Course: EEL-5737

Course Name: Principles of Computer System Design

Final Project (Fall 2019): RAID 5 Implementation

Submission Date: 12/06/2019

Submitted By:

Andrea Lardschneider

Md Jubaer Hossain Pantho

Objective:

The goal in this project is to distribute and store data across multiple data servers to reduce load by distributing requests across several servers. The design offers fault tolerance and increased aggregate capacity. The block storage follows the general approach described for RAID-5. Implementation of RAID-1 is not covered in this work.

How to Run?

To initiate the servers run the following command:

$ **python backChannel.py 4**

To run the client File system:

$ **python FileSystem.py 4**

This should generate our terminal.

Sample input format:

$ **mkdir /hello**

$ **create /hello/1.txt**

To write to a file: $ **write AbsoluteFilePath offset delay dataToWrite**

To read a file: $ **read AbsoluteFilePath offset readSize**

To move a file: $ **mv oldPath newPath**

To remove file: $ **rm AbsoluteFilePath**

Description on Testing Method:

We wrote a separate python script to check the functionality of our design. The script (***testTimeScript.py***) is added to the submitted files. This file is different than the one that we used to generate the command prompt. In this script, we create a folder and create a file within it. Then we write 4KB of data to that file and read it back. If the read data matches with the written data, a print statement notifies the user. Otherwise a fail statement is printed. Because of the huge size of the data. The read operation does not print the data. It has been commented out. The timing for read and write operation is also printed. We have provided adequate print statements to describe how data blocks are getting distributed at run time. The move statement and the remove statements are also tested with this script. However, they are commented out to highlight the read and write

operation. User can uncomment it to check the result. While reading we added a 5 seconds delay as it is instructed on the project description. The write parity delay (instructed in the project description) is provided as an argument from the user.

To run the test script run the following command:

$ **python testTimeScript.py 4**

Implementation Overview:

We designed our RAID-5 storage systems for 4 servers. The current implementation supports **only** 4 servers. In our design, the blocks are distributed across multiple servers. The parity blocks are stored in a distributed manner as it done in RAID-5. When only one server is down, the design can successfully detect corrupted block and correct it. This allows our system to keep operating with one corrupted or down server. All the operations related to RAID-5 (i.e. block distribution, virtual node generation, data integrity check) are performed on the client_stub.py file with minimum modifications on the upper layer. The contribution of this work is given below:

- Implemented RAID-5 on 4 servers. The data and parity information are distributed across servers, at the granularity of the block size.
- Performed MD5 checksums on data block and stored within the data block to check integrity of the data.
- Evaluate the performance of our design with an implementation with a single server (Homework 4).
- Utilize cached server failure information to reduce server access.

Implementation Details:

In this section we will explain different parts of our design in detail. As we mentioned before, the data and parity information are distributed across servers at the granularity of the block size. In our 4-server implementation, parity is distributed according to the following image. The servers are indexed as 0, 1, 2 and 3.

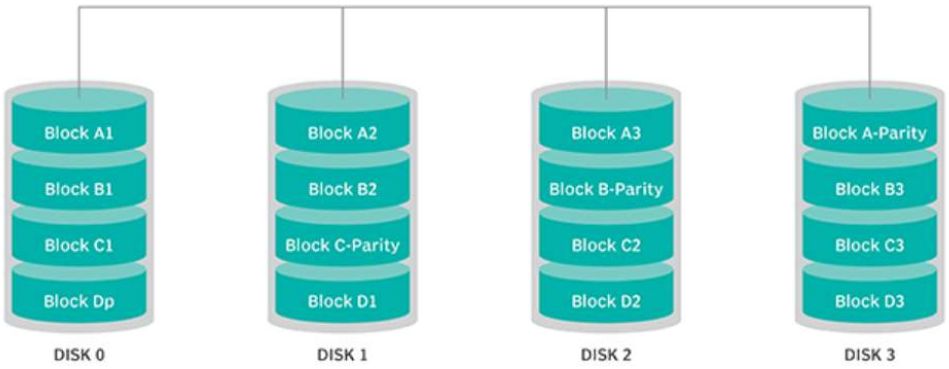

Figure 1: RAID-5 parity block distribution.

