Homework 5 final report

Zekun Yang,Haoyong Liu

1. Introduction and Problem Statement

1.1. Objective

In this project, we aim to build a Multi-client, Multi-server-based file system.. This system needs to run on multiple fault-tolerant servers to provide basic file services to multiple clients.

1.2. Project Overview

We have designed a system to offer create file,create directory,store content to file and create “soft and hard” link function. This system has the ability to protect the atomicity of file content,balance loads and create cache to reduce system load. At the same time, this system has the ability to automatically correct single block errors and continue to provide services when a single server is down.

2. Design and Implementation

2.1. Overall Design

This system is mainly divided into two parts: client and server.
The client provides functions such as creating files and directories, creating soft and hard connections, and adding, deleting, modifying, and checking files to the server. It also performs block acquisition and storage requests on the server side, and performs processing when block damage occurs on the server side or a single server fails. This includes customer reminders and functions such as restoring server content after a failed server is repaired.In the process of communicating with the server, layer a provides the conversion function between virtual address and physical address.
The server side provides the function of storing and reading blocks, and sends a warning to the client when a certain physical block is damaged.
Figure 1 briefly shows the system structure.

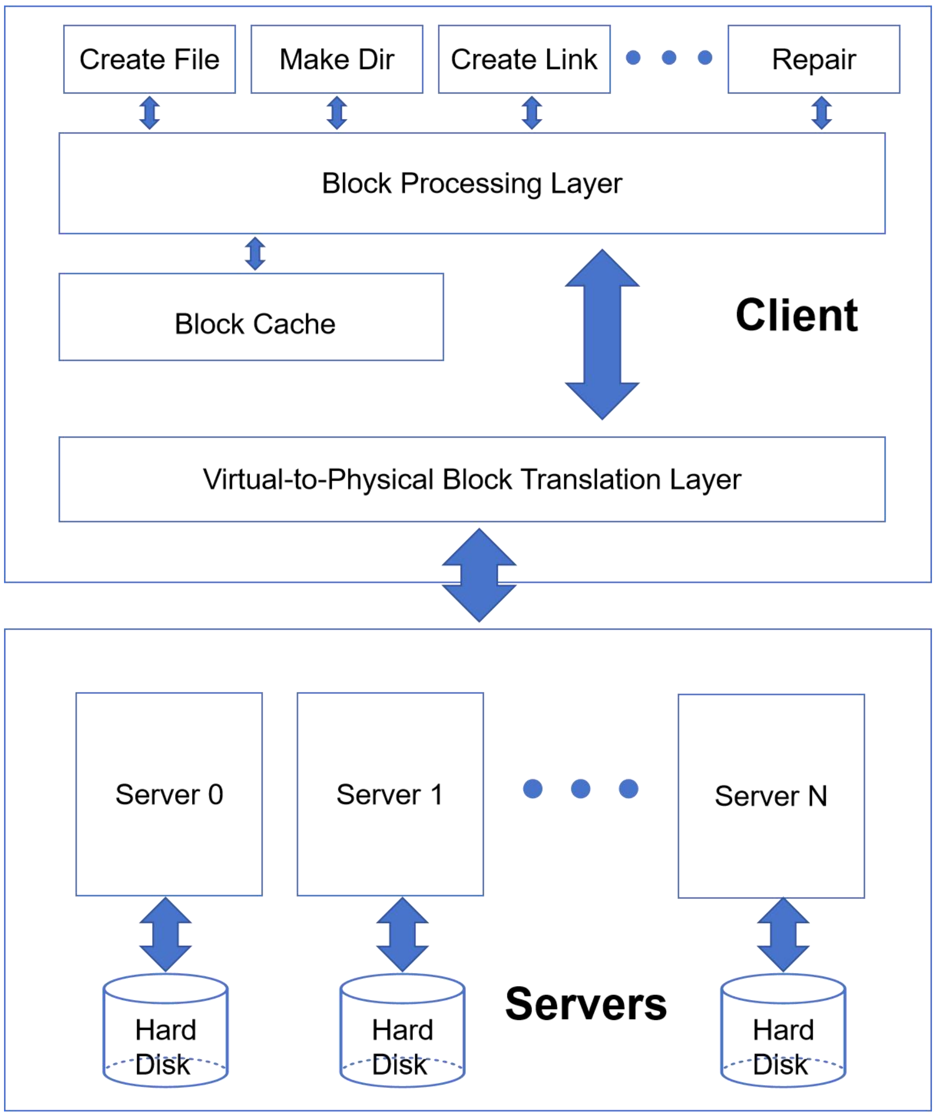


Figure 1 System Overall Design

2.2. Block.py

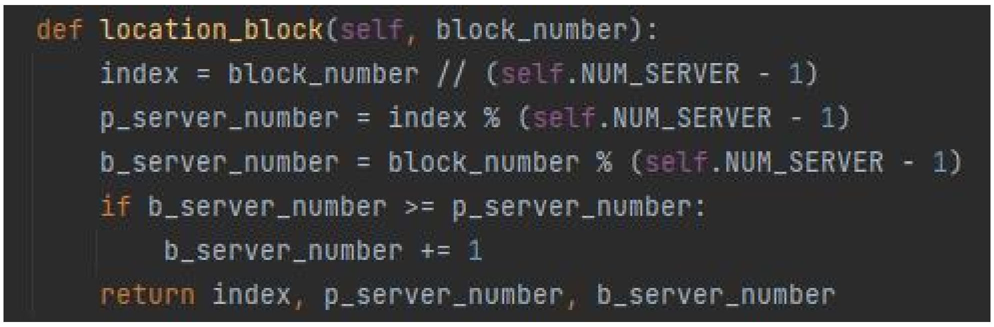
Block.py mainly includes the implementation of the client's Block Processing Layer, Block cache, and Virtual-to-Physical Block Translation Layer functions.

2.2.1. Implement the logic to handle N block_servers

When starting a new client, define the number of servers provided by the entire file system. Create an RPC connection for each server in block.py and store the connection in an array. At the same time, create a corresponding server connection array to storage these servers connection status. Because to implement raid-5, at least three servers are required. When the number of servers is less than three, an error will be reported.

2.2.2. Implement the logic to distribute blocks across multiple servers for Put() and Get() which follow the RAID-5 approach.

