PoCSD Final Project Report

Rajandeep Singh
Electrical and
Computer Engineering
University of Florida
UFID: 41917909

Mayank Gupta
Electrical and
Computer Engineering
University of Florida
UFID: 48998256

6 December 2019

1 Design

The whole design is based on the previous homework assignments. As compared to the homework 4 files additions were made largely to the client_stub.py and inode_layer.py files.

1.1 Inode_layer.py

This layer basically considers the total size of the block as (number of servers – 1) times the size of a block in each server.
This is in the case of RAID 5 setup and one of the servers is used for storing parity and all the other servers are used to store data.
Before reading and writing data from any file the block which are to be read or which are to be written into are displayed. After that a programmable delay as provided by the user using the command line arguments, the interface continues to read and write data.

1.2 client_stub.py

Client_stub.py was updated in order to handle the different servers and to handle any kind or network error due to a non-responding server or decayed data inside a server.
Also, MD5 checksum is implemented in this layer. Data read is first verified using the checksum and if it is correct then it is used further. If the checksum fails or a server does not respond to a request then this layer also has provisions to recover data from the failed or decayed server.

2 Workflow

Initially all the blocks are set to null and their checksum value is also set according to a block with all null values.
When we create a file or directory the inode tables in all the servers are updated and all the servers have the information regarding the directories and files inside the system. But the data value is different in all of the servers.
Whenever a file is to be written the data is divided among the servers and is written in slices in each of the servers. One of the servers is selected to write parity infomation. The server for parity is selected by using the modulus operation on the block number. For example if the block number is 42 and the number of servers is 5, then 42%5 will give us 2. This defines that the parity is written inside the server indexed 2. Before any data is written, the system states that to which servers the data and parity is going to be written. After a programmable delay provided in the command line, the data is written into the said blocks. Also, the parity is calculated by xor operation on all the slices of data written into blocks of different servers mapped by the block number of the top level system.
The feature of using xor operation for parity is that if only one slice of the data is changed then in order to change the parity we do not need to read all the servers again. We just need to read the old data and xor it with the new data and the old parity. This simple operation gives us the value of the new parity.
For read operation the virtual block number is fed to the read function which is mapped to the corresponding physical block number of all the different servers. This gives us all the slices from the servers apart from the one with parity. The data is combined and returned to the user. The parity value is only read in case of a fail stop, or data decay. Also, before any data is read, the system states that to which servers the data and parity is going to be read from. After a programmable delay provided in the command line, the data is read from the said blocks.
For a status operation if one of the servers fails then the system is able to recover all the data blocks from the failed or corrupted server using the parity value.

3 Data recovery

Our system handles two types of failures.
1. Fail stop
2. Data corruption due to data decay

In both of the situations the system recognizes that a server has failed and has provided the wrong value. The data lost because of the failed server is recovered using the parity server. While running the UI.py file, which is the user interface file, the system very clearly states from where data is being fetched and if there is a failure in the one of the servers.
The case of data corruption is emulated by corrupting the checksum values of a

particular server which is provided as an input to the backchannel.py file which is being used to serve as the script to start different servers.

NOTE: data recovery is only possible if only one of the servers fails and not more.

4 Checksum

MD5 checksum is used and with every block of data a 16 bit checksum is also written. This increases the actual size of the block 16 bit more than what is actually available to write data into. This is handled by changing the disk layout file.
While writing every block along with the data the checksum value for that particular data is calculated and is written along the actual data at the end of the block inside the 16 extra bits that we allocated in the disk layout file.
While reading a block its checksum value is verified and if does not match then the block is marked as corrupted. In such situation the value from all the other servers is read along with parity of that particular block and the missing data is calculated using xor operations. The decay extra credit is also implemented using the checksum. The decay is emulated by actually changing the checksum value written inside the blocks such that on read the checksum verification fails.

5 Virtual Blocks

A virtual block of the collective system has the size as (Number of servers – 1) times the block size for a single server.
The virtual blocks are mapped to the physical blocks inside the servers. The mapping from virtual to physical servers essentially corresponds to the same block numbers among all the physical servers but with smaller block size.
This is a very simple mapping and the parity server is also calculated using the block number of the virtual block as explained earlier.

6 Testing

A number of unit tests were written to test out the behaviour of the system in all kinds of situations. These test cases can be essentially summarized as follows:
1. Writing very long strings of data into a file.
2. Reading long string of data from a file.
3. Writing in between the already written files.
4. Corrupting the data using data decay (corrupts the checksum value).

5. Verifying the behaviour in case of a fail stop of a particular server.
6. Moving of files from one directory to another and performing all the read and write operations mentioned above again on the moved files.
7. Removing a file and trying to read it again.
8. Data recovery in case of a failure.

7 Instructions to use

1. Run backchannel.py
Usage: *python backchannel.py < number of servers >*
2. Run UI.py
Usage: *python UI.py < number of servers > < programable write delay > < programable read delay >*
3. After the file system is initialized a $ sign will appear which actually indicates that the system is ready to accept input.
4. Input different commands like:
a. *mkdir < directory to be created >*
Example: *mkdir /A*
b. *create < file to be created >*
Example: *create /A/new_file.txt*
c. *write < file to be written to > < data to be written > < offset >*
Example: *write /A/new_file.txt "hello World!" 0*
d. *read < file to be read > < offset >*
Example: *read /A/new_file.txt 5*
e. *rm < file to be removed >*
Example: *rm /A/new_file.txt*
f. *mv < file to be moved > < new location >*
Example: *mv /A/new_file.txt /*
This command actually moves to file new_file.txt from directory "/A" to root directory "/" with the same name.
g. *status*
NOTE: The grammar of all the commands is very strict and diverting from the style of format given in the examples might result in errors or unexpected behaviours.
5. A server's data can be corrupted by providing the server number and not the port number in the terminal running backchannel.py. This will corrupt the checksum value of all the blocks inside that particular server. Hence, essentially signifying data decay.
6. A terminal listening for a particular port-number can be directly shut down signifying fail stop.
7. The main system will continue to run and recover even in case of a failure.

8 Performance results

8.1 Reliability

In case of a single server system the system is not reliable in case of a failure as there is no way to recover data. But in RAID5 implementation we do have a parity written in different blocks such that if a server fails then we are able to recover the data lost from that server. Therefore, RAID5 implementation provides greater reliability by using parity.

8.2 Speed

In case of a single server system all the requests are handled by the only server. Therefore, every time for writing N data blocks we have to hit the server N times. Whereas in case of a RAID system, in order to write N blocks on average every server is hit N/S times (S being the number of servers). This can particularly help in increasing the speed of the system if these different requests are handles by different threads. Essentially increasing the speed of the system by approximately S times.

8.3 Extra credits attempted

1: Checksum
2: Data Decay