# Project Report

Nan Mu

## I. Introduction and problem statement

This project designs a client-server file system which supports data distribution to multiple servers, RAID5 storage mode is implemented so that client side can communicate with multiple servers by RPC, and servers can execute requests in parallel, that will improve the system throughput. Meanwhile, this file system has the ability for tolerating fault by using the parity, when one server failed or block corrupted, it can continue to work and has ability to repair to normal. This report will introduce design method and implementation procedure in second section, and different system testing results are shown on third section, fourth and fifth section is about the reproducibility and conclusion.

In general, single disk storage system has problems with the performance and reliability. That means, all requests from client side need to be processed by this single disk one by one, if the server-side processing rate is stable, then the waiting queue will be long when requests blow up, so that the latency of requests will increase. The other thing is single disk storage has no ability to undertake the server problem such like section corrupt or server disconnection. That may cause the untolerated error for the client side. That's why we need to design and implement a multi-server storage system with redundancy. That will improve the performance of latency and throughput and can increase reliability.

## II. Design and implementation

In this project, a RAID-5 file system is designed and program code is extended to support the load distribution and failure tolerance function. This part will introduce modification for server, shell part and explain the client implementation in detail about how to distribute requests and deal with errors.

### 1, Client side

In client side, the higher layer such like Inode layer or Filename layer will be reused without changing, the main modification is about Diskblock layer. A block_server list is built at DisksBlock layer to record address for each server, which will communicate with server side, also, new function will be used to converse virtual block to physical block and its relevant server, Put(), Get() and other functions has been created or modified to implement RAID5 system and deal with the server-side error.

#### 1) Virtual Block Mapping

In client, file system can initialize with different numbers of server from 4 to 8, so that we need a function to map virtual block to physical block. A new function VirtualBlockMap() is added to DisksBlock layer which will take the virtual block number and return the server number, physical number and relevant parity server number.

The mapping calculation method is as follows:

Input: virtual block number

```
    physical_server = virtual_block_number % (TOTAL_SERVER - 1)
    physical_block = virtual_block_number // (TOTAL_SERVER - 1)
    parity_server = (TOTAL_SERVER - 1) - (physical_block % TOTAL_SERVER)
    if physical_server >= parity_server:
        physical_server += 1
Output: physical_server_number, physical_block_number, parity_server_number
```

Below is the mapping table example with total 7 servers, other mapping table for different server number has put on the appendix part.

| Block \ Server | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | p |
| 1 | 6 | 7 | 8 | 9 | 10 | p | 11 |
| 2 | 12 | 13 | 14 | 15 | p | 16 | 17 |
| 3 | 18 | 19 | 20 | p | 21 | 22 | 23 |
| 4 | 24 | 25 | p | 26 | 27 | 28 | 29 |
| 5 | 30 | p | 31 | 32 | 33 | 34 | 35 |
| 6 | p | 36 | 37 | 38 | 39 | 40 | 41 |

## 2) RAID5 Layer

Original Put and Get method has been renamed to SinglePut() and SingleGet() which will take server number and physical block number as input and communicate with each server disk, it will send request and receive response. it implements the Fail-Fast scheme, once server side has corrupted error or disconnect problem, it will edit error tag and deliver it to its upper layer Put () or Get (). In this project, Get () and Put () layer has been modified to implement RAID-5 and resolve the error message.

**RAID5 Put (),** when Put () is called to update block data, firstly, it will call VirtualBlockMap() function to map virtual block number with physical block info and its relevant parity server. Then it will take two Get () for the parity update, the one will call Get () to acquire old block data, the other will call GetParity() to acquire old parity data, then these two parts will do XOR with new block data. Next it will call SinglePut() to update parity, and call the second SinglePut() to update block data. each SinglePut() will return the tag, if receive "ERROR_DISCONNECT" tag (number -2), then it will print message "SERVER_DISCONNECTED , Put, Server_Number" to the client screen.

**RAID5 Get (),** now Get () will take virtual block number as input, and call VirtualBlockMap() to map virtual block to physical block. Then it call SingleGet() with server number and physical block number, and receive tag from SingleGet(). It will use received tag to response problems, if no error, Get() will return its received data to its upper layer; if it receives "ERROR_CORRUPT" (-1), it will print message "CORRUPTED_BLOCK, virtual_block_number" to the terminal to tell which corresponding virtual block is corrupted; if it receives "ERROR_DISCONNECT" (-2), it will print message "SERVER_DISCONNECTED, Get, server_number". For both two error, Get () will call function GetRecover() with server and physical block number. Then GetRecover will return calculated data by using other server, and Get will return it to its upper layer. So single error will be masked at Get function. Upper layer will not know the error.

