CS579 PoCSD - Project Report
Gokul Kesavamurthy
kesavamg@oregonstate.edu

1. Introduction and Problem Statement

This project focuses on implementing distributed storage with fault tolerance using a client and block server architecture via RPC calls. The system is designed to support a single client and up to *N* data block servers. Given the presence of only one client, the implementation ensures *before-after atomicity*, eliminating the need for RSM functionalities.

Each data block server has a storage capacity of n data blocks, and the client can now access an increased aggregate capacity of data blocks distributed across all servers. The distribution of these data blocks should be optimized to balance the load across servers, ensuring that no single server becomes a bottleneck and that data is equally shared for efficient load management.

The design is fault-tolerant, capable of handling the failure of up to one server. If a server goes offline, the fault is masked using RAID 5 principles. Additionally, the client is equipped with functionality to repair a crashed server, restoring it to full operational status.

Furthermore, the system addresses checksum errors reported by block servers. In cases of corrupted data blocks, the client leverages parity information to recover the corrupted block, maintaining data integrity.

2. Design and Implementation

2.1. Block Corruption Implementation

Under normal circumstances, data stored in the block server does not naturally decay. To simulate block corruption, a specific block number can be designated for corruption using the command-line argument *-cblk*. The block server reads this value from the command line and intentionally corrupts the data for the specified block when a *PUT* operation is performed. When the client calls GET for this block, a checksum error is triggered and returned to the client, simulating the effects of data corruption.

To implement this mechanism, 128-bit MD5 checksums are utilized. These checksums are generated using Python's hashlib module. For every PUT operation, the checksum of the data block is calculated and stored in a separate array. During a GET operation, the previously stored checksum is retrieved and compared with the

checksum of the current block retrieved from the block array. If the two checksums do not match, the block is identified as corrupted, and an error code is returned to the client.

This implementation ensures that block corruption can be simulated, detected, and appropriately reported, laying the foundation for further error-handling mechanisms in the system. When the client receives a corrupted block error, it initiates recovery steps to restore the corrupted block and mask the error. The recovery process leverages the parity information stored in one of the servers, following the RAID 5 approach.

The recovery mechanism involves performing an exclusive OR (XOR) operation on the corresponding block indices across all the servers, including the server holding the parity information. Since the XOR operation allows data reconstruction by canceling out known values, this process effectively isolates the missing data from the parity.

Here's how the recovery works:

1. The client retrieves the block data from the same block index across all other servers, including the parity server.

2. By performing XOR on these blocks, the client erases the contributions of all intact blocks from the parity.

3. The result of the XOR operation is the original data from the corrupted block, effectively restoring it.

This method ensures data integrity and fault tolerance by efficiently recovering corrupted blocks while maintaining system performance and reducing overhead. This same approach is used for recovering data for a crashed server as well.

2.2. Virtual-to-Physical Block Translation

2.2.1. Aggregate Capacity

In a distributed storage system with N servers, the client gains access to an aggregate number of blocks spread across all the servers. This arrangement provides a significant increase in storage capacity compared to a single-server approach. However, due to the implementation of the RAID-5

parity mechanism, the capacity of one server is reserved for parity information.

As a result, the effective aggregate block capacity available to the client is equal to the total blocks on (N−1) servers. This reduction ensures fault tolerance, as the parity information stored on one server enables recovery of data in case of a server failure.

To make efficient use of this distributed storage, a mapping mechanism is necessary to translate the client's block numbers to corresponding block numbers on the servers. This mapping should ensure proper distribution, parity placement and fault tolerance. This mapping mechanism is integral to maintaining the increased capacity, load balance, and fault tolerance of the distributed storage system.

2.2.2. RAID-5 Load Sharing

The mapping from virtual block numbers (as seen by the client) to physical block numbers (on the servers) is a critical aspect of the design. This mapping ensures an even distribution of load across N servers and aligns with the RAID-5 principle of distributing parity information across all servers.

The key considerations for the mapping mechanism are:

- **Even Load Distribution**:
    - Virtual blocks are mapped to physical blocks on N servers in a way that ensures no single server bears a disproportionate load.
    - Both data and parity blocks are distributed across all servers to balance access requests, ensuring smooth operation even under high-demand scenarios.
- **Distributed Parity:**
    - RAID-5 introduces the concept of distributed parity, where each stripe (group of data blocks) has its parity block stored on a different server.
    - This approach avoids concentrating parity blocks on a single server, thereby mitigating potential bottlenecks in access.
- **Fault Tolerance:**
    - By distributing parity and data blocks, the system can recover from the failure of any single server without impacting operations.

