CloudFusion Design Document

Larkin Flodin

December 14, 2015

1 Introduction

The goal of this project was to implement a FUSE (Filesystem in Userspace) file system, with a back end that stores files as fixed size data blocks and inodes in Amazon's S3 service, using a DynamoDB table as a cache for S3. The file system is written entirely in Go, using Bazil's Go FUSE library (`https://bazil.org/fuse/`) and Amazon's provided Go SDK (`https://aws.amazon.com/sdk-for-go/`).

This paper describes the design and design decisions of this particular implementation; aspects of the system design are generally covered beginning at a low level and moving up to a higher level, before concluding with some possible directions for improvement.

2 File System Structure

As in a Unix file system, files in CloudFusion are a combination of fixed size data blocks and "inodes." Fixed size data blocks, as one might expect, simply store raw data. Inodes are small and maintain only metadata about a file (such as its size and whether it is a directory), as well as pointers to the locations of the file's data blocks, which are much larger. These inodes are then packed together into single data blocks. The abstract representation of a "file" presented to the user is really a combination of an inode and some number of data blocks. We first examine data blocks and inodes at a lower level, then consider how they are combined into files and directories, before explaining the initial mount process of the file system.

2.1 Data Blocks and Inodes

Data blocks are each associated with a unique 64 bit identifier, which is the moral equivalent of a pointer to its location in memory. In the current implementation they consume 32KB of space each, though this size could be easily varied. This is quite a large block size; 4KB is a standard size in many operating systems. The reason for choosing such a large size is that it results in better performance, both in terms of speed and cost. When considering speed, the limiting factor is not throughput (which in this case is upload/download speed), but latency. The latency of making requests to AWS services (particularly S3) is much more significant than throughput for

files on the order of 32KB, so there is little benefit to using a smaller block size. It is also more cost-efficient, as for relatively small files like these the cost of put and get requests to AWS services dwarfs the cost of the throughput. The one disadvantage is that for files much smaller than 32KB, this space is essentially wasted; however, since S3 gives effectively infinite storage capacity, and the cost of storing data is minuscule compared to the cost of accessing it, this is a small drawback. This drawback is also partially mitigated by allowing a small amount of raw data to be stored on the inode itself.

Inodes are much smaller than data blocks, only 512B in this implementation (though again, this size could be easily varied). They too have associated 64 bit identifiers. First and foremost, the inode stores metadata about a file, such as its size and time of modification (though not the file name). The inode also stores the identifiers of the data blocks of the file. Identifiers are stored for twelve regular data blocks, an indirect block, a doubly indirect block, and a triply indirect block. The indirect block stores only identifiers of regular blocks, the doubly indirect block stores identifiers of indirect blocks, and so on. This theoretically allows up to about 145,000TB to be stored in a single file, though likely this is larger than most operating systems allow. This metadata takes up a very small amount of space, only about 140B. The remainder of the inode houses a small buffer, which acts as a mini-data block for the associated file. This allows very small files to avoid allocating a full data block. As the size of the inode is 512B and the size of a data block is 32KB, there are 64 inodes combined into a single data block.

2.2 Files and Directories

As mentioned above, files (and directories) consist of an inode along with some number of data blocks. In a general Unix file system, it need not be the case that there is exactly one inode per "file," as multiple files presented in the file system can be associated with the same inode and data, but that functionality is not provided in this CloudFusion implementation. Data files work as one might expect; the raw data associated with the file is stored in the inode buffer and the data blocks pointed to by the inode.

The structure of directories is more interesting, though similar to most Unix file systems. File names are not stored on the file's inode, but instead are stored in the inode and data blocks associated with its parent directory. Directories are really just special kinds of files, whose data is a list of file names and their corresponding inode identifiers. To access a file with a particular name, the name is looked up in the parent directory's data, and then the corresponding inode is accessed. In this CloudFusion implementation, the directory data literally stores a hash table that maps file names to inode identifiers.

2.3 Mount Process

Mounting the file system when the program is started consists primarily of using external data to initialize certain resources. This external data comes from three sources: command line arguments, the configuration file, and the superblock. From the command line, the path of the configuration file, the size of the cache, and a flag for running tests are all supplied. The configuration file contains a variety of information that is expected to remain the same across mounts, such as the

mountpoint. It also contains the names to use for the AWS resources; these resources (the S3 bucket and DynamoDB table, to be discussed later on) are attempted to be created if they do not already exist.

The other source of mount information is the superblock, which is stored in S3. The superblock has a special identifier that allows it to be loaded during the mount process, and is uploaded from a cleanup function called when the program is terminated. If the superblock cannot be found on boot (such as when mounting the file system for the first time), then a new one with default values is created. The superblock stores the identifier of the inode for the root directory, as well as identifiers corresponding to the next available inode and data block that are not in use. Unused data blocks do not actually exist yet; rather, an identifier is provided for a non-existent data block, and the actual block is created when an attempt to get it from DynamoDB and S3 fails.

