# Simple Network File System with FUSE and GRPC

CS798-W17 Advanced Distributed Systems
Instructor: Prof. Samer Al-Kiswany
Group: Anthony, Zheyu Xu

February 22, 2017

## 1 System Design

Our Simple Network File System (SNFS) leverage FUSE and gRPC projects. FUSE allows us to intercept system calls generated by file-system operations and gRPC enables the client to invoke procedures declared in the server with necessary parameters.

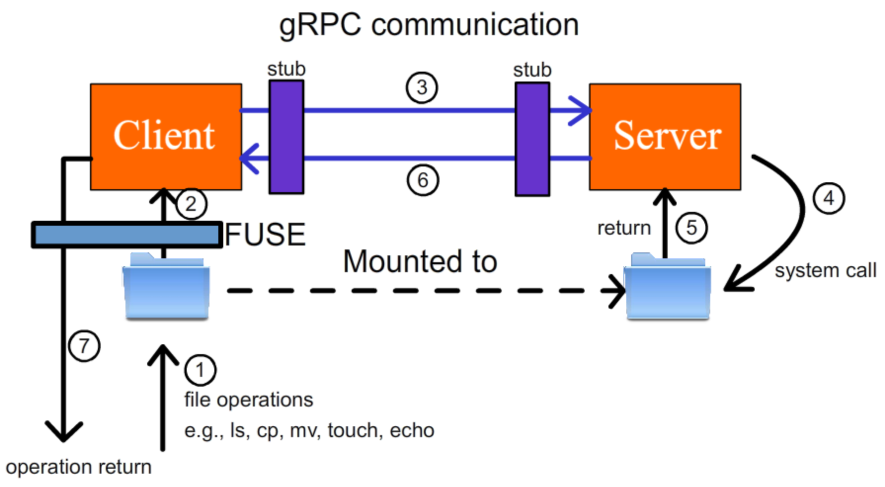

Figure 1: SNFS Design Architecture

Figure 1 depicts the design architecture of out SNFS. At first, the server starts and listens to gRPC connection. Then when the client starts, it create a gRPC channel with the server. After the channel is created, the client will perform mount operation trying to mount the server directory to the specified directory in the client.

A user can do common file-system operations to the client directory (running SNFS), including *ls, mv, touch, echo, mkdir, rmdir, tar, and others.* FUSE kernel module intercepts function calls generated by these commands and call the corresponding pre-defined fuse operations. In the pre-defined fuse operations, our system will pass the parameters and calls the remote functions at the server using gRPC.

## 2 Optimization Design

There are two optimizations implemented in our system (SNFS), i.e. batch-write and server crash-recovery that will be detailed in the following subsections.

### 2.1 Batch Write

The goal of batch-write optimization is to reduce the frequency of disk I/O. Each call to write() system call will not persist the data to disk, instead, it is being placed temporarily in the memory. Only when the commit() is being called, the data placed on temporary memory space will be flushed to persistent storage.
We store the data (and metadata) for each write() calls to C++ vector variable on both the client and server sides. This enables the server to directly response to the client's write() after pushing the data to the vector (not calling pwrite()). Eventually, when the client finished writing, it is responsible for invoking commit protocol to server. Upon receiving commit protocol, the server will merge all the data batched in its vector and call a single pwrite(). Currently, our optimization only batches multiple writes into a single file.

### 2.2 Server Crash-Recovery

Due to batch-write optimization, server-crash can cause data being buffered in the memory lost. Overcoming this issue, we employ a recovery mechanism requiring the client to send start and end offset of the data to be committed in the server. The server then compares these offset and determines if it has the complete set of the data to be written to disk. If the these offsets do not match, then the server will ask for re-transmission from the client. In this project we assumed that during re-transmission the server will never crash again. Therefore, upon receiving the re-transmitted data, the server can write the retransmitted data combined with the data it already had into disk.

## 3 Evaluation

In order to evaluate SNFS, we measure the performance in terms of three aspects: the latency improvement achieved through batch-write optimization, the extra overhead caused by server crash and rebooting, and the overall I/O overhead compared with Linux file system.

