Everett Periman
December 3, 2021
EEL 5737 – EDGE
Professor Renato Figueiredo

Homework 4

Problem Statement

Data loss is a problem that is seemingly unavoidable and is something we must learn to live alongside. There are several solutions and approaches to this issue, but the author will focus on a distributed RAID network. Work to explore this topic as well as how to deploy RAID-5 redundancy at a server-client level will be discussed throughout this paper.

Introduction

This project was conducted through several milestones to help aid in the buildup to the final assembled code base. These milestones included testing for multi-server compatibility (RAID-0), implementing checksum data validation, general RAID-5 functionality, RAID-5 corruption, fail stop case, and a repair case. Each of these steps was demanding but each yielded an important lesson and was valuable to building up to the final project.

Design and Implementation

Section 1: Checksum Data

The checksum information was stored using the 128bit md5 algorithm. This algorithm is a local library to Python and is accessible through hashlib. This checksum was used to validate the data that was requested from and individual server. In order to implement this a data structure was created in the server to store the checksums of all of the blocks in the server. Upon receiving and RPC Get call from a client the server would Get the desired block, ensure that the checksum of this data matched the previously recorded one and was set to the server. If these checksums did not match or if a cblck designator was flagged (used to simulate block data decay) the server would return -1 as it's Get response to notify the client of the issue.

Section 2: Mapping Virtual Blocks to Physical Servers

All of the virtual blocks that the client makes references too must be translated in order to work correctly with the RAID-5 server array. In order to facilitate this translation a function was designated for this work and acted in a round robin pattern. The RAID array is split into stripes or rows which span across the same height on each disk, additionally each stripe features a parity block and the remaining blocks store data. To calculate the current stripe of a virtual block the block is divided by the number of servers -1 due to there being n-1 server capable of storing data per stripe (stripe = vblck // n-1). The parity server is determined by evaluating the current stripe parity = (n-1) – (stripe mod n), this formula was chosen for this crossing round robin effect and

along with the data block formula does not give priority to underutilized servers. The final formula is to calculate the appropriate server send the data which is calculated using server = vblck mod (n-1) one again derived for its revolving coverage of the servers. A check is also ran to ensure that the server and parity servers to not interfere with one another, and if they do then the server value is iterated up by 1 to not clash.

Section 3: Reads

Since this codebase is adapted from that of HW3 many of the underlying principles remain the same. The mapping function was created such that over the course of a substantial period each server will be server B/N requests. When the client only requires one block it will only Get one block, when the client requires several, it will Get several but do to the revolving fashion of the mapping function these requests will end up at several different servers. If a Get request comes back as a -1 that means the data has been corrupted on the server side, so the Get method will try and rebuild the data from all other data drives the parity drive for that stripe. The GetRebuild() method will XOR all of the data from the other blocks in that stripe so that the corrupted block can be calculated and served back to the caller.

Section 4: Writes

The ideal case for every write to the server would require 2 writes, the first being to update the data block with the new information and the second being to the parity drive. In order to update the parity, drive two routes can be used, the first route takes the old data of the block, the data for the block, and the old parity data and xor's them together to create the new parity data. This is the method implemented by the UpdateParity() method. The next method utilizes the data of all the other data blocks in that stripe and the old parity data and the new block data to create the new parity data, this method was not used as it suffers in performance (O(n) vs O(2) of the first method) but it is also covered by the corrupted block methodology discussed in Section 3. The author intended in including both methods in the code to cover the situations that a parity drive had failed, but the GetRebuild() method will automatically serve this function for the first method if the parity drive has failed. If a server has failed and the network is in a fail-safe mode then there will only be 1 write overall, depending on whether it was the parity block for that stripe the client will try and request to write the parity and new data for that strip. One of these will fail but by updating the other one there is still enough information to eventually recover all of the data.

Section 5: Repair

In order to repair the server the method Rebuild() was created to facilitate this task. The first task in rebuilding is to reconnect and refresh the connection between the client and server. Once this is done then the client will request a GetRebuild() for each of the physical blocks that the server has. This will iterate through each block and recover the missing data using the rebuilding method mentioned in the Reads section. Now that the server has been regenerated it has also been flagged as a good server and will begin to receive requests again to balance the loads.

Section 6: Failures

These topics have been discussed throughout the other sections but it is worthwhile to cover them more in depth. The main failure that the server can undergo is the complete loss of a server. This requires much more work for each server until it is back as this system supports fail-safe.

Now every time the server writes new data only one of the writes will succeed and the time to update the parity data will greatly increase. Since either one of the data blocks or the parity block is lost, this missing data is recalculated and used in the Put to the server. The next largest failure is that of a corrupted file on the server. The checksums ensure that the server is aware of whether it has good information or not, and to allow the server to give the client a notice that it must manually rebuild the information. The client upon this notice and receipt of the new data will then try and Put the new data back to the server fixing the error.

Evaluation

Section 1: Load distribution evaluation

The load of the servers was simple to evaluate, owing this fact due to the nature of the mapping function as well as to the nature of the system which the client uses to obtain the next block for it to use. The client will go typically go sequentially in assigning new block numbers unless one has been freed up. Combining this with the round robin approach of the mapping function, data is evenly distributed to the servers theoretically.