For each **write** operation of a block we performed two reads and two writes. We calculate the md5 hash function of the data and store it at the end of the data block. This allows the design to validate the integrity of the design by checking the checksum. Therefore, when no server is down each **read** operation only require one **read** operation. When the server we want to read from is down or corrupted, the data gets reconstructed by reading the other three servers and xor – ing it.

Inode Structure:

The Inode table is stored in all the four servers. This means each server has the identical copy of the Inode table. When we request a new Inode, we request only from one server. However, the update function updates the Inode table in all the servers. This way even if a server is down, we are able to fetch the inode table from a different server. Within an Inode we don't store the physical block numbers. Instead, we store virtual block numbers, which are translated once an operation on the FileSystem is done. The InodeLayer interacts with the lower layer with this virtual inode number. The virtual inodes can be from different servers. However, the Inode layer does not have any information about it. This is translated in lower layer. The corresponding parity blocks are not stored within the Inode.

MD5 calculation:

The checksum is calculated by using md5 with length 16 Bytes. The block size written and fetched from the server is (256+16) = 272 Bytes. Within the client_stub, the last 16 Bytes are extracted and only the 256 Bytes of data are sent to the upper layers. For the upper layer the block size is 256 Bytes.

Virtual Block Numbers:

Virtual block numbers are generated on the client_stub. The client_stub class contains a list that maps the virtual blocks to a physical block and holds the information of the parity block. A virtual block number only corresponds to a single physical block number in a particular server. For example, virtual block number 0 refers always to server 0, number 1 always to server 1 and so on. In our implementation, the InodeLayer uses virtual block numbers instead of the physical data block numbers. While reading or updating data blocks, requests are sent to the lower layer with this virtual block number. Within the client_stub then, we implemented a block_number_translate method, that translates the virtual block number to the corresponding server number and the parity server.

```
def get_new_virtual_block(self):

    for i in range(0,config.TOTAL_NO_OF_BLOCKS):
            if(self.virtual_block_numbers[i] == False):
                    self.virtual_block_numbers[i] = True
                    return i
    print("Memory: No valid virtual blocks available")
    return -1

def block_number_translate(self, virtual_block_number):

    packof3index = virtual_block_number/3
    vnumbers = virtual_block_number%3
    parity_server = 3 - (packof3index%4)

    if(packof3index % 4 == 0):
            server = vnumbers
    elif(packof3index % 4 == 1):
            server = vnumbers + vnumbers/2
    elif(packof3index % 4 == 2):
            server = 2*vnumbers - vnumbers/2
    else:
            server = vnumbers + 1
    return server,parity_server
```

Corrupt Data Block Method:

We corrupted the data by xor the data block with all ‘a’s. This is done in the get_data_block function. The corruptData () method simply makes the state variable false. Before we send the data back to the user, we corrupt it if this state variable is False.

```
def get_data_block(block_number):

        passVal = pickle.loads(block_number)
        retVal  = filesystem.get_data_block(passVal)
        if(state == False):
                corrdata = (config.BLOCK_SIZE+16)*'a'
                retVal = ''.join(chr(ord(a)^ord(b)) for a, b in zip(retVal , corrdata))
        retVal  = pickle.dumps((retVal,state))
        return retVal
```

Script Modification:

We modified the given script to better implement the RAID-5. Basically, we implemented our design on our homework-4 implementation with the client and server stub files provided in the project template. We adapted the inode request and update structure in a way that all 4 servers always contain the exact same Inode table. In the final project, the write function takes an additional input as delay. We used it in the client_stub to add delay on parity write. All layers above accommodate the delay argument in the write method. The InodeLayer requests virtual Inodes instead of a physical Inode. This is explained above. Besides, the file system initialization function takes the server number as input. Because, our original plan was to provide support for variable number of servers.