**RAID5 GetParity()**, same as Get function, GetParity() will get parity data and resolve error when parity block is corrupted or its server is disconnected. The different thing from Get function is that, parity block is not a data block for the file system, so it doesn't have mapped virtual_block_number. When a Put () is called to store virtual block data and Put () need to get parity value to update new parity, if this parity is corrupted, GetParity will print message to terminal, but the block number will be the virtual_block_number who calling parity to do updating.

**RAID5 GetRecover()**, when error detected and received in Get or GetParity function, the GetRecover() will be called to mask error. It will take the failed server number and physical block number as input, then get other servers' same block_number content to do XOR calculations and return this calculated result to caller function.

**RAID5 RepairServer()**, this function will be used to repair failure server. When command line call "repair server_id", RepairServer function will take this server number as input, and calculate the server total block numbers. Then it calls GetRecover function to recover blockdata one by one, and store recovery data to server disk. In this design, the server id starts from 0, such as server 0, server 1, server 2.

**3) Isolation RSM Block**, in this project implementation, the RSM block (virtual block 0) will be isolated, and not do parity update and block recover, that will be implemented on Put () function and GetRecover() function. For example, when Put () update virtual block 0 (i.e. Put(0, block_data)) , it will not update parity, if GetRecover(server, block 0) is called to recover first block of one server (such like server 2 block 0), then the RSM block (server 0 block 0) will not be used do XOR calculation. As a result, RSM can still use to lock system, but it does not attend to update parity and calculate for block recover.

The reason is that when we execute command such like 'create', 'append', we use Acquire function to call the RSM function, RSM function operate atomically read and change RSM block from '0' to '1' in server side directly, but it does not update corresponding parity. Then when we finish command execution, we will use Release function to call Put () function, the Put will change RSM block (block 0) from '0' to '1' and update corresponding parity. RSM () does not update the parity but Release update parity, that may cause the problem when use GetRecover to calculate other block data (suck like bitmap block 2 which locate at server 2 block 0)

**4) Load Distribution**. In order to analyze load distribution of multiple server system, a list is added in DiskBlock class which called server_load, each SingleGet() or SinglePut() call will increase the counts of corresponding server. Command 'serverld' will print current load distribution record to client screen.

## 2, Server Side

**Implement Checksum.** In the server side, for Put () function, it will calculate corresponding checksum by using md5 algorithm and store new checksum for its block. For Get () function, firstly, it will read block data and calculate checksum, then compare calculated checksum with stored checksum, if comparation failed, that means the block data is corrupted, server will return corrupted error (tag -1) to client side.

Server side also implement a '-cblk' command line to imitate a corrupted block, when initial a server with '-cblk block_number', if Get () receive request for this corrupted block, it will return corrupted error to client side, then in client, it will print corresponding virtual block number on screen.

### 3, Client Shell Side

Expand command line '-ns' and 'startport'. 'ns' will configure how many servers has been initialed, 'startport' will clarify the first server port, then client program will calculate each server port and communicate with them. Build 'repair server_id' command, this command will call the RepairServer() function to repair failed server, the  server_id start from 0, for example, port 8000 match server 0, 8002 match server 2. Build 'serverld' command, this command will print load distribution on terminal.

## III. Evaluation

For the evaluation part, firstly, we need to test if file system execution normally with multiple servers, different file system configuration such like different number of server (from 4 to 8), different block size, number and different inode size need to be considered. Secondly, it need test if file system can run correctly under the server or block failure condition, and if it can repair correctly. Thirdly, the multiple server performance will be compared to single server system, the load distribution will be evaluated.