In order to implement raid-5, we created a Virtual-to-Physical Block Translation algorithm to calculate the correspondence between each virtual block and physical blocks and parity blocks. The details of the algorithm are as follows:



```
def location_block(self, block_number):
    index = block_number // (self.NUM_SERVER - 1)
    p_server_number = index % (self.NUM_SERVER - 1)
    b_server_number = block_number % (self.NUM_SERVER - 1)
    if b_server_number >= p_server_number:
        b_server_number += 1
    return index, p_server_number, b_server_number
```

index is the physical block number corresponding to the virtual block and the physical block number of the parity block in the server, p_server_number is the server number where the parity block is located, and b_server_number is the server number where the physical block is located.This algorithm is the basis for implementing raid-5.

In order to support multi-server and raid-5, we renamed the previous get() and put() functions to SingleGet() and SinglePut(), and created new get() and put() functions. The function of the new get() in the system is to parse the parameters passed to the SingleGet() function, and process the results returned by the SingleGet() function. When CORRUPTED_BLOCK and SERVER_DISCONNECTED errors occur, obtain other blocks and parity block from other servers to correct errors.The function of the new Get() function is changed to parse the parameters passed to the SinglePut() function including new data blocks and new parity blocks, and when a SERVER_DISCONNECTED error occurs, determine the type of block stored in the server where the error occurred, different processing is performed depending on whether it is a data block or a parity block.At the same time, mark the server out of service in the corresponding server connection array.

SinglePut() function also sent the MD5 of data to the server.

The Get() and Put() functions will interpret the current server connection status. When a server is offline, it will enter a special processing state. In this state, although the service can be maintained, each request to the server will The quantity is increased to 2-(N-1) times.

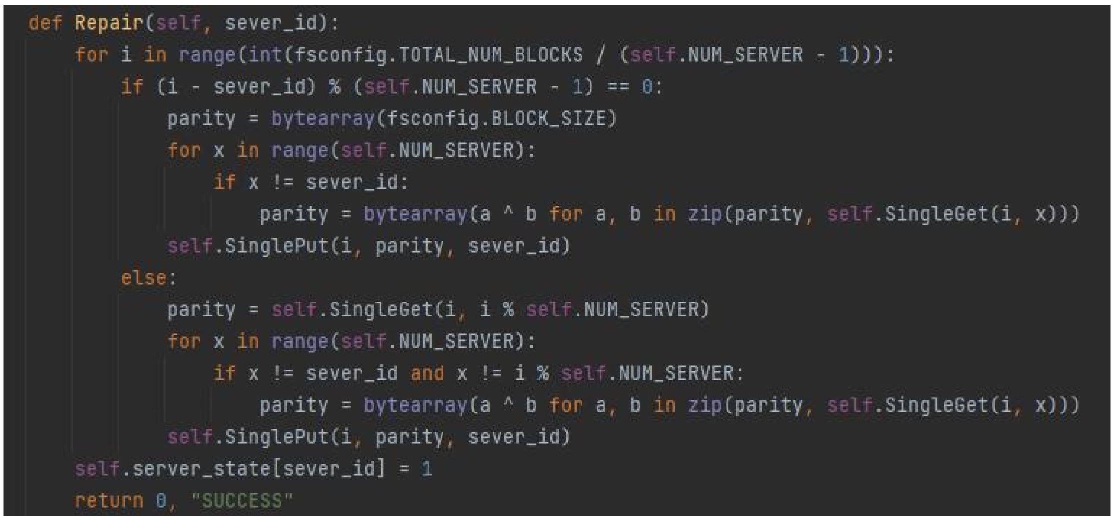
2.2.3. Implement the logic to support RSM() by using a *single server* as a special case

We still select the last last virtual block number as the block to store RSM information, and when initializing the client, use the Virtual-to-Physical Block Translation algorithm to parse the server number where the block is located.

