EEL - 5737 Principles of Computer System Design

Fault Tolerant Distributed FUSE File System

University of Florida

Table of Contents

Abstract

Reduction of load from data servers becomes important when the amount of data to be written becomes significantly high. This made the need for multiple remote data servers. Thus, file systems with distributed remote data servers and one meta server allows the client to create a

robust system and allows the client to handle fault tolerance and redundancy[3]. Such distributed file system was developed using XMLRPC servers. The design ensured it handled data corruption, server crashes thus providing fault tolerance. 3 Replicas for each data block was made to ensure recovery from data corruption. Back-up to each data servers was provided using persistent storage. All the operations related to file system were implemented and tested. The system design was tested for corner cases as discussed at the end of the report. The design of the system proved to be robust and handled all extreme operating conditions, thus providing a complete distributed File system with multiple data servers and one meta servers.

Problem Statement

The main objective of this project is to distribute data across different data servers to reduce their load and address fault tolerance, redundancy etc. Design of the distributed client-server File System should be robust and should account for the following specifications

Spec 1 – Design a distributed client-server system with "N" Data Servers and 1 Meta server.

Spec 2 – Data to be stored as blocks of 8 bytes each.

Spec 3 – Starting block (0) to be stored at a specific server found using a hash function and the subsequent blocks to be stored in a round – robin fashion.

Spec 4 – Handle Data collision when writing to same server

Spec 5 – 2 Redundant copies of same block are to be stored in the 2 next servers. Hence there will be 3 Servers carrying the same data.

Spec 6 – The data server should store the data in a persistent storage to allow recovery on crash.

Spec 7 – When a data server recovers from a crash, it should load the data from its persistent storage.

> **Spec 7.1** In case if the persistent storage is completely lost, the data server recovers from the replicas stored in other servers.

Spec 8 – Handle Data corruption

> **Spec 8.1** – Use checksum to verify the data in data-server
>
> **Spec 8.2** – When there is a read operation, the client will read randomly from any server which contains the data (Replica 1 or 2 or 3). The client system should verify the checksum and if there is a mismatch, it should continue reading from any other replica. When correct data is found, it is stored in the server which consisted of inconsistent data.

Spec 9 – To corrupt one data block intentionally and test the performance of the client-server system.

Spec 10 – Dealing when server is down

> **Spec 10.1** – All writes should write all Replicas and receive acknowledgement

Spec 10.2 – When trying to write to a particular server, if it is down, then the client system should try writing until it becomes available.

Spec 10.3 – Read should successfully return the data if any one of the replica's checksum matches

Spec 10.4 – Read should succeed when non-adjacent servers are crashed. Write should be blocked even when a single server is down

Spec 11 – Test the system with N>=4 Data servers and three replicas

Background

Architecture Design

Figure 1 shows the overall architecture of the distributed File System with 'N' data servers and one meta-server. The design follows client-server approach **[Spec 1 accomplished]**.

- **Meta-Server**

  The meta-server stores the metadata of the files/directories being created. This holds the keys i.e. path of files/directories and the values are the metadata corresponding to the files/directories. When a file is invoked by the client, the path of the file is taken as the key and the metadata corresponding to this path(key) is returned [Assignment 4]. Meta-server is assumed to be very reliable and stable [Project]. A XMLRPC based server is used for the purpose [1]. The project consists of 1 Meta-server as shown in Figure 1.

