CS754 A2

Lingyi Zhang and Yuxin Fan

October 2019

1 Design

This project is a basic implementation of NFS-like file system that supports basic Linux operation including *cp, mv* and *tar* and basic file system operations including LOOKUP, READ and WRITE. The system can also support file and directory creation and deletion. Beside the basic implementation, it also support the COMMIT for optimization.

1.1 Basic Implementation

Use Fuse to build the client side Linux system and use gRPC to communicate between the client and the server. The project implements fuse operations *getattr, readdir, read, mkdir, rmdir, rename, create, open, release, write, flush, unlink, fsync.*

For each fuse operation, the client side will send a gPRC request to the server, and the server will execute the exact same operation on this own file system. In this case, server can keep track of the various NFS protocol requests the client asked and the file is stored persistently.

The server keeps a "path to file handler" mapping which should be the primary copy. The client keeps a subset of this mapping to reduce the number of lookup.

1.2 Optimization: Commit

Notice that if we let the server and client do write operation synchronously, the server will need to write data to the disk for each write operation. This operation slow down the server, so we also adopt the COMMIT operation from NFS v3 spec to optimize the performance.

The server will keep a copy of the data that the client want to write to the file in its cache instead of writing the data directly to the disk. Each chunk of data contains an unique id (gen) to distinguish between other chunk. If the server crushes before the data is written to the disk, the server will ask the client to resend the chunks to make sure no data is lost.

The client will keep a copy of the data that it want to write to the file on its local cache, and only after it successfully commit the change to the server, it

will remove the data. If the server do not have all the copy it needs, it will ask the client to resend the chunks. In this case, the client will send all its copies to the server.

1.3 Fault tolerance

They're two major scenarios for server crash.

Case 1: Fail to send Ack to the client

This happens when the server crashes before receive the client request, server crashes during the operation (before sending the ACK), or the message lost during the communication (network failure). Every time the client sends a gRPC request, the client will wait for server to send back the status. If the status shows that the request is not successful, the client will resend the request.

We rely on gRPC's at most once guarantee to prevent client request being executed twice during the retries.

Case 2: Crush before the client calls commit

If the server crushes before the client calls commit, then the server will lost all the chunks it stores in memory. In order to prevent that, the server will compare the chunks the client want it to write to the disk during each commit request. If the server notices that it does not have all the chunks it needs to write, it will ask the client to re-transmit the chunks.

The client will keep all the chunks it wants to write before it get ACK from the server that all the chunks are successfully written to disk.

1.4 Crash consistency

For faster lookup, every time a new file handle is generated we populate it in a Hash Map, which provides a translation from file handle to path. On the client-side, we have a recursive lookup based approach where the file handle is extensively searched for starting from the root node and pass them to the server. If the server finds it does not exist, then generate a new file handle and populate it in a Hash Map. We maintain a persistent copy of this map by force writing them to the disk(a JSON file). 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.

2 Performance

There's two versions of the server:

- V1 for basic implementation.
- V2 for optimization using COMMIT.

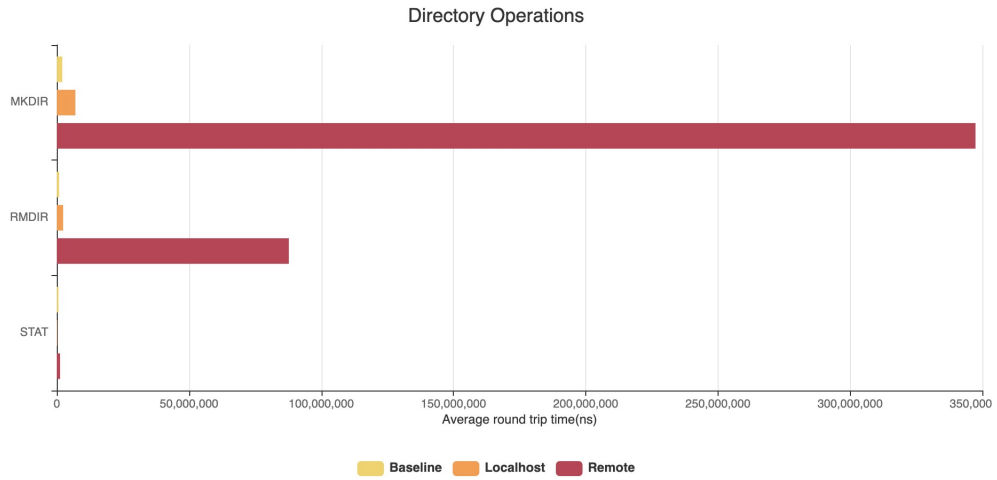

Figure 1: Performance of directory operations

2.1 Running Environment

We prepare three environments for testing: baseline, localhost, and remote. The baseline means executing file operations on the machine based on the original implementation. The localhost means running our system, both client and server on the same machine and executing file operations. The remote means running server on the different machine(EC2 instance) and executing file operations.