### 1, Multiple Server Implementation Test

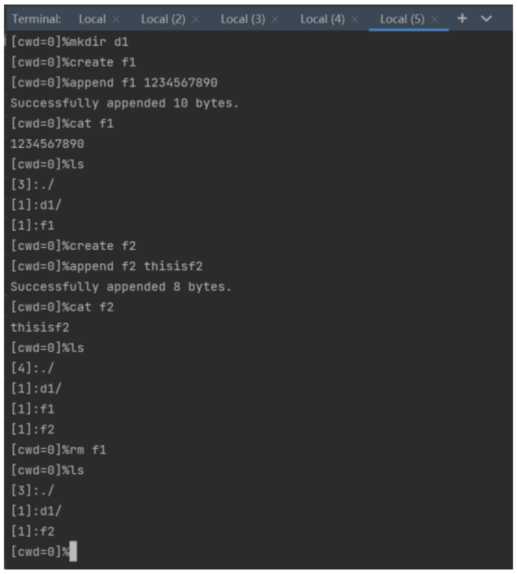
As shown in figure 1 and 2, file system can run normally for different file system configurations

**First system configuration**: server side: -nb: 256, -bs: 128

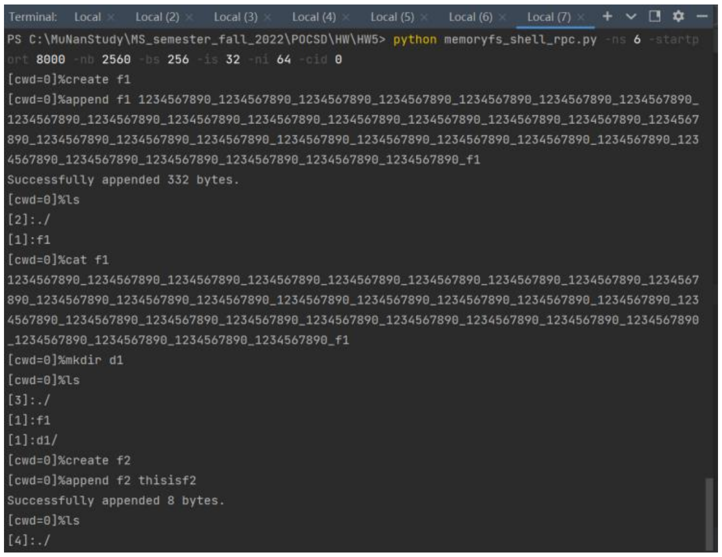
Client side: -ns: 4, -nb: 768, -bs: 128, -is: 16, -ni: 16

**Second system configuration**: server side: -nb: 512, -bs: 256

Client side: -ns: 6, -nb: 2560, -bs: 256, -is: 32, -ni: 64



```
[cwd=0]%mkdir d1
[cwd=0]%create f1
[cwd=0]%append f1 1234567890
Successfully appended 10 bytes.
[cwd=0]%cat f1
1234567890
[cwd=0]%ls
[3]:./
[1]:d1/
[1]:f1
[cwd=0]%create f2
[cwd=0]%append f2 thisisf2
Successfully appended 8 bytes.
[cwd=0]%cat f2
thisisf2
[cwd=0]%ls
[4]:./
[1]:d1/
[1]:f1
[1]:f2
[cwd=0]%rm f1
[cwd=0]%ls
[3]:./
[1]:d1/
[1]:f2
[cwd=0]%
```



```
PS C:\MuNanStudy\MS_semester_fall_2022\POCSD\HW\HW5> python memoryfs_shell_rpc.py -ns 6 -startp
ort 8000 -nb 2560 -bs 256 -is 32 -ni 64 -cid 0
[cwd=0]%create f1
[cwd=0]%append f1 1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_
1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567
890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_123
4567890_1234567890_1234567890_1234567890_1234567890_1234567890_f1
Successfully appended 332 bytes.
[cwd=0]%ls
[2]:./
[1]:f1
[cwd=0]%cat f1
1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567
890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_123
4567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890_1234567890
_1234567890_1234567890_1234567890_1234567890_f1
[cwd=0]%mkdir d1
[cwd=0]%ls
[3]:./
[1]:f1
[1]:d1/
[cwd=0]%create f2
[cwd=0]%append f2 thisisf2
Successfully appended 8 bytes.
[cwd=0]%ls
[4]:./
```

Figure 1. first system configuration

Figure 2. Second system configuration

### 2, Single Server Disconnection and Repair Test

When we test single server disconnection, first we execute some operation in normal, then we disconnect one server, and test if system can still work correctly with the 'server_disconnected' message. even though single server failed, system still can ‘ls’, ‘cat’ file and ‘create’, ‘append’ new file content. Then we will reconnect server and repair it, to check if system work correctly. Below are two examples with different system configuration, figure 3 is about first case, figure 4 and 5 is about second case. As we can see, when access block which locate at failure server, client screen will print server disconnect message, when we reconnect that server, and repair it, then the system still works normally. We can disconnect and repair server multiple times.