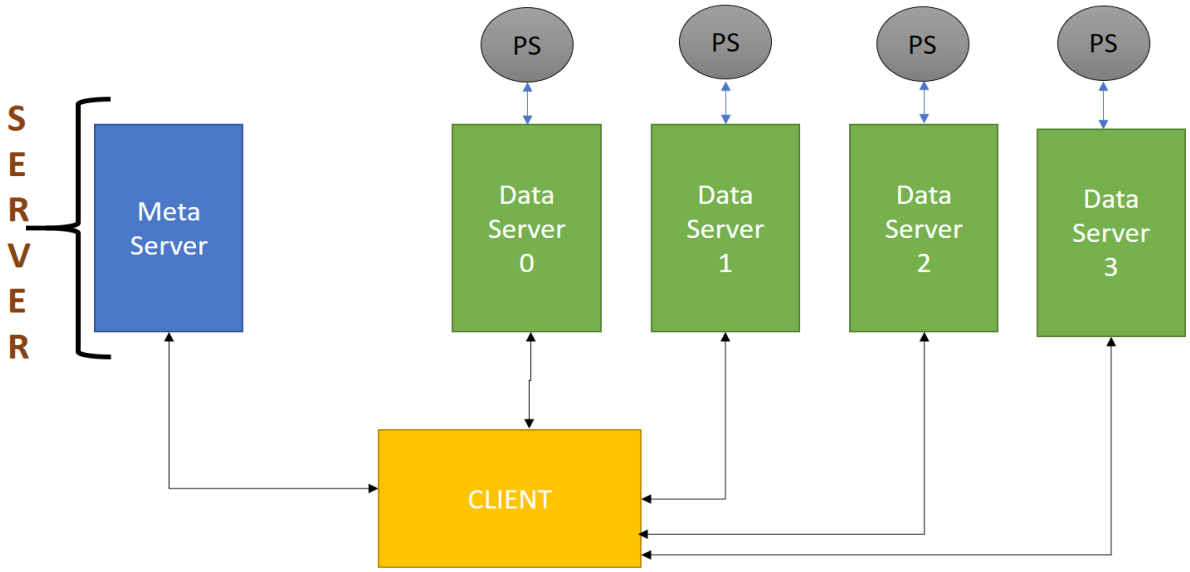


Figure 1. Distributed File System with client-server design

Here PS – Persistent storage

- **Data-Server**

  Data server is used to store the content of the file created. As the project involves design of distributed File System with 'N' Data servers, the data for a file/path is distributed into blocks of data and each block is stored in consecutive data servers in a round-robin fashion. The starting data-server is defined by a hash function. This helps in achieving load balancing. The same data is stored as redundant data in the next 2 data servers (Replica 2 and 3) as backup to account for data corruption.

  Each server stores its information in a persistent storage (hard disk) to allow recovery on crash. Whenever a server updates its data, it updates it in it's local disk as well. This allows recovery of the data-server upon a crash.

  When the data-server gets the path of a file and the block number it needs, it returns the data to the client.

- **Client**

  All the operations such as reading a file, writing to a file, renaming a file/directory, deleting a file/directory, resizing or truncating a file, making a directory/file, etc., are all controlled by the client. Since remote distributed File System is being developed, the client passes appropriate information/data/commands to the server (data & meta) and performs the required operation.

Working – Big Picture

Meta server is designed to have a dictionary whose key will be the path of the file and its value will be the metadata. It will return the metadata when it's corresponding path is given.

Since, the objective of the project is to create distributed File system with "N" data servers, data is fed into the data server in a specific defined manner. Each data block is stored as 3 Replica's (Replica 1, Replica 2 and Replica 3). Also, each block is to be associated with a checksum to handle data corruption. Hence each Data server consist of 2 Lists with each containing 3 Dictionaries. List 1 corresponds to the Data and List 2 contains the corresponding checksum with one-to-one correspondence with L1. L1 has 3 dictionaries. The reason is from the problem statement it is clear that, each data has 3 Replicas and each replica is stored in subsequent data servers in a round robin fashion. To distinguish between R1, R2, R3 we have 3 dictionaries. Always R1 goes into the first Dictionary (D1), R2 to D2 and R3 to D3, where D1, D2 and D3

follows a round-robin fashion. Thus, when a R1 copy of data block say B_M is writing again in a data-server where R1 of B0 of the same data has been written, this B_M gets appended to the first Dictionary of L1 and its copies are appended in an equivalent manner **[Spec 4&5 accomplished]**.

