Principles of Computer System Design Report

Zhengcheng Song 94555960

Introduction:

In this project, a distributed file system with one client and multi-servers is designed, implemented, tested, and evaluated through Python. This in-memory file system is built with several layers to emulate a physical file system, and several commands acting like an UNIX file system is realized. Furthermore, data is distributed to varies servers with support for redundancy. This functionality is done at DiskBlocks layer, hence flexibility is provided by modularity.

Problem statement:

Traditional centralized file system can no longer meet our desire because larger, faster, and more secure data storage is needed. Although larger disc could provide larger storage, a distributed file system has a more flexible capacity with a variable number of servers. What's more, fault tolerance can be achieved due to redundant block storage. In this project, parity blocks are implemented following a general method for RAID-5. Two types of fault can be well handled by this file system while traditional file system cannot. Either a block is corrupted, or a server is failed, all functionalities still act the same before. Since the load is distributed to different servers, reading and writing should have a faster performance than centralized storage. However, test is conducted on one same computer. Thus, evaluation is done by measuring number of requests handled per server.

Design:

The design of this project is based on assignment 3, in which a multi-clients single server file system is realized. In this project, since multi-servers is needed, naming should be re-designed for DiskBlocks Layer. Different server's ID and their disk blocks number is abstracted as a list of virtual block number. Block number conflict is avoided due to this mapping. The location of parity blocks is also computed through this mapping to distribute parity evenly to all servers. Moreover, Put and Get function have been changed as well. For Put, not only data block will be put into disk blocks, but also parity blocks and a checksum. For Get, checksum of the desired block will be compared to make sure integrity of data. Correct data will be calculated through parity blocks once checksum went wrong. For both Get and Put, only one server failure is under control because of redundancy.

Implement:

1) Memory server:

   Checksums are stored separately from blocks but having a one to one correspondence with each block. Checksum is computed in every Put and Get on server. In Get function, a flag will return to client to indicate whether the data is corrupted or not. One or two running arguments can be accepted with server. One necessary argument is the port number used by server. Another optional argument simulates a corrupted block for fault tolerance testing.

2) Client server:

   A Xor function is realized to computer the exclusive or operation of two byte arrays. This operation will be used in getting parity blocks and retrieving data.

   A new VirtualBlockNumberToServerBlockNumber function is realized. This function receives a virtual block number and returns corresponding server ID, block number and the parity block number for this block. Parity blocks will never be used as normal data blocks.

   For Put function, two extra Gets and an extra Put will be conducted comparing to only putting data into a block as before. The server ID, block number and the parity block number will be return with VirtualBlockNumberToServerBlockNumber function. The new parity data is calculated by exclusive or operations of old data (one Get), old parity data (one Get) and new data. The new data and parity data will be stored into corresponding server and blocks (two Get). If the server is failed, Put do nothing.

   For Get function, VirtualBlockNumberToServerBlockNumber function is used as well. A flag will be returned with data from server. If the Flag is True, data is damaged.
   To handle the situation that data is damaged, or server is failed in Get function, a new function GetFromParity is introduced. Providing non-functional server and block ID, GetFromParity retrieves data from all the other servers with same block ID using exclusive or operation.

   For Get function call inside Put and GetFromParity functions, data block numbers and server IDs are already resolved. In addition, corresponding blocks definitely exist cause examination have been done before. A simple SingleGet function is realized to simplified operation. This function get data giving server ID and block number.

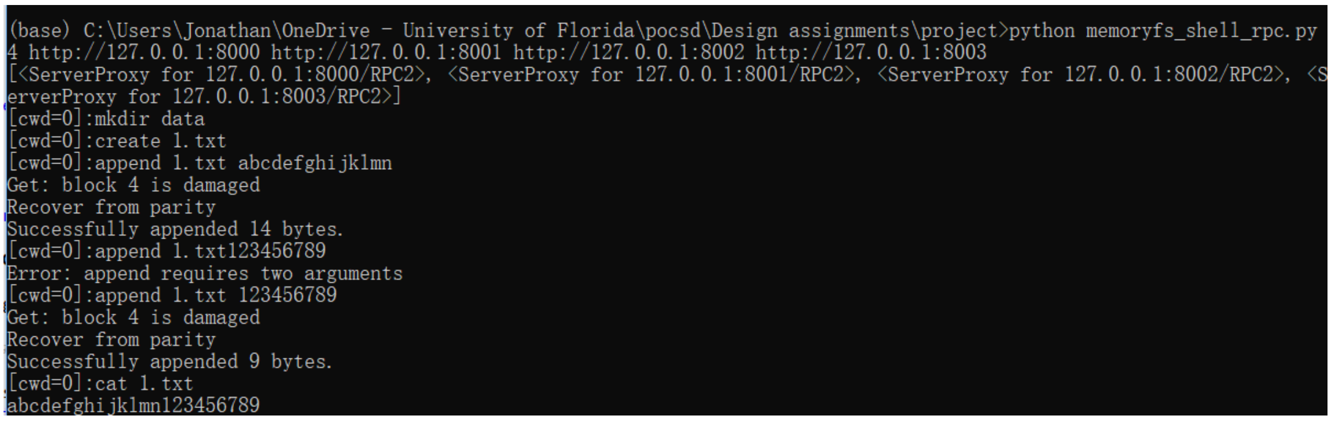
3) Client shell:

   Arguments specifying number of servers and addresses of each server. Some constant in client is determined by number of servers, such as TOTAL_VIRTUAL_BLOCKS, FREEBITMAP_NUM_BLOCKS, INODE_BLOCK_OFFSET, DATA_BLOCKS_OFFSET and DATA_NUM_BLOCKS.

Evaluation:

1) Block corrupted failure:

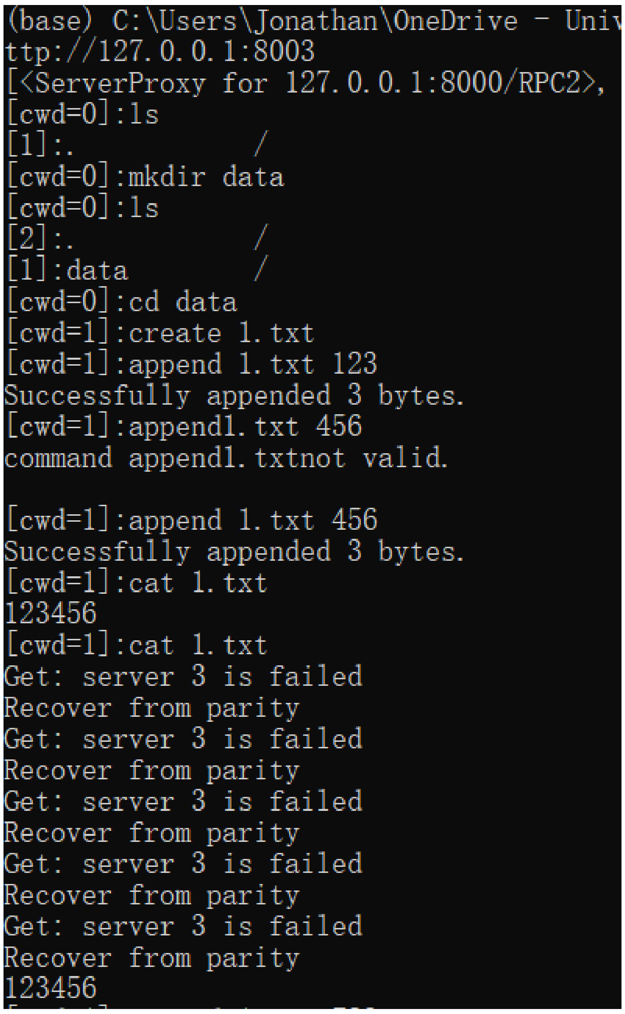
4 servers(0,1,2,3), block 4 is damaged in server 3.



```
(base) C:\Users\Jonathan\OneDrive - University of Florida\pocsd\Design assignments\project>python memoryfs_shell_rpc.py
4 http://127.0.0.1:8000 http://127.0.0.1:8001 http://127.0.0.1:8002 http://127.0.0.1:8003
[<ServerProxy for 127.0.0.1:8000/RPC2>, <ServerProxy for 127.0.0.1:8001/RPC2>, <ServerProxy for 127.0.0.1:8002/RPC2>, <S
erverProxy for 127.0.0.1:8003/RPC2>]
[cwd=0]:mkdir data
[cwd=0]:create 1.txt
[cwd=0]:append 1.txt abcdefghijklmn
Get: block 4 is damaged
Recover from parity
Successfully appended 14 bytes.
[cwd=0]:append 1.txt123456789
Error: append requires two arguments
[cwd=0]:append 1.txt 123456789
Get: block 4 is damaged
Recover from parity
Successfully appended 9 bytes.
[cwd=0]:cat 1.txt
abcdefghijklmn123456789
```

Server failure:

4 servers(0,1,2,3), server 3 is failed during processing:

