A File System for Laptops

Brian L. Stuart
University of Memphis
blstuart@bellsouth.net

ABSTRACT

Laptop computers do not fit well into the type of network that is built around one or more file servers. The need to approach diskless operation, keeping data on the file server is in direct conflict with the need for mobility of data. Here I report on work currently underway to develop a filesystem for laptops that resolves this conflict by transparently keeping copies on both the laptop and the file server.

Problem Statement

Experience has shown that there are many advantages to a file server separate from other systems in a network. Typically a file server is backed up more regularly than an individual's machine. The file server makes it easier to move to a new machine when the old one is replaced. It also allows users to make use of any machine available and still see their data. Finally, improvements to the hardware of a file server benefits all of the user community in a way that's much easier and often more cost-effective than upgrading all user machines.

In recent years, however, there has been a significant shift in computing resources. Unlike in the late '80s, most of us now use laptop computers at least part of the time, and many of us use them almost exclusively. Last year (2008), marked the first time that sales of laptops exceeded those of desk-top PCs. Unfortunately, there is a fundamental mismatch between the laptop computer and the file server-centric network. The very nature of a mobile machine demands that its persistent data be carried with it, where a file server is essentially data storage separate from an individual's machine.

Previous Solutions

More often than not, people who use laptops in an environment that includes a file server, approach things from the perspective that the file server is essentially a backup device for their laptop. Solutions created from this perspective generally make use of tar or rsync or something similar. Periodically, the contents of the laptop are mirrored to the file server. Some users take a full tar image each time, others use rsync or something similar to update the server. Approaches such as tra[1] and unison[2] improve on this synchronization by flagging a conflict when a change is made on two different machines between synchronizations.

Some have designed distributed file systems with an eye to handling the needs of laptop users. The most well-known is Coda[3]. One key element of these distributed systems is the push-like behavior of the file server. When a file changes on the server, it sends out a notification to any clients who are subscribed to that file so they can retrieve the updated version.

A Different Perspective

To a some degree the approach most people take with laptops is to treat the laptop as the primary repository of data and the file server as a form of backup. If we reverse this perspective and consider the file server primary and the laptop storage as a cache for times it's disconnected, we are led to a little different approach. In particular, when connected to the network, the laptop behaves as a write though cache. All writes are immediately relayed to the file server. Data read from the file server while connected are also stored on the laptop, making it available when disconnected. During times when the laptop is disconnected, it behaves as a sort of write back cache. Writes are recorded in a write log which is played back to the file server when the laptop is again connected to the network. Laptops that move among multiple home networks

can be handled by multiple instances of the file system with one running in connected mode and the others in disconnected mode when the laptop is on one of the home networks.

Within this perspective, there are several desirable characteristics we would like to have. First, we would prefer not to have to modify the file server. Depending on the environment, we might not have the privileges necessary to make changes. Furthermore, modifications to the file server would have to be implemented for each server OS we connect to. Second, we want the implementation to run in user space on the client. As with the file server, changes at the system level require privileges we may not have and require different implementations for each OS we might run on some laptop. Third, we want the user to be able to function the same way whether the laptop is connected to the file server or not.

First Approach

When I first started to implement a file system based on this perspective, I began with Plan9 and implemented a simple file system that maintained a local copy while talking to a file server. This server is called lapfs. The general approach was to use ordinary *read*(2) and *write*(2) calls both to the local file system and to the file server. The precondition was that the file server be mounted in the local name space. For example, suppose the file server is mounted on /n/remote and we have directories, fscache and fs in the home directory. The lapfs server is mounted to fs where it presents a namespace which is a copy of that rooted at /n/remote. Fscache is where lapfs keeps its cache. In this first implementation, lapfs maintained its own directory structures stored in regular files within fscache. When connected, on each open, lapfs would query the file server to determine if the file's last modification time was newer than the cached copy. If it was, or if there was no cached copy, lapfs would copy the file from the file server to the cache before completing the open request. Reads were generally served from the cache and writes went both to the cache and to the file server, though some experiments waited until the file was closed before sending writes.

After the success of the initial experiment, it was clear that I needed to implement this design in Linux as that is what my laptop at the time ran the vast majority of the time. This reimplementation was done using fuse. Fuse was chosen so that flapfs (as that version was called) could present POSIX semantics including symbolic links. This was important for some potential users at the time who wanted flapfs to serve their entire home directory, and some applications such as evolution were heavily dependent on symbolic links. The implementation was used heavily by myself and experimentally by co-workers. For two to three years, my laptops maintained caches of both my home file server and the file server at work. This arrangement worked quite well. I no longer had to worry about whether I had copied the latest version of a grading file to the laptop before going to work. When the time came to replace a laptop, I didn't have to retrieve any of my data from the old machine. I could simply install flapfs and begin using the file servers, and the cache would begin building on the laptop.