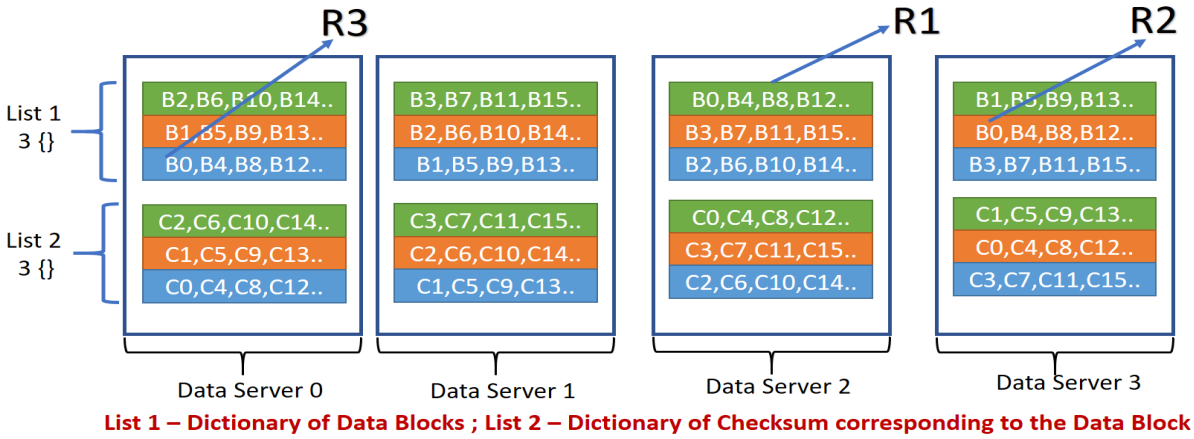


Figure 2 – Example of how data and checksum blocks are stored

Each data block comes with a checksum that is created using a md5 hash library. Thus, whenever a specific data block gets stored in the data-server, it's corresponding checksum also gets stored in the corresponding dictionary maintained by List L2. To enable recovery from crash, the server copies its content to a persistent storage. In this program, we have made use of shelve [2] to achieve this. Thus, whenever there is any write operation or update operation in any data-server, the same gets updated in the persistent storage also. Thus **L1[{D1}, {D2}, {D3}], L2[{C1}, {C2}, {C3}]** forms the storage of each data server. These get copied to the persistent storage also.

Consider an input text data separated into data blocks B0 to B12. The starting Data server is found to be Data Server 2. Figure 2 depicts an entire write operation **[Spec 3 accomplished]**.

The client writes blocks of data into the data server and copies the metadata of the file into the meta server following the client-server architecture designed. Since, the design has "N" data-servers, the client writes Block 0 in a specific data server found using a hash function.

- **Write Operation**
  When the user wants to write to a file in a directory, the client breaks the file's data into blocks of 8 bytes (fixed in this project) and looks for the offset. The starting data server number where R1 of the offset is stored/is to be stored is found and data is sent from the client to server to be written starting from that server, one block in one server following a round robin fashion. Since we use a XMLRPC protocol the data that is transferred from client to server is converted into a sequence of string and then to binary format. This upon being received, is decoded back to string and back to the original format. This is done using the following statement
  - **Client - Binary(pickle.dumps(data))**
  - **Server – pickle.load(data)**

Replica R2 and R3 are stored in the next 2 servers in a round robin manner. Checksum is also stored in an equivalent way and the persistent storage is updated. An acknowledgement is sent from the server to client only after which the client proceeds to the next operation.

- **Read Operation**

  When a read request comes to the client, first the starting point (offset) and the end or the point up to which the file to be read is found. Next, we find which data server which contains R1 of this data block. Then we find the number of blocks to be read based on the size of the file to be read. Then we create a list which stores the order of Data servers to be read to read the complete data. Now we find the starting Line number within the first block and ending line number of the last block. Since the client system can randomly read from any data server containing any replica of the data block (R1, R2, R3 of B0), a random function is written to randomly choose one copy of the replica. Then it sends the block number and path to that randomly chosen Data server. This data-server is checked for shut-down. If this particular data server is down, the client gets an acknowledgement and looks into any one of the remaining 2 replicas. If the data-server is not down. If data in that path is not present it sends corresponding acknowledgement to the client. If the requested block is present in the path, it is returned along with its checksum. Checksum for the returned data is calculated using the md5 hash library and is compared with the checksum returned from the data server. If a match occurs, the data is read from the block. If it is the first block, data is read from the line number found. If it is the last block, data is read until the end line number. Entire data is read during remaining conditions.