2.2.4. Implement the repair procedure

Due to the characteristics of RAID-5, when one server fails, the remaining servers can still provide services. When the failed server comes back online,

the content that should be stored in the server can be restored through the data of other servers.
When performing recovery, we will traverse each block of the remaining servers, use these blocks and the number of the offline server to calculate whether each block in the offline server stores a data block or a parity block, and restore these blocks to the offline server.The specific algorithm is as follows:



```python
def Repair(self, sever_id):
    for i in range(int(fsconfig.TOTAL_NUM_BLOCKS / (self.NUM_SERVER - 1))):
        if (i - sever_id) % (self.NUM_SERVER - 1) == 0:
            parity = bytearray(fsconfig.BLOCK_SIZE)
            for x in range(self.NUM_SERVER):
                if x != sever_id:
                    parity = bytearray(a ^ b for a, b in zip(parity, self.SingleGet(i, x)))
            self.SinglePut(i, parity, sever_id)
        else:
            parity = self.SingleGet(i, i % self.NUM_SERVER)
            for x in range(self.NUM_SERVER):
                if x != sever_id and x != i % self.NUM_SERVER:
                    parity = bytearray(a ^ b for a, b in zip(parity, self.SingleGet(i, x)))
            self.SinglePut(i, parity, sever_id)
    self.server_state[sever_id] = 1
    return 0, "SUCCESS"
```

2.3. blockserver.py

In blockserver.py, an array is mainly added to store the MD5 value corresponding to each physical block and a parameter cblk to mark the damaged block.
The put() function adds the function of storing md5 values based on the previous one. The get() function returns a block damage error message when block_number is equal to the parameter cblk.

3. Evaluation

3.1. One client and N Servers.

Start testing with the most basic situation of one client and four servers. Create four servers with ports 8000-8003. In this case, the basic functions of the system such as creating files, creating directories, and creating soft and hard links are carry out testing.
After completing the functional test under basic conditions, restart all servers and clients, and shut down server 0. Under this condition, the RSM service is on server 3. Under this condition, the parity block corresponding to the penultimate block that stores the last access is stored on server 0. Therefore, every time the last access information is updated, server 0 will be accessed at the same time.

At this time, the operation of the system was tested when one server was offline. After that, we restore the connection to server 0 and run the recovery function to observe the access status of the four servers. After the recovery is completed, test the recovery results.
Set the block one to cblk on server 3. Because this block is the block where the root directory is located, all operations on the root directory will access this block, so the damage to a single block can be tested.
Modify the number of enabled servers from 4 to 8 and perform the above test again.

3.2. N client and N Servers.

Testing began with the most basic situation of two clients(0 and 1) and four servers. Create four servers with ports 8000-8003.
Test basic file functions such as create, append, rm, and cat on two clients respectively, and confirm that the basic system works normally.
After completing the functional test under basic conditions, restart all servers and clients, and shut down server 0. Under this condition, the RSM service is on server 3. Under this condition, the parity block corresponding to the penultimate block that stores the last access is stored on server 0. Therefore, every time the last access information is updated, server 0 will be accessed at the same time.
At this time, the operation of the system was tested when one server was offline. After that, we restore the connection to server 0 and run the recovery function to observe the access status of the four servers. After the recovery is completed, test the recovery results.
At this time, the operation of the system was tested when one server was offline. After that, we restore the connection to server 0 and run the recovery function to observe the access status of the four servers. After the recovery is completed, test the recovery results.
Set the block two to cblk on server 0. Because this block is the block where the root directory is located, all operations on the root directory will access this block, so the damage to a single block can be tested.
Modify the number of enabled servers from 4 to 8 ,the number and perform the above test again.

By the way: During multi-client testing, we discovered a serious bug. Due to the characteristics of Python, we used a dictionary to store the contents of the block when implementing the cache. The storage method at this time was shallow copy, that is, variables with the same value stored in the same physical Address, when the InitRootInode() function is called, the block modification in the root directory will be synchronized to the cache. At this time, the calculated parity will be wrong. If a server is down at this time, use parity to process the data. An error occurs during recovery. Therefore, we back up the data stored in the cache dictionary using deep copy alone to avoid accidental modifications.