Performance Evaluation:

We tested our RAID-5 storage system with a data size of 8KBytes. We make a directory in the root directory and create a text file within it. Then we write 8KBytes of data to that file and read it back from it. 8 Kbytes of data requires 16 blocks (512 Bytes) to store. These blocks are distributed within the servers as (server0 = 5 blocks, server1 = 4 blocks, server2 = 4 blocks, server3 = 3 blocks). These write require an additional set of writes on the parity server. The parity writes are distributed as follows: (server0 = 3 parity updates, server1 = 3 parity updates, server2 = 4 parity updates, server3 = 6 parity updates). While reading, if no server is down, it will require the same number reads as data writes (server0=5, server1=4, server2=4, server3=3). This is because the integrity is checked directly from the data block. And there is no need to verify it by fetching other blocks.

However, if a server is down or corrupted, it changes.

Let's say server 0 is down and we would like to write to it. In order to compute the new parity data, we need to read the old data and the old parity as mentioned above. Since server 0 is corrupted or down, we can't get the old data and therefore we need to reconstruct. The reconstructed old data is then xor-ed with the new data, before being xor-ed with the old parity data. The result is the new parity data which is written on the parity server. This procedure allows us to "write" data on a server which is down or corrupted.

This procedure slightly changes if the write refers to the parity server. In that case, the old data is updated with the new data and no parity block is computed.

While reading from a server which is down or corrupted, a similar procedure is invoked. By reading from the other three servers and xor-ing the data together, we can reconstruct the data.

That means, for our specific test case, that we have five (each for a block in server0) additional reads per server to reconstruct the 5 blocks of servers 0 which is down or corrupted.

This is shown in a tabular form in Table I and II.

(While generating these tables. The read operations to fetch old data are not counted.)

Data Size: 8Kbyte

Table I: When all servers are up.

Servers	Write Data	Write Parity	Read (While reading the data)
0	5	3	5
1	4	3	4
2	4	4	4
3	3	6	3

Table II: When server 0 is corrupted.

Servers	Write Data	Write Parity	Read (While reading the data)
0 (down)	5	3	5
1	4	3	9
2	4	4	9
3	3	6	8

Improvements:

In our design, we made some improvements to reduce the number of server accesses. Here, the client_program remembers the faulty server and avoids server access in the consecutive sections. In the last table, the read operations to fetch old data is not counted. We generated the following table by directly counting the access request on the server port. Also, while reconstructing data from other blocks, if a block is empty, it is never fetched from the server.

Table III: Server access requests when all servers are up

Operation	Server 0	Server 1	Server 2	Server 3	Avg access
Write	22	15	17	19	18.25
Read	11	4	4	3	5.5

If we look at the table III, we can see that server 0 has more request compared to the other ones. This is because in our design we request a new inode from one server only (In this case, that is server 0). And, later update inode table on all the servers. That is why server 0 handles more requests.

In the next table we show the server access requests when server 0 is down for the same 8Kbytes of data. We assume that server 0 is corrupted sometime during write operation.

Table IV: Server access requests when server 0 is down

Operation	**Server 0 (down)**	**Server 1**	**Server 2**	**Server 3**	**Avg access**
Write	0	15	17	19	16.25
Read	0	8	9	7	6

Looking at table IV, we see the average access of servers decreases for read. This is because, while updating the Inodes, and writing our design skips server 0 when possible because it is down. This is also true while reading the data. Server 0 receives 0 requests, since the client_stub skips the down server. Therefore, the average access for read only increases a little even with the reconstruction.

At the end we compared our design with a single server design. The results are shown in Table V and VI.

Table V: Server access requests comparison

Operation	**Raid-5 (Avg server access)**	**RAID-5 with a server down**	**Single Server**
Write	18.25	16.25	37
Read	5.5	6	20

In table V, we can see that even with a server down, the average server access time is lower then in a design with a single server. Therefore, the load per server has been reduced, which was one of our goals.

Table VI: computation time comparison

Operation	Raid-5	Raid-5 with a server down	Single Server Design
Write	0.108s	0.105s	0.029s
Read	0.049s	0.0856s	0.0144s

While generating data in Table VI, we ignored the read delay of 5 seconds on the server. Similarly, the parity write delay is ignored. In Table VI, we can see that a single server design is faster than the RAID-5 design. However, in this work the memory unit is a simulation model. Requests are not pipelined. And fetching the memory block is not a bottleneck. In an efficiently implemented actual design of RAID-5, computation time for large read write operations should outperform a single server design.