Course: EEL-5737

Course Name: Principles of Computer System Design

Final Project (Fall 2019): RAID 5 Implementation

Submission Date: 12/04/2019

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.

**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**

**Implementation Overview:**

We designed our RAID-5 storage systems 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 a server down. All the operations related to RAID-5 (i.e. block distribution, virtual node generation, data integrity check) is 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 Detail:**

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.

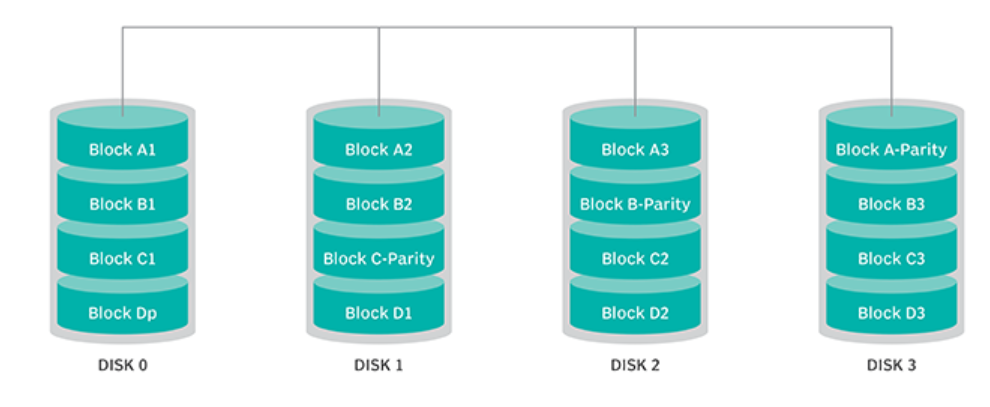


Figure 1: RAID-5 parity block distribution.

For each **write** operation of a block we performed two reads and two writes. We calculate the md5 hash function of the data and stored 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 a server is down, **read** operations get affected only if the data block is stored in the corrupted block. In that case, the corrupted data is reconstructed by making three additional **read**.

**Inode Structure**:

The Inode table is stored in all the four servers. This means each server has the identical copy of the Inode table. While request an inode, we request only one server. However, the update function updates inode 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 the physical block numbers are not stored. Instead the virtual block numbers correspond to the actual block numbers are stored. The InodeLayer interact 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. Also, the Inode block array only contains data blocks. The corresponding parity blocks are not stored within the Inode.

**MD5 calculation:**

The checksum calculated by md5 is 16 Bytes. The block size fetched from the server is (512+16) = 528 Bytes. Within the client\_stub the last 16 Bytes are extracted and only the 512 Bytes of data is sent to the upper layer. For the upper layer the block size is 512 Bytes.

**Virtual Block Numbers**:

Virtual block numbers are generated on the client\_stub. The client\_stub class contains an 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. In our implementation, the InodeLayer request 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, we implemented a block\_number\_translate method that translate the virtual block number to the generate the corresponding server number and the parity server.

**Performance Evaluation:**

We tested our RAID-5 storage system with a data size of 8KBytes. We make a directory in the root directory and created a text file within it. Then we write 8KBytes of data to that file and read back 8 Kbytes of data from it. 8 Kbytes of data require 16 blocks (512 Bytes) to store. These blocks are distributed within the servers as (server0=5, server1=4, server2=4, server3=3). These write require an additional set of writes on the parity server. The parity writes are distributed as follows: (server0=3, server1=3, server2=4, server3=6). 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 we do not have to verify it by fetching other blocks.

However, if a server is down, let’s say server 0 is down. Nothing will change during the write operations. But while reading, server 1,2 and 3 will have five (each for a block in server0) additional reads to generate the result.

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

(While generating these tables. The read operations to fetch old data is 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 |

**Table III: When server 3 is corrupted.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Servers** | **Write Data** | **Write Parity** | **Read (While reading the data)** |
| 0 (down) | 5 | 3 | 8 |
| 1 | 4 | 3 | 7 |
| 2 | 4 | 4 | 7 |
| 3 | 3 | 1 | 3 |

In Table III, After the first write parity the server gets the knowledge that the parity server is down. And, for the consecutive blocks this information is used to skip the parity write.

**Improvements**:

In our design, we made some improvements to reduce the number of server accesses. Here, the client\_program remembers the faulty server and avoid 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 that block is never fetched from the server.

**Table IV: 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 IV data, we can see that server 0 has more request compared to the other ones. This is because in our design we request inode information from one server (In this case, that is server 0). And, later update inode information to 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 V: Server access requests when server 0 is down**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Operation** | **Server 0 (down)** | **Server 1** | **Server 2** | **Server 3** | **Avg access** |
| Write | 14 | 15 | 17 | 19 | 16.25 |
| Read | 0 | 8 | 9 | 7 | 6 |

If you look at table V, you will see that, 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.

We compared our design with a single server design. The results are shown in Table VI and VII.

**Table VI: 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 VI, we can see that even with a server down. Average server access is lower than a design with a single server.

**Table VII: 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 VII, we ignored the read delay of 5 seconds on the server. Similarly, the parity write delay is ignored. In Table VII, we can see that 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.