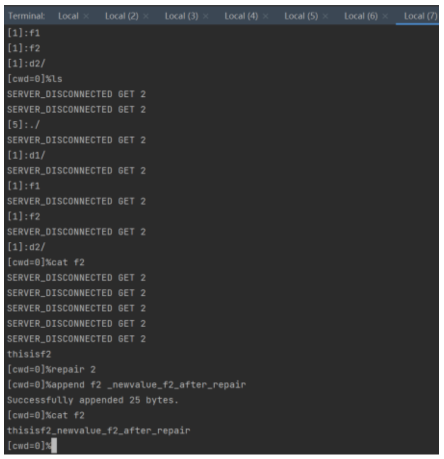
**1) First case**: Server side: -nb: 512, -bs: 256.  Client side: -ns: 6, -nb: 2560, -bs: 256, -is: 32, -ni: 64

Disconnected server: 2, (the server 2 block 2 will map virtual block 12 which is first block of inode table in this file system configuration)

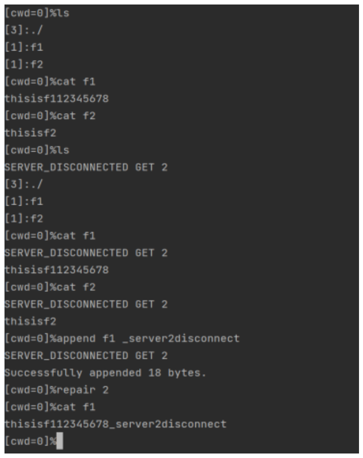
**2) Second case**: Server side: -nb: 256, -bs: 128.  Client side: -ns: 4, -nb: 768, -bs: 128, -is: 16, -ni: 16

Disconnected server: 2, (the server 2 block 3 will map virtual block 10 which is data block of root inode 0 in this file system configuration)

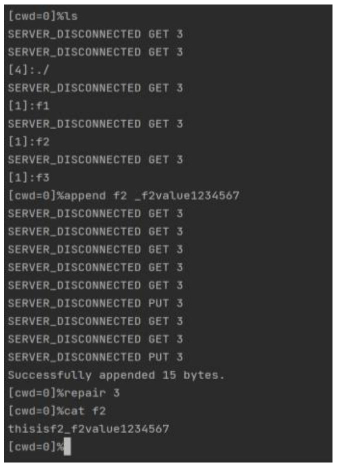
Disconnected server: 3, (the server 3 block 2 will map virtual block 8 which is first block of inode table in this file system configuration)



```
[1]:f1
[1]:f2
[1]:d2/
[cwd=0]%ls
SERVER_DISCONNECTED GET 2
SERVER_DISCONNECTED GET 2
[5]:./
SERVER_DISCONNECTED GET 2
[1]:d1/
SERVER_DISCONNECTED GET 2
[1]:f1
SERVER_DISCONNECTED GET 2
[1]:f2
SERVER_DISCONNECTED GET 2
[1]:d2/
[cwd=0]%cat f2
SERVER_DISCONNECTED GET 2
SERVER_DISCONNECTED GET 2
SERVER_DISCONNECTED GET 2
SERVER_DISCONNECTED GET 2
SERVER_DISCONNECTED GET 2
thisisf2
[cwd=0]%repair 2
[cwd=0]%append f2 _newvalue_f2_after_repair
Successfully appended 25 bytes.
[cwd=0]%cat f2
thisisf2_newvalue_f2_after_repair
[cwd=0]%
```



```
[cwd=0]%ls
[3]:./
[1]:f1
[1]:f2
[cwd=0]%cat f1
thisisf112345678
[cwd=0]%cat f2
thisisf2
[cwd=0]%ls
SERVER_DISCONNECTED GET 2
[3]:./
[1]:f1
[1]:f2
[cwd=0]%cat f1
SERVER_DISCONNECTED GET 2
thisisf112345678
[cwd=0]%cat f2
SERVER_DISCONNECTED GET 2
thisisf2
[cwd=0]%append f1 _server2disconnect
SERVER_DISCONNECTED GET 2
Successfully appended 18 bytes.
[cwd=0]%repair 2
[cwd=0]%cat f1
thisisf112345678_server2disconnect
[cwd=0]%
```



```
[cwd=0]%ls
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED GET 3
[4]:./
SERVER_DISCONNECTED GET 3
[1]:f1
SERVER_DISCONNECTED GET 3
[1]:f2
SERVER_DISCONNECTED GET 3
[1]:f3
[cwd=0]%append f2 _f2value1234567
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED PUT 3
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED GET 3
SERVER_DISCONNECTED PUT 3
Successfully appended 15 bytes.
[cwd=0]%repair 3
[cwd=0]%cat f2
thisisf2_f2value1234567
[cwd=0]%
```