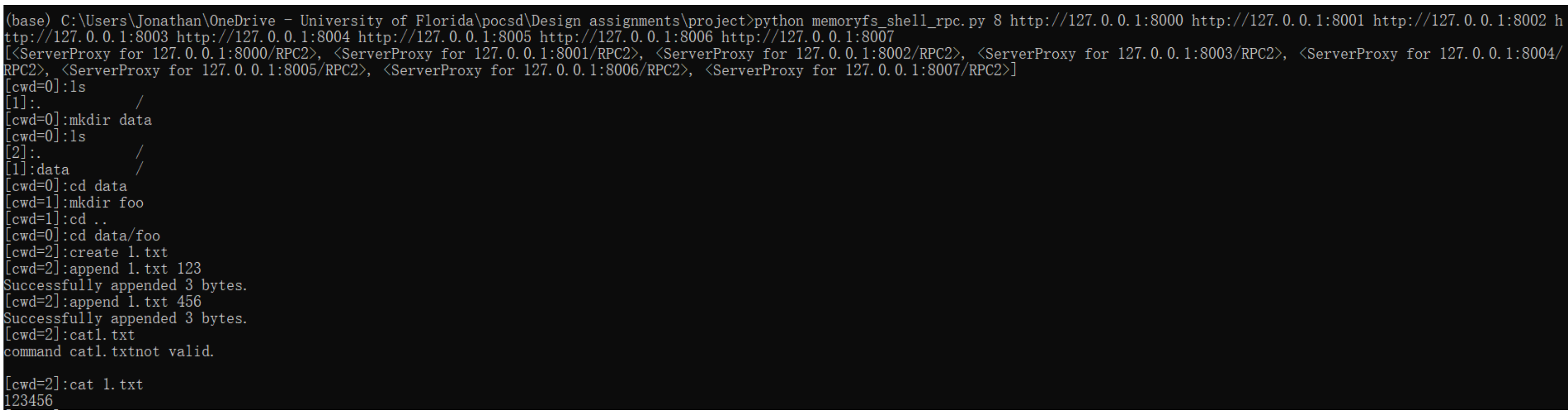


```
(base) C:\Users\Jonathan\OneDrive - Univ
ttp://127.0.0.1:8003
[<ServerProxy for 127.0.0.1:8000/RPC2>,
[cwd=0]:ls
[1]:.           /
[cwd=0]:mkdir data
[cwd=0]:ls
[2]:.           /
[1]:data        /
[cwd=0]:cd data
[cwd=1]:create 1.txt
[cwd=1]:append 1.txt 123
Successfully appended 3 bytes.
[cwd=1]:append1.txt 456
command append1.txtnot valid.

[cwd=1]:append 1.txt 456
Successfully appended 3 bytes.
[cwd=1]:cat 1.txt
123456
[cwd=1]:cat 1.txt
Get: server 3 is failed
Recover from parity
Get: server 3 is failed
Recover from parity
Get: server 3 is failed
Recover from parity
Get: server 3 is failed
Recover from parity
Get: server 3 is failed
Recover from parity
123456
```

2) Different server numbers:

8 servers:



```
(base) C:\Users\Jonathan\OneDrive - University of Florida\pocsd\Design assignments\project>python memoryfs_shell_rpc.py 8 http://127.0.0.1:8000 http://127.0.0.1:8001 http://127.0.0.1:8002 h
ttp://127.0.0.1:8003 http://127.0.0.1:8004 http://127.0.0.1:8005 http://127.0.0.1:8006 http://127.0.0.1:8007
[<ServerProxy for 127.0.0.1:8000/RPC2>, <ServerProxy for 127.0.0.1:8001/RPC2>, <ServerProxy for 127.0.0.1:8002/RPC2>, <ServerProxy for 127.0.0.1:8003/RPC2>, <ServerProxy for 127.0.0.1:8004/
RPC2>, <ServerProxy for 127.0.0.1:8005/RPC2>, <ServerProxy for 127.0.0.1:8006/RPC2>, <ServerProxy for 127.0.0.1:8007/RPC2>]
[cwd=0]:ls
[1]:.           /
[cwd=0]:mkdir data
[cwd=0]:ls
[2]:.           /
[1]:data        /
[cwd=0]:cd data
[cwd=1]:mkdir foo
[cwd=1]:cd ..
[cwd=0]:cd data/foo
[cwd=2]:create 1.txt
[cwd=2]:append 1.txt 123
Successfully appended 3 bytes.
[cwd=2]:append 1.txt 456
Successfully appended 3 bytes.
[cwd=2]:cat1.txt
command cat1.txtnot valid.

[cwd=2]:cat 1.txt
123456
```

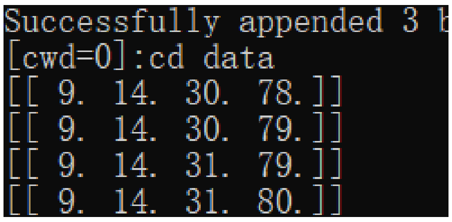
3) Load distribution:

TOTAL_NUM_BLOCKS = 256, INODE_SIZE = 16 (max file size=2 blocks)

Initialize all server load as 0. Every time a get or a put in server is called, add one to corresponding server's load.