  This procedure is continued to read all blocks chosen randomly from one of its replicas. When the checksum does not match, the client remembers the data-server. It searches in the available 2 data-servers to look into the remaining 2 replicas. If checksum match is found in one of the cases, the data block along with the checksum (Previous Replica(s)) is replaced with the correct data block and checksum. Whenever an update happens to the data-server, it is copied/updated in the persistent storage as well **[Spec 8 accomplished].**

  When Data-server crashes [Spec 7 accomplished]

  When a data-server is down, all its data is erased from the server. When it boots again, it tries to load the entire data from it’s persistent storage before performing any operation. If the persistent storage of the corresponding data-server is empty and if the persistent storage of the data-servers adjacent is not empty, then data is copied from it’s adjacent servers, but it is ensured to store the replicas in corresponding dictionaries. It is then written back to the persistent storage.

  When a read operation happens when a data-server is down, the client tries to read from any data server that contains the replica of this data. It returns the data block if available. The client checks for the checksum. If a match occurs, it proceeds and get the data block. If a match doesn’t exist it tries to read from the next available server and continues until it gets valid data. Until it gets valid data, it keeps checking. If a server that was down

during this time boots and fetches the information from its persistent storage, and if the checksum matches for a read from this server, the read operation stops. It also updates the server that had wrong checksum.

When a write operation occurs, until all the servers is available, it keeps waiting. Only after all the servers that were initially down reboots, writing in all the 3 data servers happens, and the control returns.

Unlink – Performs remove file operation. It removes all the key elements with the path in the data server including R1, R2, R3. It updates the parent path by removing the file as its child. It also removes the meta data of this path.

Rmdir – We must remove the directory if there is no files/directory/links present in this directory. If empty directory, remove this name for parent's child list, decrease the hardlink of parent directory and remove the entry from self.files.

Mkdir – To support multilevel file system, a child list entry is added to self.files (This is present only for directory and not for files or symlinks). The given path is break down as parent and filename using the function par_child. Filename can be file or directory or symlink, parent will always be a directory. When creating a file, directory or link, it was ensured that the file name is added to the child list of the directory. Overall this involves associating files with parent, incrementing the hardlink of parent by one, creating
empty child list, setting the st_mode as directory

For ex: path = /m/d1/d2/f → parent = /m/d1/d2, filename = f
when creating the file or making directory/links:
The self.files['/m/d1/d2',]['child'] list will be appended with filename f.
then a meta data created for file using the whole path '/m/d1/d2/f' → self.files['/m/d1/d2/f'] and correspondingly setting st_mode and st_nlink.

Symlink - involves associating the file with parent. Setting the st_mode accordingly for links and files.

Rename – There are 2 cases

case 1: file/link
Remove the name for current parent child list. Add the name to new parent child list if the filename already not present (overwrite case). Change the data for the file as well, remove the old name and add the new name. self.data[new] = self.data.pop(old)

case 2: directory
ex: mv /m/dir1/dir2 /m/dir3
old_par = '/m/dir1', old_name = dir2
new_par = '/m', new_name = dir3

First we need to check dir3 is already present in new_par. If true: then we need to check whether dir3 has any child. If dir3 has child, then we can't move dir2 as dir3 (we can't remove directory until its empty). If dir3 is empty or dir3 isn't already present we can move the directory. We need to move all the contents of dir2 to /m/dir3 recursively. We need to decrease the hard link for /m/dir1 by one and increase the hardlink for /m based on adding a new directory or replacing the empty directory. In replacing case the hardlink will remain same and for other case we will increase it by 1. We also must rename the contents of the directory recursively by calling this rename function again, base case for this is empty directory or file.

Truncate – Basically this operation finds the length to which the data has to be truncated. If the length of the file is lesser than truncate length, it is appended with null characters. If the length to be truncated is lesser than the file size, the data is read using read function and all the blocks after the truncate length is removed and updated.

Whenever a file is read, each block is joined using the **''.join(data)** command. This joins the content of a block to a string. And each string is appended at the end.

Implementation

Client

Since this being a client-server design, the client side (where the user operates) takes care of all operations the user needs to do on the File system located in the server. The user assumes all the operations and storage is done in the client side. The design hides or encapsulates all these details from the user and just provide an interface for the user to perform the following operation. But the client uses XMLRPC protocol to pass the information/data to the server and completes the operation. The server acknowledges by returning an acknowledgement which defines the success/failure of an operation (e.g. True/False, Empty Dictionary etc.)

