EEL 5737 HW 5 Report

Rohan Malik

Introduction and problem statement

The task for this assignment was to implement a RAID 5 filesystem using a local client and multiple servers. The client is responsible for managing the filesystem and remote RPC server connections. Each RPC server acts as a remote disk with data blocks the client can modify with Put() or Get() calls.

The python files memoryfs_client.py and memoryfs_shell_rpc.py make up the client portion of the code. The server is implemented in memoryfs_server.py.

Design and Implementation

Traditional vs RAID 5

In a traditional filesystem without any redundancy, each block is stored sequentially on a single disk, or server in this case.

RAID 5 spreads the data blocks and parity information across multiple servers. The below table visualizes how the blocks are stored on each server. Each column is a different server and the rows are the physical block numbers of a server. In this case, the client would have access to 16 blocks for data and parity and each server would host 4 physical blocks. The blocks seen by the end-user are referred to as virtual blocks.

server:	0	1	2	3	4
	0	1	2	3	P0
	4	5	6	P1	7
	8	9	P2	10	11
	12	P3	13	14	15

Parity data is used to recreate the data in the case of a loss or corruption. It is the combined XOR of all the data blocks stored in a row of the RAID filesystem. A single block can be recreated by XORing the remaining blocks in the same row. For instance:

$B(lock)\ 0 = B1 \otimes B2 \otimes B3 \otimes P0$

The parity block is updated every time a new block is written. The equation above can be used to create a parity block, but I only used that method when the data or parity servers where disconnected or corrupt. Another way to create a parity block is:

$P0(new) = B0(new\ data) \otimes B0(old\ data) \otimes P0(old)$

memoryfs_server.py

The server was modified to store a checksum for each block of data and check for a corrupt block during a Get() or RSM().

The hashlib library is used to produce a 128-bit MD5 hash for each block. The checksums are stored in a dictionary in the DiskBlock class. The checksum is updated on a Put() and verified against the hash of the actual data on a Get(). If the data block hash and checksum do not match, Get() returns an error.

The optional command line argument '-cblk <BLOCK NUMBER>' was also added, so that a corruption could be simulated. Get() returns an error if the set block is requested.

memoryfs_client.py

<u>RSM</u>

Acquire() and Release() were modified to immediately return without executing any code. In practice, this removed RSM() from my implementation.

<u>Multiple Servers</u>

A dictionary is used to store a ServerProxy object for each server. The key is the server_id.

Put() and Get() Abstraction

The provided methods used in HW4 for Put(block_number, block_data) and Get(block_number) were converted to SinglePut(server_id, block_number, block_data) and SinglePut(server_id, block_number).

This allows there to be Put() and Get() methods that can use the virtual block numbers and call the SinglePut() and SingleGet() methods as needed for RAID operations. The new Put() and Get() also handle corrupt blocks and server disconnects.

Virtual-to-physical translation

Mapping functions were created to translate the virtual blocks to physical ones. The mapping functions were also the only parts of the code that needed to be changed to go from RAID 4 to RAID 5.

The equations used are:

server_id = virtual_block_number % (NUM_SERVERS – 1)
    ** if the parity_server_id <= server_id, then server_id = server_id + 1

block_number = row = virtual_block_number // (NUM_SERVERS – 1)

parity_server_id = (NUM_SERVERS – 1) – (row % NUM_SERVERS)

** The server_id sometimes must be shifted to the right to account for the parity block.

The methods are:

getBlockInfo(virtual_block_number)
    returns the server_id and block_number for a given virtual block

getParityServer(virtual_block_number)
    returns the server_id of the parity block for a given virtual block

Error Handling

Corrupt Block

An error from a SingleGet() indicates a corrupt block. An error is printed and the other blocks in the same row are used to produce the corrupt data.

The only time nothing is done after a corrupt block is detected is when the new parity block is being created in Put(). Put() writes the new data first and then writes the new parity block. If the program experiences an error when it is working on the parity block, then it is guaranteed that the data block was written without an error and the parity block is not necessary. It is guaranteed because there will only ever be a single failure tested.

Disconnected Server

When a server is detected as disconnected, the ConnectionRefusedError is caught and a SERVER_DISCONNECTED message is printed out. The data is recovered using the same logic as a corrupt block.

To make sure that the server is not continually accessed, the ServerProxy object is removed from the block_server dictionary. Put() and Get() check if a server_id is not in the dictionary and raise a ConnectionRefusedError to simulate it being disconnected. This guarantees that the client will not be able to access the server until it is re-connected and repaired.

Repair

The repair method takes in a server_id and repopulates the server with data. A new ServerProxy object is created and added to the block_server dictionary. The program then loops through each row of the filesystem and populates the server with data using the other blocks.

memoryfs_shell_rpc.py

Command line arguments -startport <PORT> and -ns <N> were added to create the multiple ServerProxy objects in the client. PORT is the port of the first server. The port numbers are consecutive after the first. N is the number of servers.

A repair method also calls the memoryfs_client repair method. The interpreter is modified to add the method.

A few test and debug programs are also added to the shell:

test1 -> creates 4 files, appends 128 bytes of data, cat the files
test2 -> creates 20 files, appends 640 bytes of data, cat the files
test2_read -> cat 20 files named file<X>, where X is 0 through 20
test3 -> calls showblock on every block of the filesystem. Parity blocks are not included.

showparityblock(n) -> showblock for the parity block associated with virtual block n

Evaluation

I tested and evaluated my code as I implemented features. I added one feature at a time before moving on to the next. I added the showparityblock function and test3 to verify my RAID configurations. Calling showblock on every

Performance Analysis

I tested the performance my RAID 5 implementation using a small and a large filesystem with 5 servers. I added a request counter to the server and had it print to the server console.

The small filesystem I used was -nb 256 (server) and -bs 128. The small filesystem was too small to create a large enough file and did not provide much data. I created 4 files each with 128 bytes of data. The requests between servers were not close to B/N, where B is the number of blocks in a file and N is the number of servers. The table below shows the requests from running test1, the command that creates and reads the files.

	Server 0	Server 1	Server 2	Server 3	Server 4
test1	26	18	58	122	12

The large filesystem had 2048 blocks per server and 512 byte blocks. I created 2 files that each used 5 blocks. Each file stored 2049 bytes of data. When I cat one of the files, I expected to see 1 request on each server. Below are the requests from reading the files:

	Server 0	Server 1	Server 2	Server 3	Server 4
file1 requests	1	1	1	6	2
file2 requests	1	1	2	6	2

I expected to see 1 request per server, because the files had 5 blocks and there were 5 servers. The requests were close to expected. I think for a larger file you would start to see the requests per server even out, but there are some blocks that will always get more requests. The blocks where the free bitmap and inodes are stored will be called more than the others.

Reproducibility

The program can be run with 4 to 8 servers and 1 client.

The command to start the client with 5 servers starting at port 8000 is:

```
python3 memoryfs_shell_rpc.py -cid 0 -startport 8000 -ns 5
```

The command to start the first server is:

```
python3 memoryfs_server.py -p 8000 -nb 256 -bs 128
```

You would then start 4 more servers at ports 8001, 8002, 8003, and 8004.

Conclusions

All the features called out in the HW 5 assignment documentation have been implemented and tested. Overall I enjoyed this assignment and diving a bit deeper into fault tolerance and RAID. It was challenging, but I felt that help was accessible through class and office hours.