2.2.2.1. Mechanism of Distribution

- The data blocks and the parity is evenly distributed across all the server based on the below calculation:

```
def getServerBlockAndParity(self, block_number):
    datablock_per_stripe = fsconfig.NUM_SERVERS - 1 # Number of data blocks per stripe
    stripe_number = block_number // datablock_per_stripe  # Stripe number
    data_offset = block_number % datablock_per_stripe # Offset of data block within the stripe

    # Calculate parity server for this stripe
    parity_server = stripe_number % fsconfig.NUM_SERVERS

    # Calculate server for the datablock
    # Identify all disks involved in the stripe
    blocks_in_stripe = list(range(fsconfig.NUM_SERVERS))
    blocks_in_stripe.remove(parity_server)  # Exclude parity disk
    server_index = blocks_in_stripe[data_offset]

    logging.debug(f"block_number: {block_number}, server_index: {server_index}, server_block_in
    return server_index, stripe_number, parity_server # server, block_index, parity server for
```

2.2.2.2. Example Distribution:

For N = 4 servers, and four stripes, the distribution of virtual blocks and parity blocks are shown below:

Stripe	Server 1	Server 2	Server 3	Server 4
0	Parity	VB 0	VB 1	VB 2
1	VB 3	Parity	VB 4	VB 5
2	VB 6	VB 7	Parity	VB 8
3	VB 9	VB 10	VB 11	Parity

This rotation ensures parity and data blocks are evenly distributed, achieving balanced load and resilience.

By employing this strategy, the system maintains its efficiency and fault tolerance while achieving the primary goals of RAID-5: performance optimization and data reliability.

2.3. Fault Tolerant PUT and GET

2.3.1. SingleGet() and SinglePut() Functions:

The SingleGet() and SinglePut() functions are implemented to operate independently of RAID-specific concepts, providing a simplified and abstracted interface for data retrieval and storage. These functions take the Server ID and Block Number as inputs and perform the required operation (PUT or GET) on the specified server block.

SingleGet(): Retrieves the data from a specified block on the server identified by the given Server ID.

SinglePut(): Stores data into the specified block on the designated server using the provided Server ID and Block Number.

Both functions utilize RPC calls to communicate with the servers. They handle responses by returning either a successful result or an error code.

2.3.2. Fault Tolerant PUT

The Put method maps the virtual block number to a specific server and physical block index, then stores the data using the SinglePut method. If no errors occur, it updates both the data block and parity using 2 PUTs and 2 GETs. In case of an error, it recovers the old data from other data/parity servers, recalculates the parity, and updates it accordingly.

2.3.3. Fault Tolerant GET

The Get method maps the virtual block number to a specific server and block index, retrieving data via the SingleGet method. If an error is encountered, it reconstructs the missing data by performing an XOR operation across corresponding blocks from all servers (excluding the failed one). This process recovers the corrupted/crashed server block, provided there is only one server failure.

2.4. Block Server Repair

The repair process for a block server mirrors the recovery mechanism used for handling server downtime or corrupted blocks. The client performs an XOR operation on the corresponding data blocks from all other servers to reconstruct the lost data. This recovered data is then written back to the repaired server, restoring it to its functional state.

3. Evaluation

The implementation was tested using various file sizes and block sizes, and the number of requests was compared to the single-server approach. The results are summarized below:

System Configuration (Block Size 64):

Num Servers	Block Size	Inode Size	File Size
4	64	64	8 Blocks Size

Requests Per Server:

Server 1	Server 2	Server 3	Server 4
60	27	15	156

Out of the total 156 requests in server 4, 117 were directed to block 2, the free bitmap block. This frequent access to the free bitmap block significantly contributed to the higher load on server 4.

Number of requests in single server approach: 103

System Configuration (Block Size 128):

Num Servers	Block Size	Inode Size	File Size
4	128	128	8 Blocks Size

Requests Per Server:

Server 1	Server 2	Server 3	Server 4
37	15	25	80

Out of the total 80 requests in server 4, 63 were directed to free bitmap block.

Number of request in single server approach: 61

System Configuration (File Size 10 Blocks):

Num Servers	Block Size	Inode Size	File Size
4	64	64	10 Blocks Size

Requests Per Server:

Server 1	Server 2	Server 3	Server 4
53	24	15	123

Out of the total 123 requests in server 4, 77 were directed to free bitmap block.

