drive or install a new file system to attain the benefits of these systems is inviting a nightmare onto those users. Similarly, those users do not have the ability to install additional software or add features to these remote services, as they lack administrative privileges.

Users need to be able to also use these systems to extend their storage capabilities beyond that of their initial device and be able to keep this data synchronized across all of their devices automatically, without much effort. Manually running synchronizations or scheduling them regularly often leads to accidentally missing one.

Additionally, if the user wants to start using these services, they should be able to simply start the synchronization process and have their data migrated into their remote systems and be able to continue using their devices and data as normal.

With these ideas in mind, StashFS was developed.

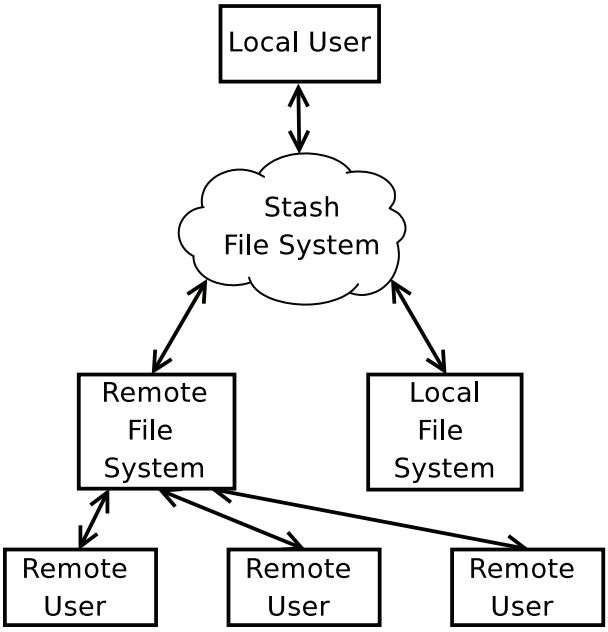

Figure 1: System Overview

StashFS is designed to live on top of an already existing local file system and utilize it for local storage. As long as the cloud or distributed service can be mounted as a file system, StashFS can use that file system as a backing store and provide both systems continuous synchronization. An example of this architecture is shown in Figure 1. StashFS also provides the ability to go into disconnected mode where it modifies local copies of their files. It can later reconnect and resume synchronizing and push the changes to the backing store.

1.1 Benefits

At a high level, here are several advantages of using StashFS:

1. Backup data into another remote file system

2. Temporarily clone data from a remote system

3. Create a copy of data from a remote source that stays synchronized

One important requirement is that the user should be able to do all of these without having to reformat their file system and preferably without installing anything specific to the underlying synchronization system.

As a result we gain several features:

1. Regular synchronization of data

2. Ability to temporarily or permanently retrieve data out of the cloud or any other remote source

3. Ability to use data from a remote source locally while offline and have it synchronize on reconnect

4. Ability to modify our data from another location and have it merge changes into the local version

1.2 Differences from SCM

There are several similarities with this system to a source control management (SCM) systems such as RCS, SVN, git, and mercurial[14, 29, 6, 15]. These similarities mainly lie in the ability of the system to determine that something has changed and attempting to help merge conflicts. However, this system is different in that it doesn't have any notion of a history. There is no way to revert files to a prior version. Building a file system interface based on an existing source control management system, similar to [18], would be an interesting area to pursue and explore.

Despite the seeming advantages of using source control management systems, there are several areas where this system will perform better. By default, with an SCM, one uses substantially more space. Not only is there a copy of each object and its modification history hidden in the repository, but there is also a copy of the object in the normal file system. This means that at least twice the space of the original object will be used. While normally this is not a significant problem as these repositories are small, if the entire file system was in an SCM, this would mean that the effective space had been cut in half.

Furthermore, SCMs function entirely on user input. A user has to run commands like `commit`, `update`, or `add`. This is not a process which would work well for general users, especially considering that experienced programmers sometimes forget to add every file they create to the repository before they commit.

2 Related Work

There has been a lot of work done in the area of distributed file systems that support disconnected operation. There also has been work done in caching file systems which serves to maintain local copies of data that resides on slow or remote media.