2.2 Evaluation

We evaluate three different directory operations: MKDIR, RMDIR, and STAT; and five file operations: OPEN, READ, CLOSE, UNLINK, and FSYNC. In order to measure commit performance, we evaluate WRITE in V1 and V2. Each directory and file operation is executed 100 times, and the average round trip time is recorded as Figure 1 and Figure 2 shows. Each writes operation transmits a 1000 bytes buffer. The average round trip time of writing is shown in Figure 3.

2.3 Multi-client Scenario

We use a single process to handle all requests. All request will be arranged in a single sequence. Therefore, one request(especially write) is processed at a time. It is safe for the multi-client scenario. This design is easy to build but has low performance and may bring high latency. All clients will be blocked when a long request is arriving. In two clients writing, the average round trip time of the write operation is 88566814ns for transmitting 1000 bytes message

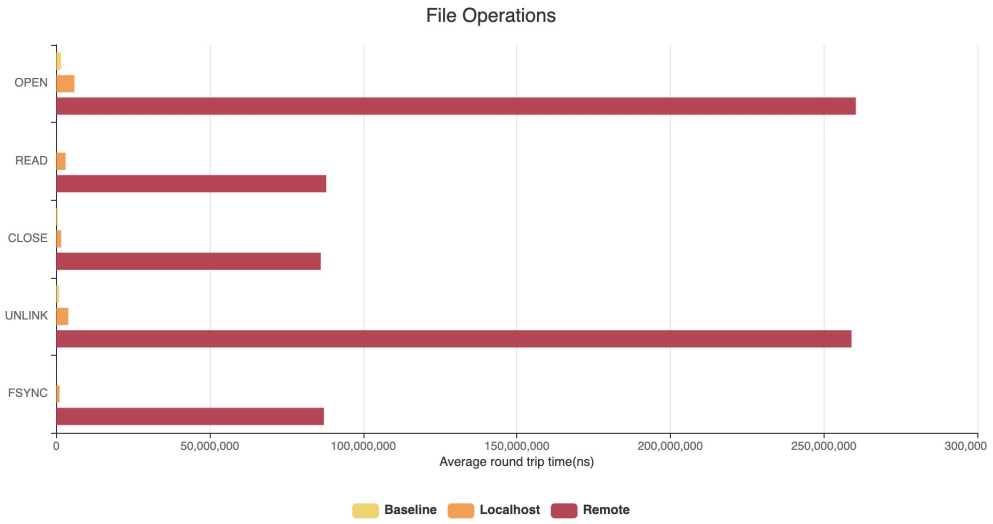

Figure 2: Performance of file operations

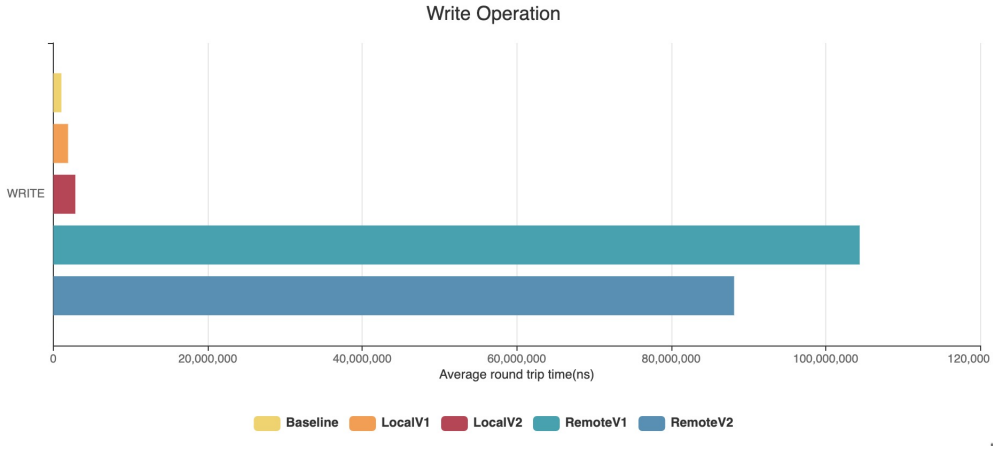

Figure 3: Performance of write operation

which takes more 416893ns than single client writing. We consider improving our server design by using a thread pool to handle requests. And mapping multiplexing requests onto some number of threads (reusing threads between requests). For each update, we assign a mutex for each file and make sure that one file is processed by one thread at a time.