Number of requests in single server approach: 130

System Configuration (File Size 14 Blocks):

Num Servers	Block Size	Inode Size	File Size
4	64	64	14 Blocks Size

Requests Per Server:

Server 1	Server 2	Server 3	Server 4
66	30	18	191

Out of the total 191 requests in server 4, 134 were directed to free bitmap block.

Num of Request in single server approach: 196

The performance results show that PUT/GET requests are distributed across different servers. The load on servers 1, 2, and 3 is significantly lower compared to the single-server approach. However, server 4 experiences a higher number of requests in the RAID 5 setup. Upon analysis, it was found **that the free bitmap block (virtual block number 2)** was mapped to server 4, block number 0. This block was frequently accessed by the client, leading to a higher number of requests on server 4. Despite this, as the file system grows, the average number of requests should balance out across all servers.

4. Test and Reproducibility

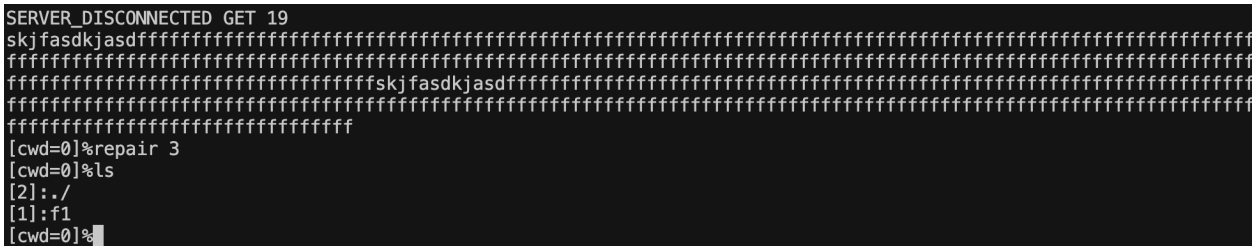
The implementation was evaluated under various scenarios, including different numbers of servers, server crashes, and block corruption cases. The system performed as expected, maintaining fault tolerance and ensuring file consistency. Below are some key testing results:

5 servers running:
python3 blockserver.py -bs 64 -nb 256 -port 8000
python3 blockserver.py -bs 64 -nb 256 -port 8001
python3 blockserver.py -bs 64 -nb 256 -port 8002
python3 blockserver.py -bs 64 -nb 256 -port 8003
python3 blockserver.py -bs 64 -nb 256 -port 8004

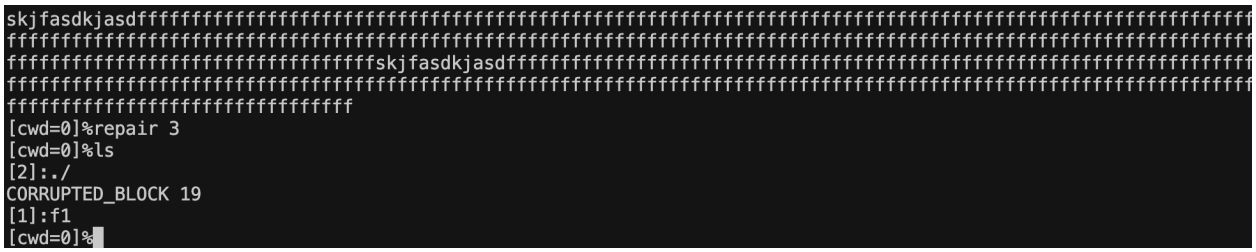
Client Running:
python3 fsmain.py -ns 5 -startport 8000 -nb 1024 -bs 64 -is 64 -ni 32 -cid 0

Disconnect Server 3:

Repair 3



Corrupt Block 4



Note: Server Disconnected message is printed only for a virtual data block PUT/GET. This is not printed when updating a parity block. Also corrupted block (-cblk) should not be configured 0.

5. Conclusions

In conclusion, this project successfully demonstrates the implementation of distributed storage with fault tolerance using a client and block server architecture, leveraging RAID 5 principles. The system effectively handles server crashes and block corruption while ensuring consistent file operations and data integrity. By distributing data blocks and parity across multiple servers, the design achieves load balancing and enhanced aggregate capacity, reducing the dependency on a single server. Through rigorous testing under various scenarios, the system proved resilient and capable of maintaining performance and fault tolerance. This project highlights the potential of distributed storage solutions in improving reliability and scalability, offering valuable insights for real-world applications.