Disconnected operation refers to the ability to have a copy of the working set of data on a local machine and being able to modify it without having to be connected to the network. Then, upon reconnecting, there will be data synchronization and conflict management.

The Andrew File System (AFS), strictly speaking, is not a disconnected file system, however it does serve as a basis for Coda and was also one of the early distributed file systems to include local file caching. One important aspect of AFS is that rather than always fetching the required data from the server, it attempts to serve the data from the client's local cache. This could be done because of how locking occurs in AFS. By default, an exclusive lock is granted to a particular user for a particular file. When an exclusive lock needed to be revoked, AFS uses a callback to remove that exclusive lock and synchronize changes to the file, after which the file system would need to renegotiate access to the file[11]. Similarly, when a file is closed by the user, AFS then flushes its copy of the data back to the underlying server[9].

Through these mechanisms AFS and an AFS-based follow up, DCE's DFS[7, 19], could potentially provide limited file system access while a network connection or server was acting up. However, these systems do not work disconnected for an extended period of time, this motivated the development of Coda for disconnected access.

Coda is a disconnected distributed file system that has several interesting innovations. Coda is built inheriting several features and design decisions from AFS[24]. While connected, Coda utilizes the same callback system that AFS does. At the same time, while Coda is idle, it hoards various files that a user wants to ensure are available in case of disconnection.

Furthermore, Coda balances the local disk cache between the needs of the user at that instant with the set of files that the user specified they wanted to have accessible while disconnected. Coda tries to ensure that low space on the local disk did not hinder working with additional data[12].

Another important thing about Coda is its handling of conflicts when reconnecting. If it detected that there was a write/write conflict with the remote server, it would end up aborting the replay of the log, and it would be up to the user to fix the state.

Coda also utilizes a lazy system of conflict resolution that is based upon a series of remote procedure calls enti-

tled ASRs[13]. Instead of using a remote procedure call framework, StashFS has a built in system and utilizes systems already built into the operating system.

StashFS's major improvement over Coda is removing the necessity of a central server that is running Coda. By allowing StashFS to shadow any mount point we gain more flexibility and potential uses. StashFS can serve as a purely local caching file system in addition to providing disconnected access to a networked file system.

Sun developed the Cache File System (CacheFS) in SunOS 2.3 to speed up and help address bottlenecks on servers. CacheFS is designed to use the local file system to store data and speed up usage with NFS and other slow medium such as CD-ROM and tape. The local file system can either be volatile (tmpfs) or non-volatile (ufs). If it is non-volatile, it would provide an already warmed-cache when a system came back up. To ensure synchronization, CacheFS polled attributes to see if files changed using a system similar to NFS[28]. Since then, other systems have also popped up that allow more generalized caching[26]. StashFS improves on ideas introduced with caching file systems by allowing them to work with a broader set of file systems and more importantly to provide disconnected operation.

Intermezzo is a follow up from the people who worked on Coda. Unlike Coda, Intermezzo does not use its own protocol for the underlying structure. Furthermore, it doesn't have its own underlying file systems, but instead utilizes the existing local file system. For example, it will use the existing ext2 file system which is different from how Coda and AFS work[20].

Intermezzo does this by introducing a layer that sits on top of the VFS. Intermezzo handles the logic behind file synchronization. However, Intermezzo ends up having to also handle a lot of journaling and it still operates utilizing a custom multi-RPC callback system. While Intermezzo has several interesting innovations, unfortunately it was dropped with the release of the Linux 2.6 kernel branch and thus is not readily usable anymore.

A follow up to Intermezzo is Lustre. Lustre itself seems to share some of the similarities in that it uses and exploits the client file systems and provides a wrapper layer thus creating a distributed cluster[17]. Unlike Intermezzo, Lustre does not support disconnected operation. StashFS could be used with a Lustre client to provide disconnected operation.

3 Historical Cycles and Current Issues

When it comes to bottlenecks with networked file systems the same problems have risen again. One of the reasons for the development of CacheFS was the fact that Network bandwidth was precious, but disk space was not as precious. But as the communication pipes and servers grew even larger, the ability to utilize them increased, and CacheFS became unnecessary as network response times became quite competitive[5].