### 3.1 Batch-write Optimization

In our experiment, we wrote a one-kilobyte message to a single file. This process repeated for 1024, 5120, 10240 and 15360 times, respectively. During the process, the file keeps open until the file close() function is called in our C++ program. In consequence, the sizes of the batches are 1 MB, 5 MB, 10 MB and 15 MB. A batch of data is written into the disk in one pwrite() system call, aggregating small I/O operations into a large one, while the write-through strategy does an I/O operation every time it receives the one-kilobyte message.

As is shown in Figure 2, the latency of writing operations decreases significantly when we use the batch-write optimization. With the growth in the size of a batch, the difference of the two I/O strategies becomes larger, indicating the overhead caused by small I/O operations We attribute this to the fact that batch write requires only one disk seek and one system call. Whereas in write through strategy, the number of both system calls and disk seeks increase proportionally to number of writes.

### 3.2 Crash-recovery Overhead

In our implementation, the client will re-establish the gRPC connection channel with the server if it detects the crash of the server. This process is done in the background, so the users of SNFS will not notice that server has crashed(if the server reboots within a short time) but may suffer from performance degradation. However, if the server crashes and reboots during the batch-write operation, the data buffered are lost, causing the inconsistency problem. In order to solve the problem, we designed a retransmission protocol and measure the extra time cost of connection re-establishment and data retransmission.

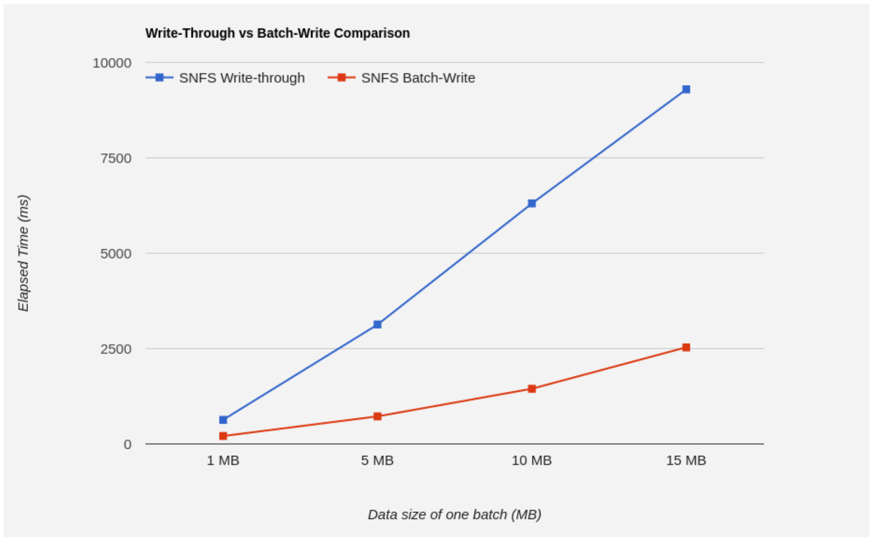

Figure 2: Comparison of Batch Write and Write Through

### 3.2.1 Assumption

We made the assumption that the server will only crash for at most one time, either during the commit step or during the writing steps, i.e., the server is always reliable at the retransmission stage.

### 3.2.2 Server Crash During the Commit Step

When a user closes a file, FUSE client will send a commit message, which contains the offset, to the server. If the server finds the offsets not matching, it will force the client to retransmit all the data. Only when the ACK of the commit message is received will the client releases the data in its buffer. We achieved this fault injection by calling an rpc_kill() function at the server when users close files, which terminates the server.

### 3.2.3 Server Crash During Writing Steps

If the server crashes and reboots between multiple writing operations, the client will continue to send its data generated by following writing operations. Before writing the data into the disk, the server will make sure it has consistent data with the client, or it will require the client to resend the data that are lost due to the server crash.

### 3.2.4 Experiment

We measured the overall latency of writing a 10 MB file. In the first case, a fault is injected at the commit step, forcing the client to retransmit all the data. In the second case, a fault is injected when 5 MB of the data are sent, i.e., the client only has to resend the first half of the 10 MB data. We compared their time cost with that of the non-crash case. Although the latency is increased unavoidably, the results in Table 1 show that SNFS tolerates the server crash at reasonable cost.

| | No crash | Crash during commit | Crash during writes |
|---|---|---|---|
| Time(ms) | 5319.79 | 11736.22 | 8590.68 |

Table 1: Cost of Server Crash and Recovery

### 3.3 Comparison with Linux File System

We also compare the performance of SNFS with that of NFS4 Linux. The latency of writing an 10 MB file is measured. Figure 3 shows that the performance of SNFS is extremely poor with regard to the-facto NFS system. The extra time cost is caused by more stages of function calls (including kernel and userspace) in FUSE interface and message passing that require copying data must go through network stack of the both client and server in gRPC implementation. Although SNFS provides the interface that allows users to operate remote files in the same way as they do to local files. The performance of remote file operations are inevitably degraded.