Figure 3

Figure 4

Figure 5

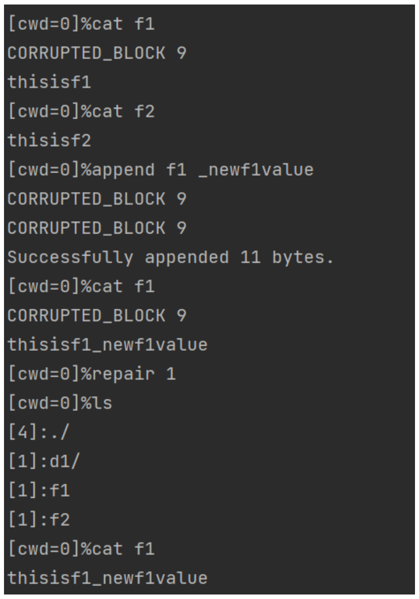
## 3, Block Corruption Test

For testing data block corruption, we need to set ‘-cblk’ in one server, and test in client side if Get () corrupted block can print message and calculate result by other server block. the system will not work for corrupting same physical block of different server at same time. In this part, two scenarios need to be tested. The one is the corrupted block is the data block, then in client terminal, it will print the corresponding virtual block number. The other is the corrupted block is the parity block, then it will also print the virtual block number that is who need to use this parity, in this condition, terminal may print different virtual block number, the example below shows this scenario.

**1) First case**: Server side: -nb: 256, -bs: 256. Client side: -ns: 5, -nb: 1024, -bs: 256, -is: 16, -ni: 16

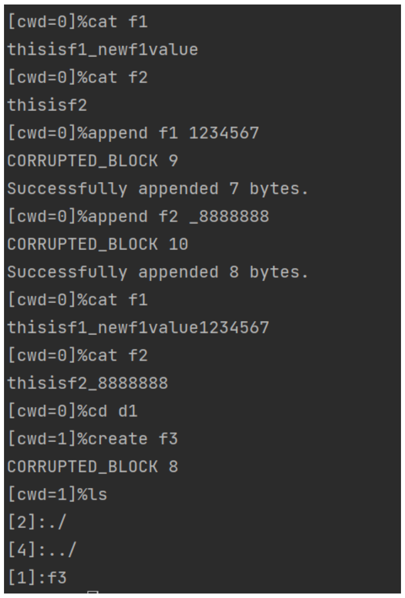
**First time Corrupted block**: server 1, block 2. That maps virtual block 9 which is data block of file ‘f1’ in this file system configuration. As shown in figure 6, each Get for f1 will print the corrupt block message.

**Second time Corrupted block**: server 2, block 2. That is a parity block which map virtual block 8, 9, 10 and 11. The block 8 is directory ‘d1’, block 9 is file ‘f1’, block 10 is file ‘f2’ in this system configuration. As shown in figure 7, when corrupted block is parity block. each time we modify f1, f2 or d1, that will update parity, then terminal will print corrupt block message with corresponding virtual block number.



```
[cwd=0]%cat f1
CORRUPTED_BLOCK 9
thisisf1
[cwd=0]%cat f2
thisisf2
[cwd=0]%append f1 _newf1value
CORRUPTED_BLOCK 9
CORRUPTED_BLOCK 9
Successfully appended 11 bytes.
[cwd=0]%cat f1
CORRUPTED_BLOCK 9
thisisf1_newf1value
[cwd=0]%repair 1
[cwd=0]%ls
[4]:./
[1]:d1/
[1]:f1
[1]:f2
[cwd=0]%cat f1
thisisf1_newf1value
```