However, currently our local machines are becoming quite powerful again. Even laptops have multicore processors, potential for GPGPU, and several gigabytes of RAM. Now utilizing workstation's local caches to decrease load on servers seems viable again. This will hopefully allow a benefit from decreased server load and faster response times as most users could fit their working set entirely in RAM and still have extra space.

There are several shortcomings that StashFS is attempting to rectify. In Coda, you have to use Coda as the underlying system for both the clients and servers. You cannot take advantage of any file systems that have had substantial improvements in their underlying structure and workings such as data checksums, silently layer things on top of LVM[16], or use ZFS[4].

While Intermezzo does allow this, it does not allow other operations. For example, Intermezzo does constrain the remote backing store to run Intermezzo. But if you don't have control over that remote system, then Intermezzo is not viable. It is a hope that StashFS will allow an arbitrary remote file system to be run, whether it is based on a DFS, accessed via sshfs, or some other system.

Another improvement to these systems is to add an ability to temporarily cache desired data, as opposed to using a permanent store. With the increase in RAM to a few gigabytes, most non-multimedia based workloads can be contained entirely in RAM.

4 Design and Architecture

At a high level, StashFS is meant to take either a local or remote share and mirror it into the opposite environment and provide synchronized and disconnected access to it. StashFS is also charged with detecting and managing conflicts that arise during use.

The design for this system falls into these categories:

File System Interposition: How the existing file systems on the system are utilized and provide a similar interface.

Disconnected Operation: How data is managed between remote and local file systems, during both the connected and disconnected operation.

Handling Conflicts: How conflicts from normal operation and reconnection are handled.

User Interactions: The tools the user has to aid in management and startup.

4.1 File System Interposition

Definitions This system is divided into two different pieces, which correspond to different mount points on the system: the local mount point and the remote mount point.

The local mount point refers to a part of the local file system that will ultimately be shadowed and have a FUSE based file system[1] interposed or mounted on top of it. The remote file system refers to some mount point that represents data that is stored elsewhere, generally a remote server, distributed file system, or cloud based service, though this could be a local file system. The local mount will provide disconnected access to the remote mount point.

Interposition Effects At the core, StashFS provides a layer that sits on top of an already existing local file system and interposes itself on all the various system calls that occur on that file system. Thus it becomes no different from any other mounted file system as seen by the operating system. The different operations that occur are taken and sent to both the local and remote systems and validation is done to ensure consistency with the remote system.

The core of this is handled via FUSE (Filesystem in Userpace). FUSE is a kernel module that provides a means for a file system to be run and debugged entirely in user land. The kernel module gathers file system requests and securely serializes them to the user land. The return values and any modifications are then sent back to the kernel where they are returned to the calling process. Several file systems have been implemented in FUSE, such as NTFS-3G[2] and SSHFS[3].

The second important piece is mounting another file system on the local machine that will shadow an existing file system. On invocation, StashFS will chdir to the target directory on the local file system and maintain a file descriptor to it. This allows StashFS to access the original underlying system. The user can only access it via StashFS which receives all of the intended system calls through FUSE. This flow of control is described in figure 2.

The end result of this is that the user will still see and have the same file system interface as before the interposition of FUSE. The data that is present is something that will also be maintained in the remote system.

To ensure that this is maintained persistently, the user can optionally add an entry to the operating system's fstab, so StashFS is mounted during normal system start up.

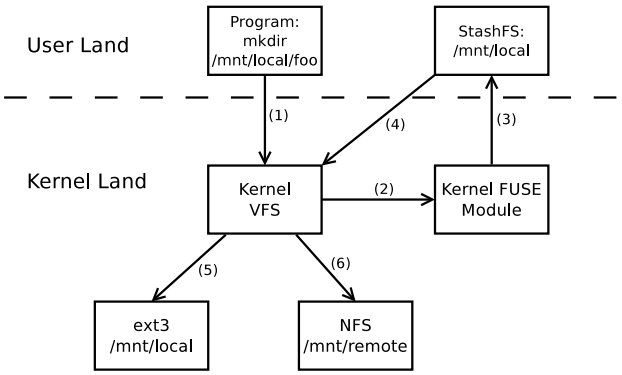

Figure 2: System call flow of control. The user calls mkdir, which goes to kernel VFS (1), then to the FUSE module (2), sent to StashFS (3), which makes the calls on the actual local and remote file systems (4-6).