A New Approach

Recently, I once again had to replace my laptop. Even before making sure that the version of flapfs on my local file server was indeed the latest, I began to consider another, cleaner approach. The key observation was that there is no reason why the laptop file system should have to query the status of files on the file server all the time. When connected, all operations can go directly to the file server with appropriate copying to the cache. A substantial portion of the flapfs code had been devoted to modification time checking, and the associated copying operations. This along with attempts to improve performance mitigating the large number of *stat*(2) calls that were being made on directory searches was the most complicated part of the code. If we no longer worry about the versions, we can work at the level of individual operations, rather that at the level of files. In effect, the file system becomes a bidirectional form of tee.

Because I've been more immersed in Inferno than Plan9 of late, I decided to implement this new approach in Limbo. It was only after I began that I realized that this makes the file system operable on any system that supports Inferno, not just those that have a port of fuse. (However, there are a few open questions about using it on systems without P9Ps ability to mount 9P/Styx servers.)

Rather than use ordinary file calls to interact with the file server and the cache, it made more sense to establish channels to them and pass Styx messages back and forth. If the server (or

even the cache) is over a network connection, then the channel is established with *sys->dial*(2). If it is provided by a local server, *sys->pipe*(2) and *sys->export*(2) are used to establish the channel. In most cases, the file server will be on a network connection and the cache will be managed in the host OSs namespace by way of the #U server. In this implementation, all T messages from the client are forwarded to the file server, and all file server R messages are relayed back to the client. Client T messages, with the exception of Tread and Tstat, are also passed to the cache. In the case of Tread and Tstat, lapfs waits for the Rread and Rstat coming back from the file server. It then transforms them into corresponding Twrite and Twstat messages that are passed on to the cache. Special treatment must be given to Twalk. If both the file server and the cache produce successful results indicating that the full path has been traversed, then we need do no more. However, if the cache doesn't have all of that branch of the tree, then we must create it. From wherever the traversal stopped in the cache to the point where the traversal stopped in the file server, we create directories in the cache, except for the last path component where we might create either a directory or a file. In the current implementation, the handling of walk is the most complex part of the code.

Status

As of the time of this writing, the Limbo version of lapfs, working at the message level, has been implemented with the exception of three significant mechanisms. First, the disconnected operation has not yet been implemented. For this, the messaging strategy changes to simply relaying all client requests to the cache channel and all cache replies back to the client. This mode of operation also requires the second missing mechanism, that of the write log. The third major limitation is in the walk implementation. What is currently coded works as long as the cache can make one step of the walk so that it returns an Rwalk message. If, however, it is unable to traverse the first path component, then it returns an Rerror message. Path creation in response to the Rerror message has not been implemented as of this time.

It should also be pointed out that the current implementation has been *very* lightly tested. There is no doubt that significant bugs will be encountered and fixed as it is used on a regular basis.

Finally, the current implementation is based on a simplifying assumption, namely that the client will not have more than one T message in flight at a time. Lapfs serializes requests in the sense that it will not take or process another T message until the previous R message has been processed and returned. This avoided some fairly significant complications related to keeping T messages around until we know we can drop them. They would be needed because in transforming Rread to Twrite for the cache, some of the information in the original Tread is necessary.

Further Development

Obviously, the first step is the completion of the missing features. This must be done before it can be used seriously enough to evaluate further. However, once it is working to a functional level, it will be important to determine if it imposes a significant performance hinderance. This is of concern, because the Linux (and FreeBSD) flapfs did create noticeable delays in many circumstances. It also remains to be seen whether the serialization of requests is a significant limitation. If it is, then the bullet will have to be bitten and a list of T messages will have to be maintained and then referenced upon receipt of the corresponding R message.

References

[1] Cox, R and Josephson, W., "File Synchronization with Vector Time Pairs," MIT Computer Science and Artificial Intelligence Laboratory Technical Report, MIT-CSAIL-TR-2005-014, Feb 2005.

[2] Pierce, B C and Vouillon, J., "What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer," Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[3] Satyanarayanan, M, "Coda: A Highly Available File System for a Distributed Workstation Environment," Proceedings of the Second IEEE Workshop on Workstation Operating Systems, Sep 1989.