- **Functionalities**
  - **Data as blocks of fixed size**

    This was done as part of Assignment 2 and the same implementation is used here. When the user tries to write contents in the file, the write functions first splits it into blocks of length 8 bytes and the collection of these blocks are stored in a list as shown below

    split_data = [data[i:i+self.blocksize] for i in range(0, len(data), self.blocksize)]

    Here self.blocksize is defined as a global variable of size 8 bytes. Thus, the above function splits the data into a list of blocks with each 8 bytes long. The final block may or may not be filled. Each element in a block is referred by "**Line Number**" in the program.

This is used to identify if the block is filled or not and If not filled, the line number tells the client write program how many it can still save. For example, if the user tries to write "This is POCSD Project Fall '17", the **split_data list will contain ["This is ", "POCSD Pr", "oject Fa", "ll '17"].** Here blocksize is 8 and the lists contains 4 Blocks and the last line number of final block is 6, which means it can store 2 more bytes of data **[Spec 2 Accomplished]**

- **Write Operation**

When the user tries to write into a File, the write function from the client side breaks the data into blocks of fixed size as mentioned above. Now four cases are to be considered.

- **Offset = 0 and File size = 0**
  Writes into an empty file. Calls **put_data_segment()** function passing the path and split_data, for this starting blocks are zero. (this is explained below)
- **Offset + length of new data < File Size**
  Here, the new data overwrites the previously existing data. Hence, first we need to find the starting block number of the existing file on which the new data must be replaced. To find this, the write function calls the "get_data_segment" function which returns the first block from which the new data must be over-written.

  Get_Data_segment () – This function calls the following functions to obtain these data

  Startblocks(path, startblocknumber) – Returns the starting block number from a particular data server based on the hash function defined in the project.

  Ser_Order(path, startblocknumber) – Returns a list which consist of the order in which the data servers need to be accessed for writing the data starting from the startblock as obtained above.

  Calls Get_Block(path, starting block, ser_order, count) – This function performs the following

  1. Randomly choose a data server from the possible 3 data servers
  2. Check if the data server is available. If yes, check if the data is available. If yes, read the data and check for the checksum. If the checksum matches return the data. If not, continue to search in other possible data servers.
  3. If the data server is not available, search in the next possible server from the ser_order list.
  4. If checksum matches replace the content in the server for which the Replica's checksum failed and the server didn't crash.

Once the results are obtained after calling these function, the get_data_segment returns these details to the write function. Now the write function has the new data separated in blocks and knows the target blocks to be replaced. These information is passed to the **put_data_segment()** function to write it in the data server.

- **Offset > File size**
  In this case, the file is appended with "**NULL CHARACTER**" from the end of file to offset-1. Consider this as a new input data 2 (series of Null characters) and append it before the actual input data to be inserted (input data). Like above, call the get_Data_Segment() function and after getting the appropriate data, call the put_data_segment() function to write the data into the data server.
  If the File is empty, then you are writing this new string in an empty file which is case 1. Call put_data_segment() and starting blocks are zero.
- **Offset <= File size and offset + length of new data >= File size**
  It works similar to the above function by starting from the offset and calling the **put_data_segement() function.**

  Put_Data_Segment()

  - Calls put_block() function after obtaining the starting block and order of the data servers to be filled
  - Put_Block() function, get the checksum for the data and pass it to the Data server using XMPRPC protocol, it also handles the replication of data blocks to different servers and retrying.

- **Read Operation**
  - Calculate the starting block and ending block and the line number
  - Calls get_Data_Segment() function as explained above and gets the blocks needed and removes the unnecessary line numbers from starting and ending blocks.
  - Merge all the data using join function and print it to the user
- **Truncate Operation, Unlink, Symbolic Link, Remove Directory, Rename, Read Link, Read Directory, Make Directory, Create, Get attribute** functions make use of the get_data and put_data function to get the data and after that, they work the way explained above.