4.2 Disconnected Operation

There are a few different pieces that need to be considered for disconnected operation. The first is handling normal file system operations. These operations have several potential parts: updating the local copy, validating that the operation makes sense in the remote system, and running the operation on the remote server.

We also have to handle the logic of pulling over files that are requested and ensuring that the correct working set is being stored on the local system. Furthermore, on reconnect we need to synchronize all the changes that have occurred.

Maintaining State Across system reboots, there is a need to maintain different pieces of information about the state of the file system. There are three different states that data can be in:

- The data is currently only on the local system (it still needs to be initially pushed out or was modified while disconnected)
- The data is only on the remote system (it still needs to be initially pulled in or is not cached)
- The data exists in both places and the timestamps and crc of the remote version is stored locally.

Conflicts are another important area for distributed and disconnected file systems. If a conflict is detected, then the data is simply uncached and put into the second state described above. Then it will be recached when a more recent version is needed.

To maintain this state, a hidden directory inside each directory is used. This would be implemented via something simple similar to a dot file on a Unix system and

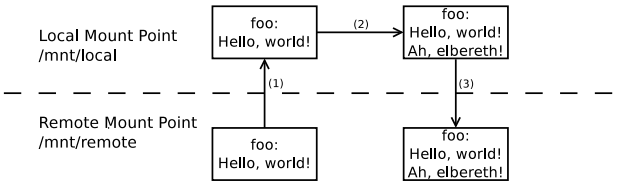

Figure 3: Flow of open, write, and close system calls

would not show up in any readdir calls. The file stores a list of all the dirents that exist in the directory as the actual directory will not necessarily have all of the dirents present. This file would also describe the state of each dirent's synchronization.

When a user enters a folder for the first time, the system will make an empty directory in local storage. It will then read all the dirents from the remote server and use that information to create the local status file. When a readdir call comes, it would use this status file to generate the list of readdir entries.

When a file is accessed, the status file is consulted before proceeding. From the status file one can find out if there is a locally cached version of the file. If there is, it will first compare the timestamps present with the remote ones. If they differ, it will compute the cyclical redundancy codes[21] for both files to validate that they are different and update the file if necessary. Figure 3 mirrors the flow of control when working with a file. If there is no locally cached copy, it will pull the file over and cache it (1). The reads and writes to the file will all go to the local copy of the file (2). When the file is closed, the entire file is copied back to the remote mount point (3). While better methods such as [30] exist for synchronizing differences, this approach was taken due to its simplicity, allowing efforts to be focused on other parts of the system.

If a file is removed from the cache, then this status file would be updated to note that it is no longer cached and the actual file would be removed from the underlying local storage. If any dirent of the directory is removed via the `unlink` system call, then the corresponding entry in the status file will be removed, along with any locally cached versions of the entry. Finally, when the directory itself is removed, its status file is removed along with the directory itself.

The status file is in charge of maintaining several different things:

- The name of each dirent present
- Whether that dirent is local, remote, or dirty
- The most recent copy of the stat information from `stat(2)`
- The crc of the remote file

Detecting Changes For any file that is currently synchronized between the server and the client, it is easy to detect whether or not it has changed by comparing the mtime for the file stored locally with the mtime from the remote server. If the mtime on the server is larger than the locally stored mtime and the crc values differ than the file has changed.

Files will be compared with their remote versions upon certain state-important system calls (i.e. open, stat, close, etc.) and when resuming from disconnected operation.

Log Definition To help maintain state, synchronize with the remote file system, and control disconnected access, a log of file system operations is used. The primary use for this is for disconnected access. During disconnected access, each operation will be appended to the log. From this information, it will be possible to replay the entire log to synchronize the local storage with the remote storage when the user reconnects. During regular connected operation the log does not need to be used.

Furthermore, the log is designed to ensure if an operation occurs during a crash that there are some guarantees about the validity of the data, as losing data is one of the worst things a file system can do.

The log is going to be structured as a linked list of entries that will be appended to when operations that change file and directory state occur. Each entry will consist of the following fields:

- The file system operation that occurred.
- The path(s) affected
- Any arguments that are necessary for replaying this operation (i.e. a mode_t to chmod)

There are a few important additional things to mention. Any time a file system operation occurs based on its file descriptor, for example, fstat, ftruncate, etc., before it is logged, the file descriptor is translated back to a path.

