1 Introduction

We have built a working NFS like distributed file server which provides transparent remote access to file systems. In the following sections we discuss the design and implementation of our work.

2 Design Flow

2.1 Basic Layout

The client side file system semantics is provided via FUSE while on the other end, the GRPC server provides the functionality of the NFS server.

2.2 Mount

The remote file system to be mounted on client is passed as an command line argument to the FUSE filesystem which makes an appropriate mount call to the server via *init* FUSE operation at the mount point provided. We have tried to stick to NFS-2.0 semantics and our file handle consists of a volume ID, inode number and a generation number.

3 Performance Optimisations

3.1 Server

For faster lookup every time a new file handle is generated we populate it in a Hash Map to prevent a redundant file handle to path translation every time a file handle is handed over to the server. The file is opened during the very first I/O access (read/write) or during explicit creation form the client (via create). Moreover we have an open file descriptor to file handle map in the server side so that we can reuse the file descriptor for subsequent file accesses without incurring the cost of an $open()$ and $close()$ every time. We are not persisting this map table over server crashes, all the files will be reopened and the table will get repopulated once the server recovers and starts serving client requests. One of the issues of this file map is that during periods of peak load and high utilization this map can be prohibitively large, and the number of open file descriptors might reach its limit. In order to tackle this, we must have a garbage collection mechanism for removing the unused file descriptors from this map. We cannot rely on the client's $close()$ for deleting closed file descriptors because- firstly we do not have a separate close protocol in NFS secondly, even if we include one such close protocol, what if the client is malicious and exits without closing the file? A number of approaches can be taken up for handling this- we can keep an upper bound on the number of entries of the hash table(which should be less than OS specific restriction on the total open file descriptors allowed) and garbage collect by LRU policy once we have reached this threshold. A simpler timer based garbage collection can also be used where an entry is removed on time out since its last access. For the sake of our demonstration, since we are running with small number of files, we can safely assume that we will not run out of file descriptors. Hence we just delete it when client explicitly removes the file.

3.2 Client

We maintain a file handle to path translation hash map at the client side also, but here the key is the path. This way we save a NFS *Lookup* call each time we access a previously accessed file/directory. This hash map is not persisted, the client starts afresh every time. For the path and file descriptor mapping, we rely on FUSE's internal file descriptor handling mechanism.

3.3 Commit

We have implemented a simple commit strategy where on a $fsync$ request from the client, we flush file. We do a file descriptor specific $fsync$ instead of a $fflush$ on the server side to stick to the per client commit policy. A point to be noted is that in case two clients are writing to the same file, they will be sharing the same file descriptor on the server side, and a *commit* from either of the clients will force write the changes of both the clients. We rely on the OS buffer cache to do the batching for the writes. In order to do an explicit sever side buffering, we would have to mmap the files and on a client commit request, do a *mync*. For the sake of simplicity we avoid this and just rely on OS based buffering.

4 Crash Consistency

We maintain a persistent copy of the hash map of the file handles by force writing them to the disk. Every time the server crashes and restarts, it recovers the entries from this persistent copy. For ideal crash consistency we should force write a log entry alongside every hash map entry. But this is very taxing performance wise, as it incurs the costly $fsync$ for every entry. Hence we optimized it by $fsync$ ing only after the insertion of a fixed number of entries, the tradeoff being that the hash entries entered after the last log write till the crash is lost. Now after the server recovers, if it receives a client request corresponding to one such file handle as described above, the server will return an error message which is functionally incorrect. To tackle this we have a recursive lookup based approach where the file handle is extensively searched for starting from the root node in case it fails to find it in the hash map. In the event, a file handle lookup fails this search, we can confidently conclude that it is an invalid file handle. In fact we can do away with the persistent logging of the hash map entries and can just have this recursive lookup instead. But the tradeoff here is that the first access for a valid file handle from the client will be very slow especially when the filesystem is large and the file handle to look for has a long path. Another optimization can be *fsync* ing the hash map entries together with the client data each time a *commit* is done. This way we can piggyback the cost of file handle to path translation persistence with client data persistence.

5 Multi Client Scenario

We are running GPRC in sync mode where the server has an (internal) thread pool that manages multiplexing requests by mapping them onto some number of threads (reusing threads between requests).The only point of race condition in our system is the hash map, hence we perform all the updates to the table atomically via mutex.

5.1 Remove

For removal we atomically insert a tombstone entry for each record deleted along with the actual deletion of the file/directory so that whenever other clients with the old file handle try to access it, it will show up a "File not found error". This violates the POSIX semantics that allows operations with open file descriptors even if the associated file is removed/renamed. But for the sake of simplicity we avoid it here.

5.2 Commit

We can have two cases when a server crash occurs during a *commit* protocol. First is fysnc goes through successfully but the server crashes before sending a reply to the client. So a reductant call to *fsync* can be avoided by using replay cache the next time client calls *commit* after server recovers. In the other case, the crash occurs in the middle of a *fsync*. for this scenario, we have to retry *fsync* after server recovery. Conclusion the design has some limitations like it does not support linking and full POSIX semantics. But we have tried to stick to the original NFS design as closely possible over easier implementation options.