This theory was tested by using two of the data dump files from previous assignments filesystem_hw1.dump and bigfs_nb2048_bs512_ni128_is32.dump as they are a constant data set, and they introduce variable block counts (256 and 2048), and size of blocks (128 and 512). These data sets are used to test the load distribution as shown in Figures 1 – 4. To collect this information each server recorded how much new information was inserted after the load function was called. These figures demonstrate the distribution of the block values between the servers for each of the loading sessions. Figures 1, 3, 4 all show that the blocks are more regularly spread and is a balanced load. Figure 2 seems to be quite anomalous and further investigation into such an odd spread would be worthwhile.

Section 2: Throughput evaluation

To test the loading throughput of all of the server's data was collected on the amount of time taken to load a filesystem for each server with the large and small file systems. Figure 5 showcases the time to load the filesystems for each test case. The load times are drastically higher for the RAID setups where the small fs took ~.4s for 4 servers, ~15.9s for 8 servers, and just .1s for the small server. Though the load time was much longer for the small FS the large FS was a lot more taxing with ~.4s for 4 servers, 15.9 for 8 servers and .5 for 1 server. This rate of scaling shows that increasing the amount of data to be stored is much more taxing on the single server than a RAID setup. On the large file system, the single server took 550% time to load, the 4 servers 4.6% and the 8 servers .5%. This slow scaling might be indicative of the overhead of parity writes, where RAID servers would have written to many of these blocks already a single server would write much less for the same amount of useful data.

Server	0	1	2	3
Block Count	257	256	255	264

Figure 1: Small File System 4 servers

Server	0	1	2	3	4	5	6	7
Block Count	66	66	89	162	164	160	161	164

Figure 2: Small File System 8 servers

Server	0	1	2	3
Block Count	2046	2050	2053	2051

Figure 3: Large File System 4 servers

Server	0	1	2	3	4	5	6	7
Block Count	503	502	503	501	501	500	541	553

Figure 4: Large File System 8 servers

	Write Time (s)
Small File 1 Server	0.106
Large File 1 Server	0.580
Small File 4 Servers	0.421
Large File 4 Servers	0.441
Small File 8 Servers	15.821
Large File 8 Servers	15.905

Figure 5: Time to load file sets

Reproducibility

Section 1: Testing CBLCK

Start 4 Servers

```
python3 memoryfs_server.py -nb 2048 -bs 512 -port 1080 -sid 0 -cblck 3
python3 memoryfs_server.py -nb 2048 -bs 512 -port 1081 -sid 1
python3 memoryfs_server.py -nb 2048 -bs 512 -port 1082 -sid 2
python3 memoryfs_server.py -nb 2048 -bs 512 -port 1083 -sid 3
```

Start 1 Client

```
python3 memoryfs_shell_rpc.py -nb 256 -bs 128 -port0 1080 -port1 1081 -port2 1082 -port3 1083 -cid 1 -ns 4
```

Scan Server 1 Terminal for “Bad Block #”

```
127.0.0.1 - - [01/Dec/2021 20:58:57] "POST /RPC2 HTTP/1.1" 200 - Bad block 3
```

Section 2: Fail Safe and Rebuild

Start 4 Servers

```
python3 memoryfs_server.py -nb 256 -bs 128 -port 1081 -sid 0
python3 memoryfs_server.py -nb 256 -bs 128 -port 1081 -sid 1
```

```
python3 memoryfs_server.py -nb 256 -bs 128 -port 1081 -sid 2
python3 memoryfs_server.py -nb 256 -bs 128 -port 1081 -sid 3
```

Start 1 Client

```
python3 memoryfs_shell_rpc.py -nb 256 -bs 128 -port0 1080 -port1 1081 -port2 1082 -port3 1083 -cid 1 -ns 4
```

Load files in terminal window of client

```
load filesystem_hw1.dump
```

Stop 1 Server - CTRL+C in the terminal of server **3**

```
python3 memoryfs_server.py -nb 256 -bs 128 -port 1081 -sid 3
```

Optional: In the client modify the file system as you desire

Restart 1 Server

```
python3 memoryfs_server.py -nb 256 -bs 128 -port 1081 -sid 3
```

Repair Server in the client terminal

```
repair 3
```

Check File System Integrity

Note: With this check I have noticed that there is a bug in my code that causes faults if the first server I repair is 0 or 1 in a 4 server setup. For example in this test you disable and repair server 3 it will work, and then if you proceed to disable server 1 and repair it will also work. If you had disabled and repaired server 1 first then that would not work. I could not determine the cause of this bug but I would appreciate any feedback you have for it.

Conclusions

Overall, this project provided a lot of insight into the intricacies of RAID setups and their deployment in a server system. Moving from RAID-0 to RAID-5 while integrating additional features provided a plethora of challenges but helped conceptualize theory from class. The experiments were quite interesting and proved that the round robin distribution worked quite well. The time to load the servers was also quite interesting, since the Puts and Gets are still serialize but not have to be navigated between servers it makes sense that the initial setup time is much longer than for the single server setup. The scalability is what surprised me the most and might be something I investigate more after this project, the single server scales so much worse than the RAID systems meaning most of the delay was most likely not in loading more files but in overhead and handling so many more RPC call. Though I do believe that my code might fail

some of the tests I do firmly believe that I have gained a thorough understanding of the underlying topics and their usage.