3.3. Evaluated the load distribution

First, create 50 files in the state of HW4, and write ten characters a-i in each file, and count the number of accesses to a single server as a baseline.
After creating four servers, create 50 files and write ten characters a-i in each file, and count the number of accesses to every server.Divide the total number of accesses by the number of servers to calculate the average number of accesses to a single server.Comparing with the baseline, it is found that under the baseline condition, the number of requests for a single server is approximately 2.6 times that of the four-server condition.

Second, creating four servers, create 50 files and write ten characters a-i in each file, and count the number of accesses to every server.Divide the total number of accesses by the number of servers to calculate the average number of accesses to a single server as the baseline. Restart all servers and clients, and shut down one of the servers (non-RSM server).eate 50 files and write ten characters a-i in each file, and count the number of accesses to every server.Divide the total number of accesses by the number of servers to calculate the average number of accesses to non-down server.

Thirdly,create 50 files in the state of HW4,and write 257 characters (two blocks) in each file, and count the number of accesses to a single server as a baseline. After creating four servers, create 50 files and write 257 characters (two blocks) in each file, and count the number of accesses to every server.Divide the total number of accesses by the number of servers to calculate the average number of accesses to a single server.Comparing with the baseline.

4. Reproducibility

Test1:
Running the System:
Servers:

```
python .\blockserver.py -nb 256 -bs 256 -port 8000
python .\blockserver.py -nb 256 -bs 256 -port 8001
python .\blockserver.py -nb 256 -bs 256 -port 8002
python .\blockserver.py -nb 256 -bs 256 -port 8003
```

Client:

```
python fsmain.py -cid 0 -ns 4 -nb 768 -bs 256 -startport 8000
```

Shut down the server 0 (port 8001)
At Client:

```
create f1
```

ls
append f1 this_is_f1
ls
cat f1
repair 0
append f1 _more_data
cat f1
Shut down all the servers and client

Test2:
Running the System:
Servers:
python .\blockserver.py -nb 256 -bs 256 -port 8000
python .\blockserver.py -nb 256 -bs 256 -port 8001
python .\blockserver.py -nb 256 -bs 256 -port 8002
python .\blockserver.py -nb 256 -bs 256 -port 8003
Client:
python fsmain.py -cid 0 -ns 4 -nb 768 -bs 256 -startport 8000 -logcache 1
python fsmain.py -cid 1 -ns 4 -nb 768 -bs 256 -startport 8000

At Client 0:
create f1
ls
append f1 this_is_f1
ls
cat f1

At Client 1:
create f2
ls
append f2 this_is_f2
cat f1
cat f2

At Client 0:
ls
cat f1
cat f2
Shut down all the servers and client

Test3:
Running the System:
Servers:
python .\blockserver.py -nb 256 -bs 256 -port 8000

```
python .\blockserver.py -nb 256 -bs 256 -port 8001 -cblk 2
python .\blockserver.py -nb 256 -bs 256 -port 8002
python .\blockserver.py -nb 256 -bs 256 -port 8003
```
Client:
```
python fsmain.py -cid 0 -ns 4 -nb 768 -bs 256 -startport 8000
```
At Client:
```
create f1
ls
append f1 this_is_f1
ls
cat f1
```
Shut down all the servers and client

Test4:
Running the System:
Servers:
```
python .\blockserver.py -nb 256 -bs 256 -port 8000
python .\blockserver.py -nb 256 -bs 256 -port 8001
python .\blockserver.py -nb 256 -bs 256 -port 8002
python .\blockserver.py -nb 256 -bs 256 -port 8003
python .\blockserver.py -nb 256 -bs 256 -port 8004
python .\blockserver.py -nb 256 -bs 256 -port 8005
```
Client:
```
python fsmain.py -cid 0 -ns 6 -nb 1024 -bs 256 -startport 8000
python fsmain.py -cid 1 -ns 6 -nb 1024 -bs 256 -startport 8000
```

At Client 0:
```
create f1
ls
append f1 this_is_f1
ls
cat f1
```

At Client 1:
```
create f2
ls
append f2 this_is_f2
cat f1
cat f2
```

At Client 0:
```
ls
cat f1
cat f2
```

Shut down all the servers and client

5. Conclusion

5.1 Summary of Achievements

Through this project, we have built a client/server-based file system where the client accesses data blocks stored on the server. In this final project, we will extend the file system to multiple servers supporting redundant block storage. This project will provide an understanding of practical issues in system design including: client/service, networking, fault tolerance, etc.

5.2 Lessons Learned

Through this project, We have gained a deeper understanding of how distributed storage systems work. Especially under the RAID-5 architecture, how to achieve data redundancy and fault tolerance through parity and data distribution. This project also made me realize the importance of how to handle read and write requests, failure detection and recovery procedures when designing such a system. During the programming practice, We learned how to implement client/server communication in Python and how to efficiently manage concurrent operations and failure handling.