To better handle the cases where multiple operations are modifying the same file (i.e. multiple writes occurring in a row) as opposed to logging each individual write, we put the close call in the log. Then when replaying the log, we copy the entire file over to synchronize it.

If there are multiple write operations to a file across multiple open and close operations only the most recent copy of the data is synchronized while replaying the log.

Log Persistence Two different strategies are used to maintain the log. The first is to always append new entries to the log with synchronous writes a lock to guarantee atomicity. The second is to write the offset of the

most recently replayed entry to the first few bytes of the log. This way, in the event of a crash or pause in replaying, the state of the log and the most recently replayed entry can be determined.

Detecting disconnection Disconnection is defined by having an operation on the remote mount point time out. Every syscall on the remote mountpoint has a watchdog to detect the timeout. This is useful when using hard mounts with NFS[23], which will not time out when disconnected. NFS failures while testing have been successfully detected with this strategy. When this happens, the system switches to disconnected mode. Once the system manages to successfully establish communication again, the reconnection process can begin.

During disconnected mode, any attempts to open files that are not locally cached on the system will fail. Any files that were already open would be accessible because they are already present in the local cache.

Handling reconnection Our approach to handling reconnection is not to try to automatically reconnect to the system and replay the log. Rather, the user issues a command which checks our connection status and begin the reconnection process, via the console. When the reconnection command is issued, it will suspend all other operations in the file system and attempt to replay the log and deal with conflicts as necessary. Automatic reconnection was not used due to concerns about flapping: where the connection status quickly flips between disconnected and connected operation.

4.3 Detecting and Handling Conflicts

There are several different categories of conflicts to be concerned with. They are read/read conflicts, read/write conflicts, and write/write conflicts. In general, read/read conflicts are not a problem. It does not affect consistency if two people are reading a file at the same time. Similarly with read/write, it does not have an impact on consistency, as long as it does not occur in the same exact byte range. However, write/write conflicts are something that are an issue and are the primary concern for us to solve.

Conflict Detection There are two different occasions when StashFS might end up detecting changes due to a conflict. A conflict is defined as a file where the local modified time is older than the remote timestamp and the crcs differ. They are checked:

- On a syscall that requires checking remote state while connected.
- When replaying the log

The first case has two subproblems: files with and without local changes. In the case of an open, if there is a conflict and something is out of date, all that will happen is that the file will be synchronized before continuing. In the other subproblem, where there have been local modifications, all the writes to the file will be aborted. A hidden local folder could be used to store the conflicting copy while the main copy is purged from the local cache and is synchronized on the next request for it.

In the second case, once a conflict is detected the automated replay of the log is aborted. Replay then enters into a manual merge mode until all conflicts are resolved. Once complete system calls are no longer blocked and normal operation resumes.

Conflict Resolution In general, most conflict resolution will occur during log replay. This will be assisted via the console interface. When some issue occurs, the console will alert the user as to the nature of the error. It will then prompt the user as to whether it should skip replaying this entry or try replaying it again. The latter option is provided to allow a user to modify state on the remote mount point directly before continuing.

During the case of replaying a close operation, the options are slightly different. The user is permitted the chance of either overriding the remote version or to skip it and bring over the more recent version.

There are also times where it may be beneficial to the user to skip an entry while replaying the log, for example, if the remote server returns `EEXIST` to a mkdir call.

4.4 Interactions and Console

Local File System Considerations When deciding on what type of local file system to use when interacting with a remote server, it is important to choose the correct underlying file system. There are two main categories of file systems that could be used: volatile and non-volatile.

For a non-volatile system, all the data is persisted on storage such as a hard disk, cd-rom, flash, or tape. This will allow for a warmed cache across system reboots and crashes. It can also take advantage of any interesting features that the local filesystem has, for example, data deduplication.

The alternative is to use a RAM-backed file system such as tmpfs or some other volatile file system. Obviously, this will not allow for persistently warm caches, but RAM does provide for much faster access times.

In any of these situations the filesystem can be mounted read only. This allows for a purely caching file system for slower media such as tape, cd-rom, remotely stored data, or USB storage devices.