Figure 6



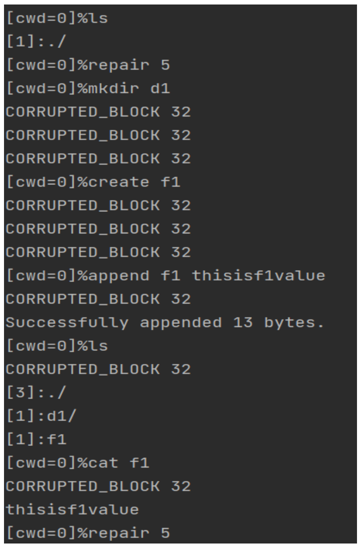
```
[cwd=0]%cat f1
thisisf1_newf1value
[cwd=0]%cat f2
thisisf2
[cwd=0]%append f1 1234567
CORRUPTED_BLOCK 9
Successfully appended 7 bytes.
[cwd=0]%append f2 _8888888
CORRUPTED_BLOCK 10
Successfully appended 8 bytes.
[cwd=0]%cat f1
thisisf1_newf1value1234567
[cwd=0]%cat f2
thisisf2_8888888
[cwd=0]%cd d1
[cwd=1]%create f3
CORRUPTED_BLOCK 8
[cwd=1]%ls
[2]:./
[4]:../
[1]:f3
```

Figure 7

**2) Second case**: Server side: -nb: 256, -bs: 128. Client side: -ns: 8, -nb: 1792, -bs: 128, -is: 32, -ni: 64

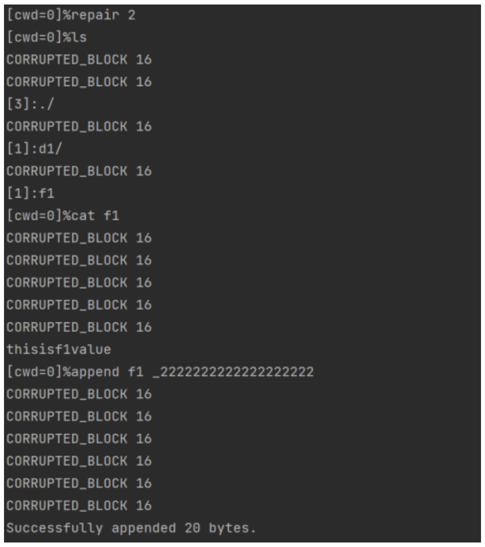
**First time Corrupted block**: server 5, block 4. That will map virtual block 32 which is data block of root inode 0 in this configuration.

**Second time Corrupted block**: server 2, block 2. That will map virtual block 16 which is first block of inode table in this file system configuration. As shown in figure 8 and 9, corrupted virtual block number printed at screen.



```
[cwd=0]%ls
[1]:./
[cwd=0]%repair 5
[cwd=0]%mkdir d1
CORRUPTED_BLOCK 32
CORRUPTED_BLOCK 32
CORRUPTED_BLOCK 32
[cwd=0]%create f1
CORRUPTED_BLOCK 32
CORRUPTED_BLOCK 32
CORRUPTED_BLOCK 32
[cwd=0]%append f1 thisisf1value
CORRUPTED_BLOCK 32
Successfully appended 13 bytes.
[cwd=0]%ls
CORRUPTED_BLOCK 32
[3]:./
[1]:d1/
[1]:f1
[cwd=0]%cat f1
CORRUPTED_BLOCK 32
thisisf1value
[cwd=0]%repair 5
```

Figure 8



```
[cwd=0]%repair 2
[cwd=0]%ls
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
[3]:./
CORRUPTED_BLOCK 16
[1]:d1/
CORRUPTED_BLOCK 16
[1]:f1
[cwd=0]%cat f1
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
thisisf1value
[cwd=0]%append f1 _2222222222222222222
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
CORRUPTED_BLOCK 16
Successfully appended 20 bytes.
```

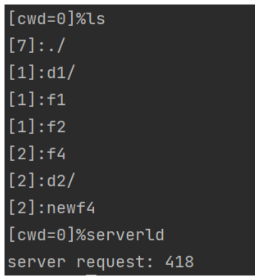
Figure 9

## 4, Load Distribution Evaluation

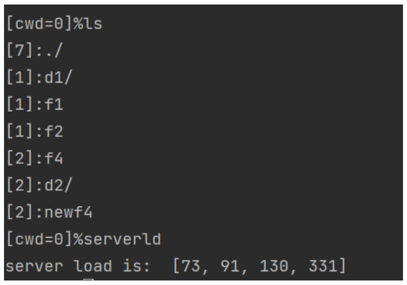
For distribution evaluation. We need to execute same serials operation in single server (HW4 code) system and multiple server system, then comparing the number of requests that each server processed.

