## Introduction and Problem Statement

This project is a client/server file system where the blocks for a client are stored across N servers where N can be specified by the user when they initialize the client. This file system also implements RAID-5 which distributes both the data and parity blocks across multiple servers to evenly balance the load across the servers.

Through the redundancy implemented by RAID-5 this filesystem can handle corrupt blocks and fail-stops on a single server. By detecting these failures and handling them by using the parity blocks and XOR operations, the filesystem can update and recover blocks from a server that failed. Furthermore, there is a ‘repair -server\_ID’ command that, once the server is back up and running, can recover all the blocks for that server and return the system to its usual functionality.

## Design and Implementation

1. Memoryfs\_server.py

In memoryfs\_server.py I added command-line arguments “-sid” and “-cblk” to input the server\_ID and corrupt block number, respectively. The server\_ID isn’t used for much other than to display the integer representing the server, but the corrupt block is important because it specifies which block to emulate decay for.

The main modification to the code from Design 3 was implementing the checksums and emulated decay. For checksums I used a dictionary (self.checksums{}) which I update on every Put() with the md5 checksum of the data. Then, in the Get() function I retrieve the checksum from the dictionary and check if the block number is equal to the corrupt block number. If it is, I overwrite the block data with corrupt data (0xFF for all the bytes), and then recalculate the checksum and store it in a variable. The “test” is then compared to the checksum stored in the dictionary and if they’re not equal then a checksum error is returned by the server to the client. If they are equal, then the data is returned as usual.

Graphical user interface, text, application

Description automatically generated

Figure 1. Put() and Get() functions in memoryfs\_server.py.

1. Memoryfs\_client.py

The bulk of this project was implemented in the client and I’m going to explain the changes made in subsections corresponding to those described in the canvas assignment.

1. Handle N Block Servers

To handle N block servers, I simply store the port arguments into an array and then iterate through each server to initialize an array of servers with corresponding ports.

Text

Description automatically generated

Figure 2. Initializing N block servers based on command-line arguments.

1. Distribute Blocks Across Multiple Servers for Put() and Get()

To implement RAID-5 I needed to map virtual blocks to physical data and parity blocks and followed the suggested implementation of creating two separate mapping functions.

For mapping to data blocks I handle everything pretty much the same as RAID-4 with taking the modulus to get the server and using division to get the block number for that server. The main difference is I check if the server I’m mapping to is greater than or equal to the server where I’m storing the parity for that block. If it is then I just need to add 1 to the server\_ID and that will ensure the mapping for the rest of the blocks in that row will be correctly mapped as well.

Mapping the parity blocks was just about keeping the same mapping for the physical block as in the data mapping, but then adjusting the server\_ID mapping where I basically use the physical block number instead of the virtual block number since that’s what affects which server the parity block is stored on.

Graphical user interface, text, application

Description automatically generated

Figure 3. Helper functions used to map virtual data and parity blocks to physical data and parity blocks.

The blocks are stored and retrieved with Put() and Get() like before, but now these functions use the helper functions to translate the virtual blocks to a server-block pair and call ServerPut() or ServerGet() which handle the extra logic of generating parity and handling failures when calling a server’s Put() and Get() functions.

Graphical user interface, text, application

Description automatically generated

Figure 4. Modified Put() and Get() functions that take in a virtual block number, translate it to a server-block pair, and then call the ServerPut() and ServerGet() functions for that server-block pair.

1. Implement the Logic to Deal with Failures

The first type of failure is a corrupt block which means the block decayed and the data there is no longer valid which I emulated with the -cblk command as previously mentioned. If the data read from a block is equal to CHECKSUM\_ERROR, then I log that that block is corrupt and recover it using the self.RecoverBlock() function and return the recovered data.

The second type of failure is a failstop which occurs when a server is disconnected while the client is using it. I use try-except clauses to detect when a server crashes and after that I check if the server equals FAILED\_SERVER so that I don’t have to wait for the RPC to timeout every time when I already know that server is offline.

Handling a failstop in ServerGet() is straightforward – you read from the server in a try clause and if it’s failed then in the except clause, I set the global FAILED\_SERVER flag (default value is -1) to the integer value of the server, log the failure, and use self.RecoverBlock() to recover the data from the failed server.

Handling a failstop in ServerPut() is a bit more complicated since you also must worry about generating parity. First, I generate parity for the virtual block and then I attempt to store that parity in the try clause. If that fails then in the except clause, I set FAILED\_SERVER = parityServer and log the failure. If it passes then I attempt to store the new block data and if that fails I set FAILED\_SERVER = dataServer and then log the error.

It makes sense to check the parity first because if the parity is stored successfully but the Put() to the data block fails, I already generated the parity so I don’t have to do anything else. If more than 1 server could fail at a time then I would have to change this implementation but since the parityServer and dataServer of a virtual block are always different and we’re assuming only 1 server can fail at a time this is fine.

Graphical user interface, text

Description automatically generated

Figure 5. ServerPut() implementation.

Text

Description automatically generated

Figure 6. ServerGet() implementation.

My RecoverBlock() function is really simple since I realized whether the block you’re recovering is parity or data, you still end up doing an XOR of every server except the one you’re recovering for.

For GenerateParity() I just translate a virtual block into (server, block) pairs for the data and parity blocks and then XOR the old data with the new data and XOR the result of that with the old parity to generate the new parity which I then return since I handle the actual storing of the parity in ServerPut().

Text

Description automatically generated

Figure 7. RecoverBlock() and GenerateParity() functions used in ServerGet() and ServerPut().

1. Memoryfs\_shell\_rpc.py

The only real changes to the shell were adding the command-line arguments for the numbers of servers and specifying ports, as well as the “repair” command for the interpreter and the corresponding function which calls a repair function in the client.

Text

Description automatically generated

Figure 2. Command-line arguments for the number of servers and the ports.

Text

Description automatically generated

Text

Description automatically generated

Figure 3. Interpreter command for repair function and the shell repair function which does some error checking and calls a client-side repair function.