NFS-like File System

Zhichao Yang, 20661179 & Liwen Dai, 20552153

Design

Communication Protocol: GRPC
From the beginning we attempted to use UDP/TCP to implement our system. After some effort, however, we had to abort, since serializing C/C++ structures to packets would be too much work (endianness, memory layout, offset calculation). In contrast, GRPC with protobuf makes implementation much easier.

Server Design: single-threaded GRPC service
For simplicity. Ideally we would like to build a server with a pool of threads. Staged / events-based architecture looks unrealistic for this assignment.

Server Persistent Storage: default linux file system
We considered to deploy an independent file system on the server, but didn't find an appropriate way to access the disk directly. So we chose to make NFS works on client side first. This means our NFS server doesn't use FUSE integration.

FUSE Interface: high-level, <fuse.h>
This interface is easier for us to understand. The low-level interface should provide a better performance though.

Functionality:
Out NFS support basic file system functionality, including create/delete files and directories, read / write. In client_fuse_ops.cc the list of implemented fuse operations can be found.

Crash Recovery: not implemented
The main reason is that when the connection is lost, the client side generates a segment fault (assertion failure) from grpc library, which cannot be caught like an exception. The client gets killed immediately and we didn't find a way to get around this problem. In addition, we also have to find a way to gracefully unmount the file system.

Two versions of code submission:
We submit two versions of NFS, v2 (mostly by Zhichao) and v3 (mostly by Liwen) (v1 is the socket one). The reason is that we found it hard to fix a design of interface at the beginning. Understanding the code of each other was also nontrivial.

Step-by-step path lookup: implemented in v2
This function was hard to implement and made code harder to read. Both versions use file-handles to to read/write. In v

Batching writes: implemented in v3 ?
In v3 the server doesn't store the written buffer explicitly. Because different write requests can have every different offsets, it was unclear to us how to "put them together". Instead, in v3 the server calls write(), which doesn't guarantee whether the data is synced to disk. Flush only happens when the client calls release(). This approach makes Linux server decide when to flush the buffer.

NFS v2

1) Run the system
    a) First the server ip address in client_fuse_ops.cc and server_grpc.cc need to be changed.
    b) To run the server, run command "./server_grpc"; to run the client, run command "./client_grpc serverpath mountpoint". Serverpath is the directory path at server you want to mount to, mountpoint is the directory path at local machine you want set to be the mount point.

2) In nfs, the file handle has three attributes: inode number, inode generation number, and the filesystem id. Since our example is simple, we discard the filesytem id. Therefore, we only use inode number and inode generation number as file handle.
3) When the client setup, the client will mount by get the file handle of the root path in nfs_init().
4) To maintain the file handle, on the client side, it keeps a map structure from path to file handle. When open(), create(), and mkdir() are called, the file handle for that file or directory will be saved for later read or write. Upon file open, the path of the file is resolved step by step, each level of the parent directory will be checked if we have the file handle, if not nfs_lookup() will find the file handle for that directory. On the server side, server keeps a map structure from released file handle to local path. When server get a read() or write() call, it will use the map to find the local path for the file handle.
5) For nfs v2, the current implementation cannot handle server crash properly, it requires client to do a new mount. Since upon server crash, the file handle to path map stored in server would be lost. When server come back, client use old file handle do operation would get error from server. One naive idea would be, save the map into a file on disk at every time we try to insert value into the map, and when server come back, it can fill its map from this disk file.