**1) First case**: single server vs four servers. Below two figures show the request counts

Server side: -nb: 256, -bs: 128. Client side: -ns: 4, -nb: 768, -bs: 128, -is: 16, -ni: 16.



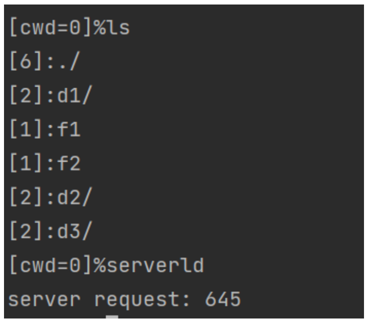
```
[cwd=0]%ls
[7]:./
[1]:d1/
[1]:f1
[1]:f2
[2]:f4
[2]:d2/
[2]:newf4
[cwd=0]%serverld
server request: 418
```



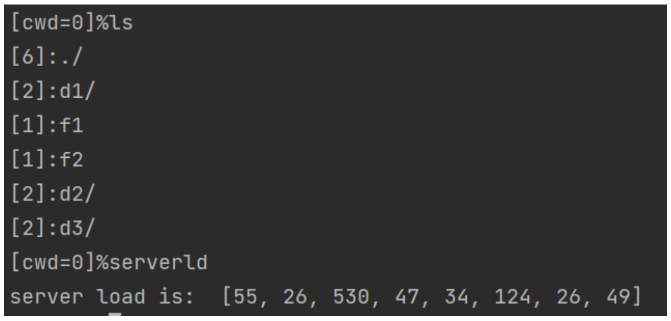
```
[cwd=0]%ls
[7]:./
[1]:d1/
[1]:f1
[1]:f2
[2]:f4
[2]:d2/
[2]:newf4
[cwd=0]%serverld
server load is:  [73, 91, 130, 331]
```

**2) Second case**: single server vs eight servers. Below two figures show the request counts

Server side: -nb: 512, -bs: 256. Client side: -ns:8, -nb: 3584, -bs: 256, -is: 32, -ni: 128



```
[cwd=0]%ls
[6]:./
[2]:d1/
[1]:f1
[1]:f2
[2]:d2/
[2]:d3/
[cwd=0]%serverld
server request: 645
```



```
[cwd=0]%ls
[6]:./
[2]:d1/
[1]:f1
[1]:f2
[2]:d2/
[2]:d3/
[cwd=0]%serverld
server load is:  [55, 26, 530, 47, 34, 124, 26, 49]
```

| Load distribution | Single server requests | Multi server requests | Single/multiple requests | Largest distribute | Smallest distribute |
|---|---|---|---|---|---|
| Small file system | 418 | 625 | 625/418 = 1.49 | 418/73 = 5.72 | 418/331 = 1.26 |
| Large file system | 645 | 891 | 891/645 = 1.38 | 645/26 = 24.8 | 645/530 = 1.22 |

Table 1 load distribution

By analyzing two test case. As shown in table 1, for both small file system and large file system, firstly, the total request number of multiple server is larger than single file system, the reason is that RAID5 use parity to improve reliability, so each “write” data block need to update corresponding parity block, so the total number of server requests increase. Secondly, multiple server system has distributed requests to different server so that requests number of each server is less than single server system, for small file system, the largest distribution ratio is 5.72, smallest ratio is 1.26; for large file system, the largest distribution ratio is 24.8, smallest ratio is 1.22. Thirdly, the request load distribution at multiple server system does not distribute evenly, some block such like inode table, root inode block need to be accessed more. For example, for small file system, block 8 is first block of inode table which locate at server 3, so that server 3 execute most requests, block 10 is root inode 0 which locate at server 2, so that server 2 also has more request. In second case, block 16 is inode table which locate at server 2, block 32 is root inode block which locate at server 5, so that server 2 and server 5 undertake more load. Fourthly, apart from these special block, other server has relative balanced load distribution, also, for large system, each file contains 5 data blocks, so when read a file, the Get request will distribute to server and execute parallel, so that multiple server system will improve the throughput and reduce the total latency when do more operations.