Use 4 servers as example:

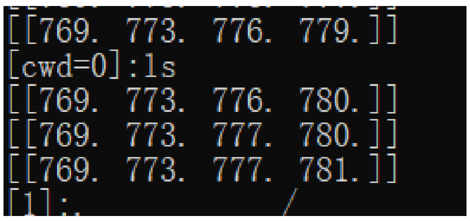
After several steps are done:



```
Successfully appended 3 b
[cwd=0]:cd data
[[ 9.  14.  30.  78.]]
[[ 9.  14.  30.  79.]]
[[ 9.  14.  31.  79.]]
[[ 9.  14.  31.  80.]]
```

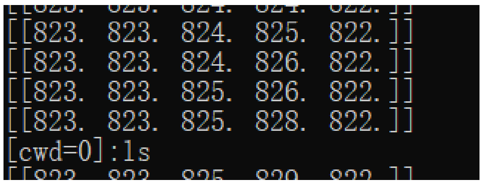
The load is not distributed evenly, since root inode is the most frequently visited inode in those steps.

To verify this, the load is counted with initialization in which inodes are visited almost equally, and parity is also done in initialization.



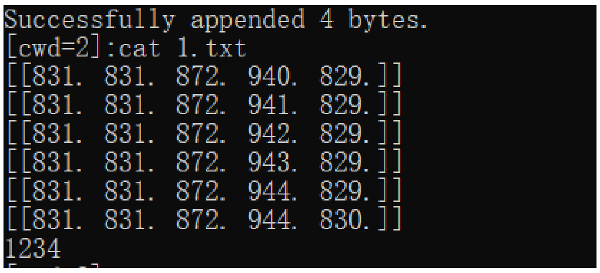
```
[[769.  773.  776.  779.]]
[cwd=0]:ls
[[769.  773.  776.  780.]]
[[769.  773.  777.  780.]]
[[769.  773.  777.  781.]]
[1]:.           /
```

As we know, the root inode is in the fourth block in this file system. To further prove this hypothesis, 5 servers is used. After initialization:



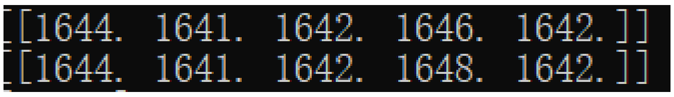
```
[[823. 823. 824. 825. 822.]]
[[823. 823. 824. 826. 822.]]
[[823. 823. 825. 826. 822.]]
[[823. 823. 825. 828. 822.]]
[cwd=0]:ls
```

After few steps:



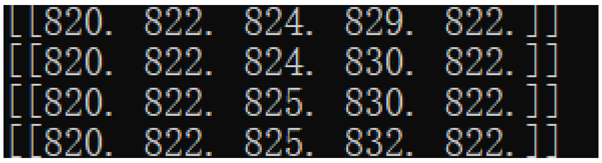
```
Successfully appended 4 bytes.
[cwd=2]:cat 1.txt
[[831. 831. 872. 940. 829.]]
[[831. 831. 872. 941. 829.]]
[[831. 831. 872. 942. 829.]]
[[831. 831. 872. 943. 829.]]
[[831. 831. 872. 944. 829.]]
[[831. 831. 872. 944. 830.]]
1234
```

TOTAL_NUM_BLOCKS = 512, INODE_SIZE = 16 (max file size=1 blocks)



```
[[1644. 1641. 1642. 1646. 1642.]]
[[1644. 1641. 1642. 1648. 1642.]]
```

TOTAL_NUM_BLOCKS = 256, INODE_SIZE = 64 (max file size=7 blocks)



```
[[820. 822. 824. 829. 822.]]
[[820. 822. 824. 830. 822.]]
[[820. 822. 825. 830. 822.]]
[[820. 822. 825. 832. 822.]]
```

Reproducibility

All the argument settings are the same as project requirement. I initialize all the block to start from 0 including parity blocks, then initialize the root inode.
For load count, memoryfs_shell_rpc_loadcount.py can be ran with same arguments. Load will be counted and outputted after every get or put from server.

Conclusion

A distributed file system with one client and multi-servers is realized in this project. This file system is able to handle single block corrupt in one server or only one server failure. Through this project, I understand more about the importance of layer and modularity, since we only need to change the DiskBlocks layer to implement this project from previous homework; I understand more about the advantage of redundancy to tolerance fault; I also find the advantage of distributed storage that is it can reduce load leading to faster performance.