- **Data Server**
  - **Design**
    The init() function, initially defines the lists and dictionaries for the data server. It tries read from file if it throws exception, it creates an empty persistent storage using shelve (shelve_initialize function). Initializes empty data server and its corresponding empty persistent storage at the beginning. In the case of server crash and persistent storage not present it gets data blocks from adjacent servers using get from servers function. Whenever a write operation comes, it stores in

the corresponding L1 dictionary and L2 dictionary. When a read operation comes, it reads the dictionaries, and sends it back to the client.

- **Functionality**
  - **Initialize Function**
    Initializes all the data servers and its persistent storage to be empty. Initializes the data structure for each data server.
  - **Shelve**
    - **Initialize –** Creates empty persistent storage. Creates a dictionary of 2 lists. Each list being a list of dictionaries for data and list of dictionaries for checksum.
    - **Read –** Copies the entire dictionary back to the data server.
    - **Write –** Updates the corresponding dictionary within the dictionary of lists (List 1 for data and List 2 for checksum)**[Spec 6 accomplished].**
  - **Get from Servers –** Returns the data and checksum from the specified server with specific replica and updates call the server data if present.
  - **Get Replica –** When the user calls for replica, this function returns the data
    and checksum for this replica
  - **Put Block –** Put the corresponding block of data and checksum in the specified dictionary
  - **Get Blocks –** Returns the specific block inside the dictionary of replica specified in the argument (both checksum and replica)
  - **Delete –** Deletes the specific file by deleting all the blocks associated with it by popping out the dictionary key of that path
  - **Delete All –** Removes all the blocks of all the files inside the directory or inside the path given
  - **Corrupt –** This function corrupts a random block chosen from a random data server by changing the checksum value. **[Spec 9 accomplished]**

- **Meta Server**
  - **Design**
    The design of meta server design is straight forward. It consists of a dictionary with the path as its key and the meta data as its value. It returns the metadata of the path when requested and creates or modifies the value of a particular key(path).
  - **Functionality**
    - **Get, Put, Delete, Delete All –** Operates as mentioned above.

Testing & Observation

We have tested all the normal file system operations:

1. reading file from current folder and from some other relative/absolute path

2. writing – both appending to file and overwriting the same file
3. truncating the file to smaller size and larger size
4. renaming the file and folder.
5. copying and moving the file/folder
6. creating the symbolic link for the file and folder, and accessing the file using the link
7. creating the directory and checking the hardlink for the parent directory, (hardlink should be incremented by 1 when new folder is created)
8. tested the hierarchical file system by comparing with the linux file system, wrote a test script for testing.
9. We also **wrote a C program** to seek a file and overwrite a particular block and read the file block.

```
echo hello > one.txt
echo world how are you >> one.txt
cat one.txt
echo four > four.txt
cp one.txt two.txt
mv two.txt three.txt
cat three.txt
stat .
mkdir folder_one
stat .
mv one.txt folder_one
ls folder_one
cd folder_one
cat one.txt
mkdir folder_two
stat .
stat folder_two
cd ..
rm folder_one
stat .
stat folder_one
cp -r folder_one folder_one_cpy
ls -LR
rmdir folder_one_cpy
rm -r folder_one_cpy
ls -LR
mkdir folder_four
mv folder_one folder_four
ls -LR
cat folder_four/folder_one/one.txt
rm three.txt
ls
mkdir st_mode
ls -LR
```

rmdir st_mode

For the specifications described the project:

We manually tested the file crashing: for up to 3 crash and various combinations server crash and checksum errors

Ex: 1. servercrash1, servercrash2, checksum error
2. servercrash1, checksum error, correct checksum
3. servercrash1, servercrash2, servercrash3, …..

We checked the server's redundancy specifications by manually crashing the server and also deleting the persistent storage. In case of server crashing and persistent storage doesn't exist we checked that the server got its data from the nearby replica (by using the print statements), in this server needs to get **its own data and the other two replica which it stored previously.**

For Data corruption, we first stored some data to the file, corrupted the file using the corrupt.py program (written by us using the specification provided) and we again read same file. When we tested this, we found that our program corrected this error.