## IV. Reproducibility

This RAID5 system supports running multiple servers from 4 to 8. Each server configuration should be initialized equally, and the client-side total block number should be (total_server - 1) * number_block of one server, the block size should be same as each server.

**1), How to run system**. below is an instance of system configuration

Server side, python3 memoryfs_server.py -port 8000 -nb 256 -bs 128

Client side, python3 memoryfs_shell_rpc.py -ns 4 -startport 8000 -nb 768 -bs 128 -is 16 -ni 16 -cid 0.

**2), For testing the server disconnected and repair**

For example that showing on evaluation section, Firstly, we need to initialize servers and client, then do some operation, then we can quit a server (such like server 2), then do operation in client side, that will print relevant message, we can reference the mapping table(showing on appendix) to check some block such like inode table or root inode block to see if it can response as expected. Then reconnect that server, and in client, use 'repair server_id' to repair server, the file system will still work.

**3), For testing the corrupted block**

Same as the server disconnection test, we quit one server, then reconnect it with a '-cblk', for example: python3 memoryfs_server.py -port 8001 -nb 256 -bs 128 -cblk 2. Then repair that server, do some operation which need to access that block to see if it can print the mapped virtual block number on shell screen, when block is parity, it will print the virtual block number who calling the parity to do updating.

## V. Conclusions

In this project, a client/multi-server file system has been designed and implemented. It uses RAID5 mode to distribute different blocks to multiple servers, so that client side can use the RPC to communicate with each server at same time. That will increase the system throughput and reduce the total system latency. Meanwhile, this file system has the ability of fault tolerance, it takes advantage of parity to mask errors, that will increase the system reliability, it keeps the system operating normally with one server failure. By learning and implementing this design, the importance of system design principle has been demonstrated, the layering and modularity method play an important role for the system abstraction and error isolation. Also, it provides the opportunity to reuse upper layer when modify lower layer.

## Appendix

### 4 servers system mapping table

| Server Block | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | p |
| 1 | 3 | 4 | p | 5 |
| 2 | 6 | p | 7 | 8 |
| 3 | p | 9 | 10 | 11 |
| 4 | 12 | 13 | 14 | p |
| 5 | 15 | 16 | p | 17 |
| 6 | 18 | p | 19 | 20 |

### 6 servers system mapping table

| Server Block | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | p |
| 1 | 5 | 6 | 7 | 8 | p | 9 |
| 2 | 10 | 11 | 12 | p | 13 | 14 |
| 3 | 15 | 16 | p | 17 | 18 | 19 |
| 4 | 20 | p | 21 | 22 | 23 | 24 |
| 5 | p | 25 | 26 | 27 | 28 | 29 |
| 6 | 30 | 31 | 32 | 33 | 34 | p |

### 5 servers system mapping table

| Server Block | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | p |
| 1 | 4 | 5 | 6 | p | 7 |
| 2 | 8 | 9 | p | 10 | 11 |
| 3 | 12 | p | 13 | 14 | 15 |
| 4 | p | 16 | 17 | 18 | 19 |
| 5 | 20 | 21 | 22 | 23 | p |
| 6 | 24 | 25 | 26 | p | 27 |

### 7 servers system mapping table

| Server Block | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | p |
| 1 | 6 | 7 | 8 | 9 | 10 | p | 11 |
| 2 | 12 | 13 | 14 | 15 | p | 16 | 17 |
| 3 | 18 | 19 | 20 | p | 21 | 22 | 23 |
| 4 | 24 | 25 | p | 26 | 27 | 28 | 29 |
| 5 | 30 | p | 31 | 32 | 33 | 34 | 35 |
| 6 | p | 36 | 37 | 38 | 39 | 40 | 41 |

### 8 servers system mapping table

| Server Block | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | p |
| 1 | 7 | 8 | 9 | 10 | 11 | 12 | p | 13 |
| 2 | 14 | 15 | 16 | 17 | 18 | p | 19 | 20 |
| 3 | 21 | 22 | 23 | 24 | p | 25 | 26 | 27 |
| 4 | 28 | 29 | 30 | p | 31 | 32 | 33 | 34 |
| 5 | 35 | 36 | p | 37 | 38 | 39 | 40 | 41 |
| 6 | 42 | p | 43 | 44 | 45 | 46 | 47 | 48 |
| 7 | p | 49 | 50 | 51 | 52 | 53 | 54 | 55 |