Console Design The basic design for the console is a Unix Domain Socket coupled with a simple terminal that facilitates communication over the socket. The console will be usable by the user who started StashFS, though that user can change the permissions on the bound socket to allow other users access to the console.

Console Actions The general idea for the console stems from the desire for a user to be able to control the ability to modify what they consider important and decache something for more space while disconnected.

More specifically the console will allow someone to:

- Purposefully disconnect
- Initialize reconnection
- Handle conflicts
- Uncache local copies of files

5 Testing and Results

5.1 Test Setup

For our tests we are using a client running Debian Lenny with an Intel Core 2 Quad Q6600 2.4GHz cpu with 3 GB of RAM and a Samsung HD160JJ 7200 RPM SATA II hard drive. We are connected over gigabit ethernet to a series of Dell R710s and MD3000s running GPFS[8] exporting NFS shares.

We are running the postmark[10] benchmark in two different test configurations with five different file system configurations. The first test configuration is supposed to emphasize a high number of transactions, but with a relatively small file size. The second test configuration takes the opposite approach, emphasizing fewer transactions, but significantly larger files. Table 1, shows the details of the two different test configurations.

We ran these tests in five different file system configurations. We ran them on the client's local hard drive formatted with ext3 and then ran them on an NFS mount. Next, we ran two different tests with StashFS using ext3 fro the local mount point file system. The first utilizes the local disk as the remote share (StashFS: ext3) and the second utilizes the NFS mount as the remote share (StashFS: gpfs). Finally, the last test is done while StashFS is disconnected (StashFS: DC).

5.2 Results

The results from running postmark can be found in table 2. The overall performance of StashFS while connected is about four to five times slower on the large file tests and about eight times slower on the small file

Item	Small Version	Large Version
Smallest File	50 KB	100 MB
Largest File	100 KB	500 MB
Number Files	10000	100
Number Transactions	100000	1000

Table 1: Postmark Configurations

FS	Type	Read	Write
ext3	small	3.14 MB/s	3.78 MB/s
gpfs	small	2.29 MB/s	2.76 MB/s
StashFS: gpfs	small	378.68 KB/s	467.42 KB/s
StashFS: ext3	small	400.79 KB/s	483.23 KB/s
StashFS: DC	small	694.10 KB/s	836.87 KB/s
ext3	large	17.76 MB/s	21.00 MB/s
gpfs	large	15.52 MB/s	18.35 MB/s
StashFS: ext3	large	3.71 MB/s	4.38 MB/s
StashFS: gpfs	large	3.41 MB/s	4.38 MB/s
StashFS: DC	large	13.42 MB/s	15.87 MB/s