During the server crash we tried to write a block to server which was crashed we observed that our program was blocked. Server crashing can be identified using the exception which we get when we try to write block.

```
Retry = 1
while(retry):
        try:
                write to server;
                retry = 0
        except:
                retry = 1
```

Reading a block when a server crashed, in this case when there is not a correct data from all the servers it should get blocked, even if a single replica is good it should proceed I.e it should not get blocked. We observed that our program got blocked when the consecutive three servers are crashed, and for the combination of server crash and checksum errors. In combination of server crash and checksum we **should not try to correct the server which is crashed otherwise the program will hang (blocked)**.

```
Retry = 1
while(retry):
        Servercrash = [0, 0, 0]
        for i in range(3): // three replicas
                try:
                        res = read from the replica I
                except:
```

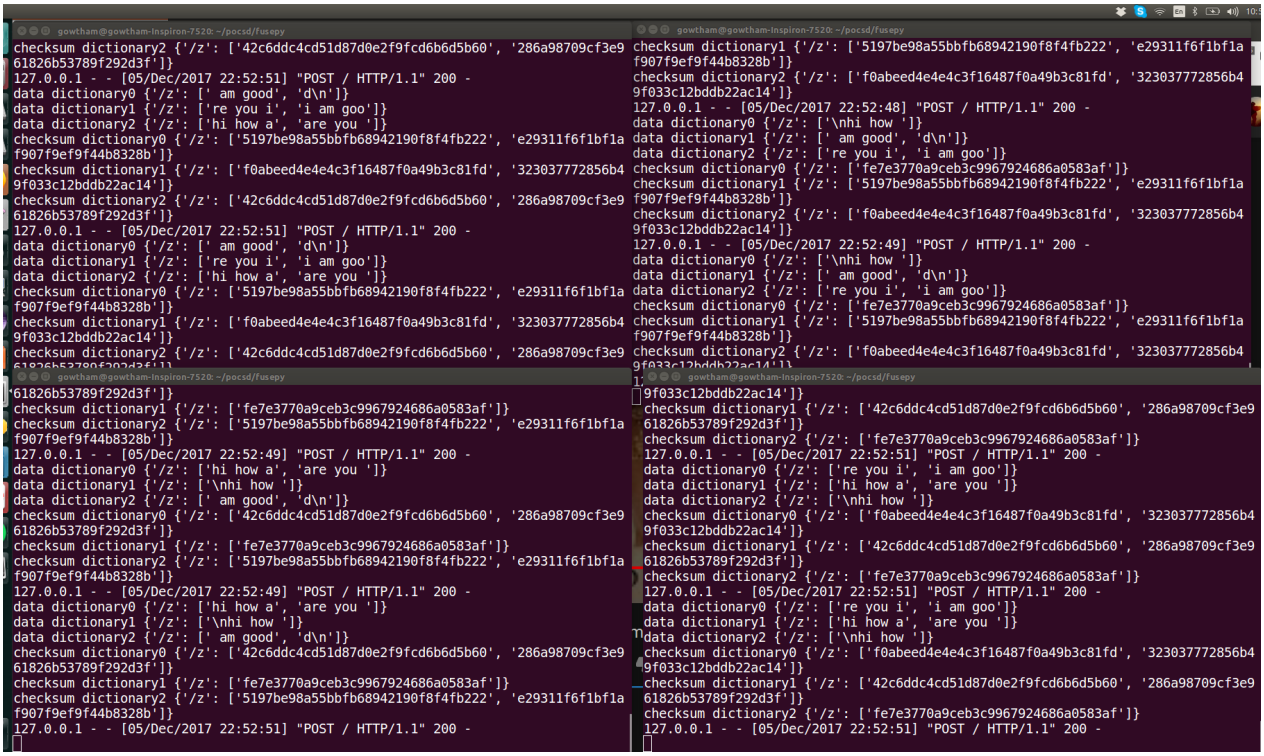
```
                servercrash[i] = 1
                continue
        if (checksum(res.data) == res.checksum)
                retry = 0
                for k in range i:
                        if(servercrash[k] == 0)
                                write the correct data and checksum
```

While testing **we came across an interesting case which is not mentioned in the project pdf** and the case is **server didn't crash and persistent storage doesn't exist i.e. someone deleted the storage.**