Old data block identifiers are never reused, as there is really no reason to do so when the blocks are not pre-allocated. Inodes work in a similar way, but note that since there are 64 inodes per block, as files (and thus their corresponding inodes) are deleted, the blocks would begin to become very fragmented if inode identifiers were not reused. To avoid this situation, when an inode is deleted its identifier is added to a linked list, and new inode identifiers are taken from the linked list when possible. The result of this is that the linked list must also be stored on the superblock. It is actually possible for the list to span multiple blocks, in which case further superblocks are initialized, though this situation is unlikely to occur (it would require the file system to be unmounted after deleting thousands of files, but without creating any new ones).

3 AWS, FUSE, and Caching

Most of the work CloudFusion does is converting between the functionality provided by FUSE, and the file system back end, stored in Amazon's S3 and DynamoDB services. As such, it is worth examining these systems in more detail.

FUSE is a technology that allows things other than standard Unix files to be represented as Unix files. Through a combination of FUSE and the Go FUSE library used in this project, when the file system is mounted, all system calls interacting with the FUSE file system are intercepted, and are then carried out using CloudFusion's code. The work that CloudFusion must do is to translate a syscall that says something like "delete file A from directory B" into something like "delete all relevant data blocks and the inode associated with file A from S3 and DynamoDB, and delete file A's entry in the data hash table for Directory B." Each of these tasks is then broken down much further; for example, "delete file A's entry in the data hash table for Directory B" is really something like "get all the data pointed to by directory B's inode from S3/DynamoDB, unpack it into a hash table, zero the entry for file A, pack the hash table back into data, then reupload that data and inode to S3/DynamoDB."

For back-end storage of blocks, both S3 and DynamoDB are used. S3 is a long-term key-value store, with relatively high latency and (literally) inexpensive accesses, whereas DynamoDB is a NoSQL database, with relatively low latency and expensive accesses. S3 is used primarily to store the state of the file system while it is unmounted, and DynamoDB is used as a cache while the file system is running. Thus, reads are first tried in DynamoDB, and resort to S3 only if they fail; similarly,

writes always write to cache, and data is written back to S3 only when the cache is full. The cache employs a least recently used eviction algorithm, achieved via a combination of a linked list and a hash table that allows for constant time access to list elements.

Because the internal structure needed to maintain the cache is rather complicated, the cache is simply emptied into S3 during file system cleanup. It might be desirable to maintain the cache state when the file system is unmounted, but this would require compressing and storing these data structures across one or more blocks, and then decompressing them during the mount process. While emptying the cache after every unmount may seem expensive, note that it cannot really be avoided even if the cache state is saved; if the file system is mounted with a smaller cache size than the previous mount, the cache would have to be partially emptied during the mount anyways.

For the purpose of actually storing blocks in a key value store, each block is assigned a key which combines its type (superblock, inode block, or data block), its 64 bit identifier, and a hash of the first two values. The hash is prepended to the rest of the key, because this supposedly allows for better S3 performance in high throughput settings – S3 does some amount of partitioning using the first characters of the key, so if they were all the same, the partitioning would be ineffective.

4 Directions for Further Improvement

Though the current implementation does what it set out to do, there are certainly areas in which it could be improved. There are possible improvements that could be made in performance, robustness, and functionality, given more time.

Likely the greatest deficiency of the file system currently is that it is slow. To some degree this is unavoidable; reads and writes will be inherently limited by the available network bandwidth, for example. Also, there is significant latency inherent in accesses to S3 (and even DynamoDB). Thus, the gold standard would be to achieve maximum bandwidth on reads and writes, with the latency equal only to the time required to access S3. Currently, the system does not achieve this. The main barrier lies in making reads and writes more asynchronous; to some extent these operations are already asynchronous, in that multiple syscalls from the kernel are handled asynchronously by the file system server. However, it should theoretically be possible to split each of these syscalls up into multiple concurrent portions.

Since most of the slowdown derives from high latency requests, issuing many requests concurrently could lead to a large speedup. There are nontrivial barriers to this model, though. While most operations are easily parallelizable since they operate on distinct blocks, there are certain things that are interdependent, like allocating new indirect blocks. If two concurrent calls both allocated a new indirect block corresponding to the same portion of the file, one would overwrite the other. These sorts of effects could probably be avoided with some restructuring of the existing system. Another possible performance improvement would be to maintain the cache state across mounts, as described above, though the benefit of this is far more marginal.

The robustness of the system could also be improved. Currently there are certain operations that, if interrupted, can result in varying levels of "corruption" of the file system. The cleanup operation is one of them; if the cleanup operation fails, the most up-to-date version of the superblock is not uploaded, which causes the identifiers for newly allocated data blocks and inodes to be too small,

overwriting existing data. This is difficult to fix, as the only foolproof solution is to re-upload the superblock after every allocation, which is very expensive. Perhaps a hybrid data storage approach which stores the superblock locally would be the best solution. Also relating to robustness, the test suite associated with the file system could be improved. Currently it supports only end-to-end tests and unit tests, without much room for a middle ground, such as testing individual components of the system (like the cache) end-to-end. Enabling these types of tests would require significant restructuring of the code so that components are more loosely coupled.

Finally, more functionality could always be added. The file system could be modified to allow simultaneous mounts from multiple machines, though this would be a major feature to implement in a way that is convenient for users. Another possibility would be to add some more complicated Unix file system features, such as symbolic linking. This would not be particularly difficult, as the necessary infrastructure should already exist, but it also does not provide any huge benefit.