Table 2: Postmark Test Results

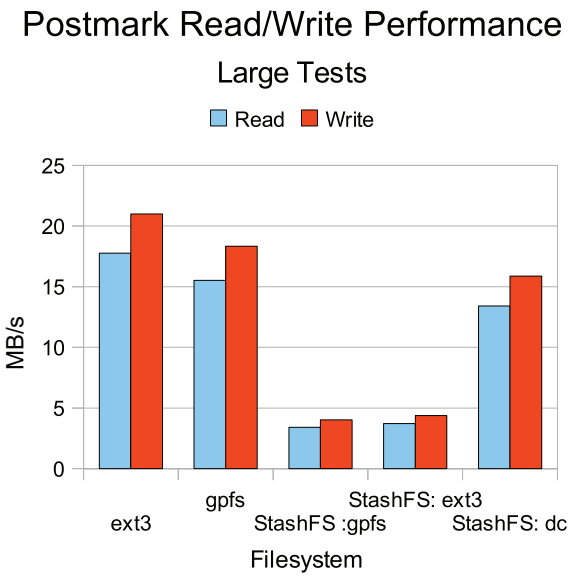

Figure 4: Postmark Large File Results

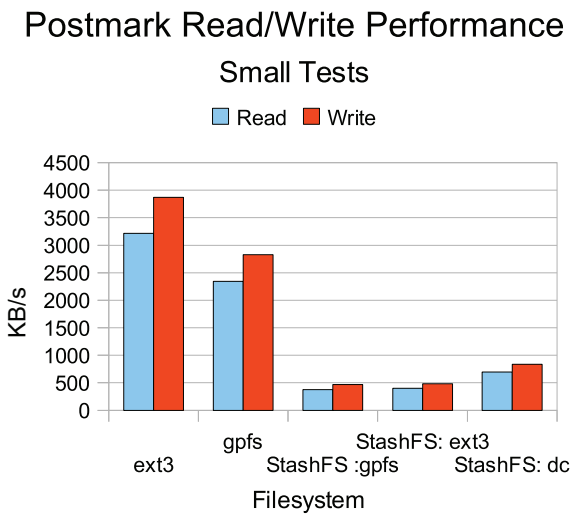

Figure 5: Postmark Small File Results

tests. These kinds of performance results makes it clear that StashFS is not well suited for high availability replication in server environments. Furthermore, there is an overhead associated with using FUSE[22], though it does not account for all of the performance differences seen.

From figure 4, we see the relative performance results from working with large files. The overhead in the large tests is mostly a factor of the synchronization that must occur at the end of each series of writes. With each close comes the synchronization of the entire file and thus the copying of several hundred megabytes of data. However, the performance while in disconnected mode manages to retain about 75-85% of the performance of connected operation.

From figure 5 we see that performance is much worse when there is a higher number of transactions on small files. This primarily comes from the way that we must detect disconnection, using watchdog threads for each system call. This adds a fair amount of overhead to each system call that occurs on the remote mount point. Furthermore, the performance in disconnected mode with small files is not as good as with large files, because the log file is being appended to synchronously. This guarantee helps with consistency, though in a high transaction environment it causes a large number of synchronous writes causing more performance overhead.

While it may seem that this overhead is not acceptable, it is important to remember that these workloads are not the most accurate representations of typical user workload. Generally users are not trying to maximize I/O operations per second for their laptops and desktops. Thus the relative overhead can be seen as an acceptable trade off for the cost of synchronizing data and allowing disconnected access.

6 Future Work

Large Files As the system currently stands, when a file is opened for the first time, the entire file will be brought across from the remote mount to the local mount. Say this file is a video file, that means it is in the realm of a couple hundred megabytes, that might take some time to transfer. Now say this file is a VM, that would be a few GB in size, and that could take significant time to transfer. Currently there is not a good way to deal with this in StashFS. Eric Schrock suggested a solution to this kind of problem in his work on Shadow Migration[25] by only copying across the data necessary to service a read request. StashFS could benefit from incorporating a similar system.

A similar issue exists as we perform operations that sync data back to the remote server. Currently the design does not call for maintaining where writes occur, just in the end that the file has been closed. Thus we end up synchronizing the entire file as opposed to just the changes. There are several strategies for tackling this such as rsync or ZFS snapshot send/receive [30, 27].

Remote Renames and Moves Oftentimes files will be renamed or moved on a file system. This is problematic as there is currently no way of detecting that something was moved or renamed at the current abstraction level FUSE provides. Rather StashFS will see it as one file that used to exist was unlinked and a new file was also linked into place. One possible future solution is to try and take care of some logic with respect to hashing the contents of the file and using that to detect that a file already exists and add another hard link to it, in effect adding rudimentary file level deduplication. Another approach is to try and work at the underlying filesystem's inode level rather than at the path level.

Log Optimizations There are several optimizations that could be made to improve reconnection time and performance. The first and perhaps the most important would be creating and storing the log as a DAG (directed acyclic graph) of changes as opposed to a linked list. This would allow dependency resolution to be done and potentially allow for branches of the log to be replayed in parallel and to occur even if an unrelated branch hits a conflict.

Another optimization that could be done is similar to one mentioned in Coda[12]. Here, we would use knowledge of subsequent writes or other changes to files to remove earlier entries from the log, thus reducing what is stored and replayed. Furthermore, like Coda, we could schedule cache walks periodically to update a user specified set of files. This allows for more files to be synchronized and thus a smaller likelihood for conflicts to arrive

while disconnected.

Continuous Replay and Modification Another improvement that would be quite useful is to allow modifications to be done to the file system and other synchronization to occur while the log is being replayed. Thus the user should be able to use data as per normal while dealing with synchronizing and data transfer. Also, during this time, something like a cache preference walk could occur to ensure that we get all of the most recent versions of files.