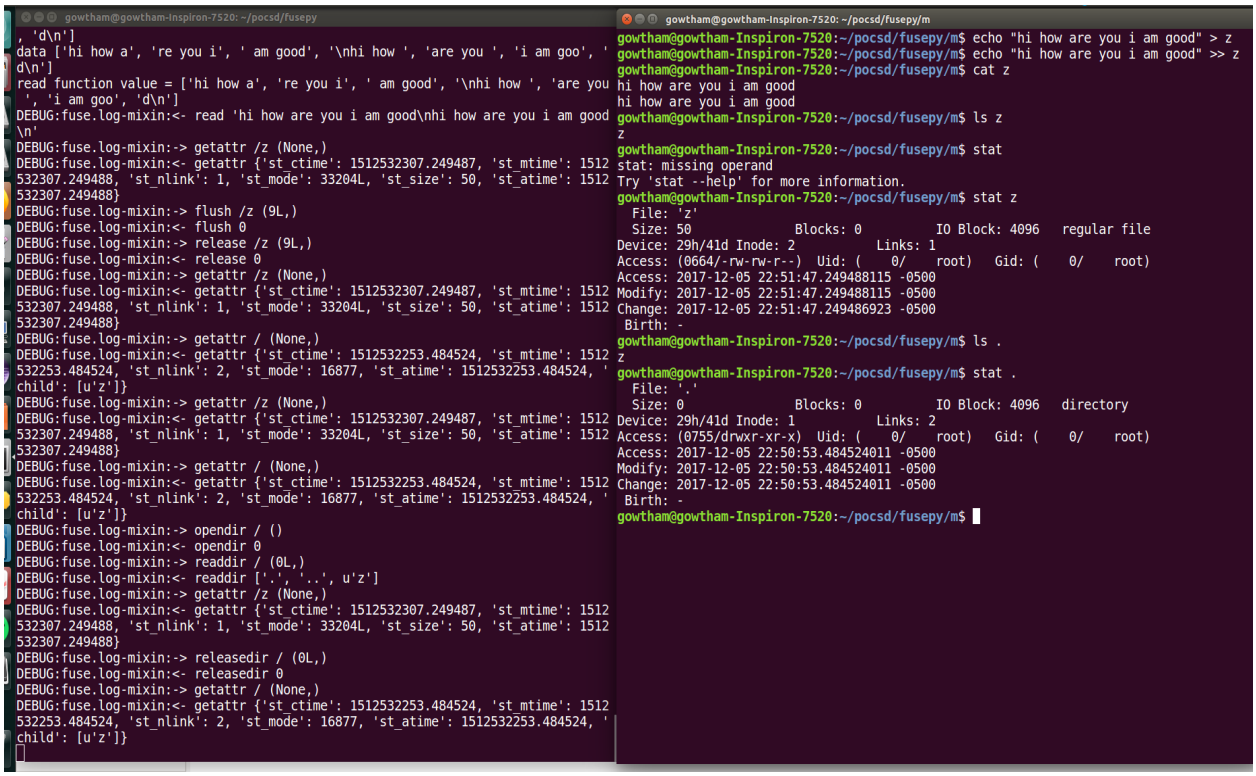
- To address this, we need to check the persistent storage exist or not while reading if it doesn't exist then we need to initialize with the data existing in memory. Server didn't crash, so memory is not affected; if server had crashed it would have followed the normal procedure **[Spec 10 accomplished]**

We tested our program for 4 and 5 data servers and with 3 replicas (own data, replica1, replica2)

Sample Output:
4 data servers (meta server not shown) **[Spec 11 accomplished]**



Fusepy daemon and accessing the file system mounted at m



Conclusion

Thus, the distributed file system with multiple data servers and one meta server was designed covering all the design specification mentioned in the problem statement and the performance was tested for various testing conditions. A number of corner cases were tested, and the system proved to be robust in all the testing conditions. It can also be seen throughout the development stage of the client-server architecture all the 11 Design specifications were considered, implemented and tested.

References

[1] https://gist.github.com/gauravyeole/b522d9c978a14f0128afb823c0cde535

[2] https://docs.python.org/2/library/shelve.html

[3]https://ocw.mit.edu/resources/res-6-004-principles-of-computer-system-design-an-introduction-spring-2009/online-textbook/part_ii_open_5_0.pdf