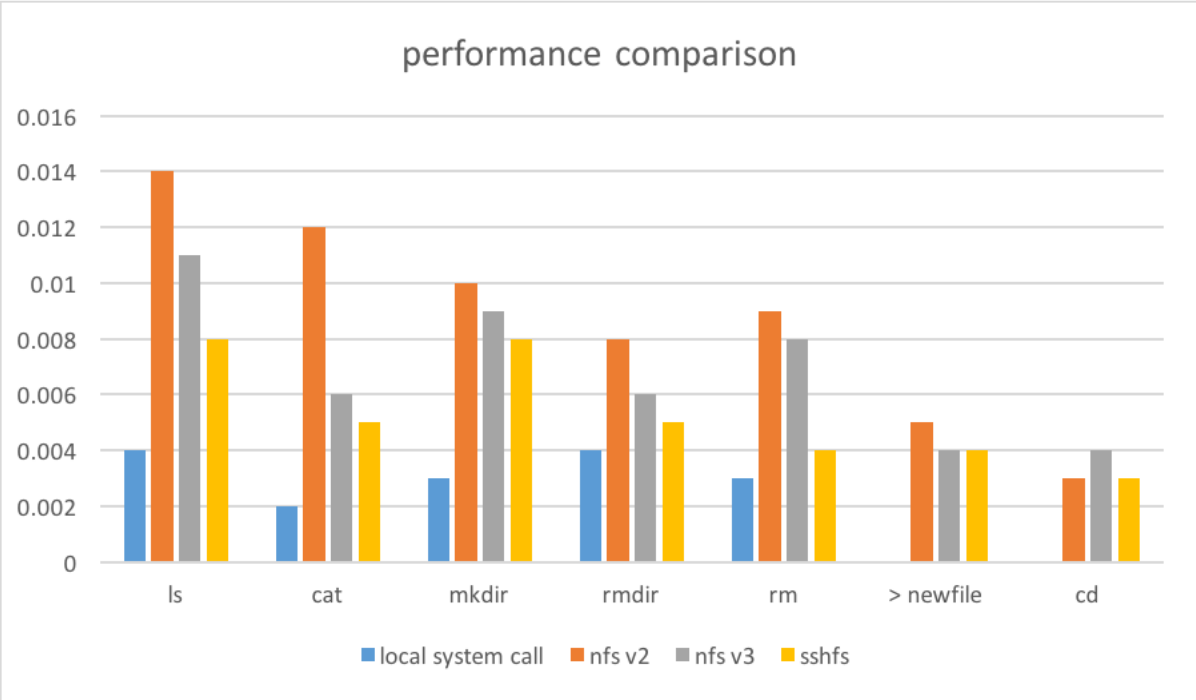
NFS v3

The idea of v3 is to provide a better software engineering: a easy-to-understand programming interface. While v2 started with implementing path lookup on the hardway, v3 started with a naive approach, without optimization at all, which is similar to "example/passthrough.c". Code for file handles and batching writes were added afterwards.

V3 uses file descriptor as the file handle, which is not a good way (yet works with single client). File handle should follow an approach like in v2.

Run the system:
Modify “server.config” and “client.config” to have correct address and port. Then
$ ./server_grpc <dir>
$ ./client_grpc [-f] <dir>

Performance 1:



We have tested the run time for the most common linux command as shown in the above graph. The run time in our implemented systems is 2 times longer than the run time at local linux system in average. The behavior of nfs v2 and nfs v3 is similar to sshfs file system. For the most interesting part, the “> newfile” and “cd” command takes almost 0 second to complete. While the other three take around 0.004 second to finish.

Performance 2:

The following measurements compare the performance of nfs_v2, nfs_v3 and linux local file system (basically disk test). NFS server and client run on two different nodes, but connected by private network (Digital Ocean droplets, internal net speed is about 2Gbs).

The contents of working directories are identical. When measuring nfs, the current working directory on the client is the root of mount point.

Write throughput test:
$ dd if=/dev/zero of=./testfile bs=256M count=1 oflag=dsync
Client reads a local file and writes it to server.

Local file system: 185 MB/s;
NFS_v2: 60.7 MB/s;
NFS_v3: 24.8 MB/s

NFS_V2 wins. This indicates the batching writes implementation on NFS_v3 is probably wrong (though it works).

Latency for reading directory:
$ time for i in `seq 1 1000` ; do ls > /dev/null ; done
Local file system: 1.997s;
NFS_v2: 5.029s;
NFS_v3: 4.778s

Read throughput
$ time for i in `seq 1 1000` ; do cat fuse-3.0.0/README.md > /dev/null ; done
Local file system: 1.533s;
NFS_v2: failure(1.094s for 200 iterations);
NFS_v3: 3.017s

Write Latency
dd if=/dev/zero of=./testfile bs=512 count=1000
No local file system test because of cache issue.
NFS_v2: 1.12643 s, 455 kB/s
NFS_v3: 0.75183 s, 681 kB/s

Discussion

Even if network throughput is significantly faster than disk, the performance of our NFS just cannot be as good as local file system. The overheads include FUSE, GRPC, network, and file system calls on the server side.

However a good implementation can still make a difference. For example, NFS_v2 beats v3 in the writing throughput test.

Implementing a reliable file system is hard. Both v2 and v3 are suffering from the “too many opened files” problem, and we didn’t find a way to handle server crash. Building a good NFS requires knowledge, good engineering practice, hard work and cooperation.