Console Namespace Integration One feature that would be both powerful and useful would be to transition some of the functions of the console into the normal namespace. Thus if a user owns the file and wants to decache it, they should be able to just use something similar to `rm` on it and flush it from the cache but not unlink it from the file system. Further operations in a similar vein could be advantageous and worth investigating.

7 Conclusion

Traditionally, disconnected access has required both remote machine support and has not worked across all filesystems. Further, with data outsourcing there are concerns over data availability in the face of network requirements and corporation viability. We introduced StashFS to address these concerns. StashFS only runs on clients, there is no server component. It works across any filesystems that the operating system knows how to mount. StashFS provides tools to mirror an entire data set from a remote site or temporarily clone the data for use with a ramdisk and provides disconnected access to this data with an acceptable overhead for users.

8 Acknowledgments

I would like to thank Professor Tom Doeppner for his invaluable help, critiques and reviews. I would also like to thank those in Technology House and the CS department who put up with me bouncing ideas off of them and helped critique this paper.

9 Availability

A copy of the source code and a bit of additional information on StashFS is available at

`http://cs.brown.edu/~rm/stashfs.html`

References

[1] Filesystem in userspace. Available from `http://fuse.sourceforge.net`.

[2] Ntfs-3g. Available from `http://www.ntfs-3g.com`.

[3] Sshfs. Available from `http://fuse.sourceforge.net/sshfs.html`.

[4] Jeff Bonwick and Bill Moore. Zfs: The last word in file systems. Available from `http://www.opensolaris.org/os/community/zfs/docs/zfs_last.pdf`".

[5] Peter J. Braam and Philip A. Nelson. Removing bottlenecks in distributed filesystems: Coda & intermezzo as examples. In *Linux Expo*, May 1999.

[6] Ben Collins-Sussman. The subversion project: buiding a better cvs. *Linux J.*, 2002(94):3, 2002.

[7] Thomas W. Doeppner, Jr. Distributed file systems and distributed memory. *ACM Comput. Surv.*, 28(1):229–231, 1996.

[8] Scott Fadden. An introduction to gpfs version 3.3. Technical report, IBM Corporation, 2010.

[9] John H. Howard. An overview of the andrew file system. In *Proceedings of the USENIX Winter Conference*, pages 23–26, 1988.

[10] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

[11] Michael Leon Kazar. Synchronization and caching issues in the andrew file system. In *Proceedings of the USENIX Winter Conference*, pages 27–36, 1988.

[12] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10:3–25, 1992.

[13] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, Berkeley, CA, USA, 1995. USENIX Association.

[14] J. Hamano L. Torvalds. Git-fast version control system. `http://git-scm.com`.

[15] Matt Mackall. Towards a better scm: Revlog and mercurial. In *Proceedings of the Linux Symposium*, pages 83–90, July 2006.

[16] Heinz Mauelshagen and Matthew O'Keefe. The linux logical volume manager. *Red Hat Magazine*, July 2005.

[17] Sun Microsystems. Lustre file system high-performance architecture and scalable file system. Technical report, Sun Microsystems, October 2008.

[18] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Third USENIX Confrence on File and Storage Technologies*, 2004.

[19] OSF. File systems in a distributed computing environment: A white paper. Technical report, Open Software Foundation, 1991.

[20] Michael Callahan Peter J. Braam and Phil Schwan. The intermezzo file system. In *In Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*, 1999.

[21] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1), January 1961.

[22] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *25th Symposium on Applied Computing*, 2010.

[23] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Summer 1985 USENIX Conference*, pages 119–130, June 1985.

[24] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[25] Eric Schrock. Shadow migration internals, October 2009. Available from `http://blogs.sun.com/eschrock/entry/shadow_migration_internals`.

[26] Gopalan Sivathanu and Erez Zadok. A versatile persistent caching framework for file systems. Technical Report FSL-05-05, Stony Brook University, 2005.

[27] Sun Microsystems. *Solaris ZFS Administration Guide*.

[28] SunSoft. Cache file system (cachefs) white paper. Technical report, SunSoft, February 1994.

[29] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[30] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, June 1996.