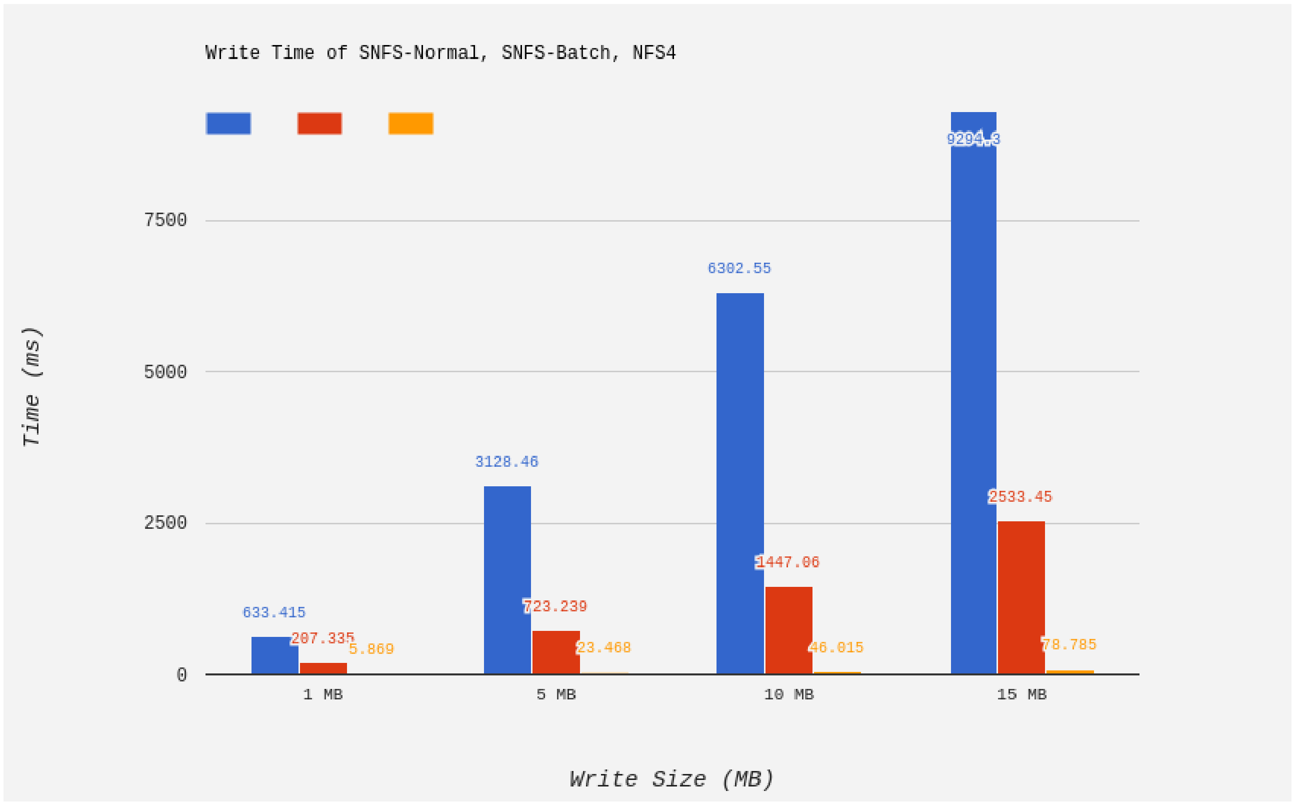

Figure 3: Overhead of FUSE and gRPC

## 4 Conclusion

SNFS is a simulation of the Network File System, which allows users to operate remote files in the same way as local files. It leverages FUSE to build the interface of the virtual file system and uses gRPC for communication. SNFS implements batch-write optimization, minimizing the overhead of I/O operations. It also has good fault tolerance with regard to server crashes.