

# **A Fault Tolerant Hierarchical FUSE File System**

Pratik Karambelkar and Alden Lobo  
Department of Electrical and Computer Engineering  
University of Florida, Gainesville, Florida, USA

## **I. Background:**

With advances in processing power, modern computer systems are capable of handling large data sets and are also capable enough of performing advance computations on these data sets. However such systems are generally complex and expensive. This drawback has led to an industry shift to implementing a Client/Server architecture [1]. In such an implementation, a client or a particular program on the client machine requests for a particular service, data or operation that the server provides. On receiving a request from the client, the server performs the required task and returns the results back to the client. A main advantage of such a model is that it establishes a level of abstraction on the client side wherein all the complex operations are done on the server side and the client is only concerned with providing the necessary input conditions and data (if necessary) and with the output obtained from the server.

However, due to the inevitable failure of storage systems, the possibility of data loss is of grave concern. To tackle this roadblock, one of the commonly used method is to utilize a multiple server implementation to create several levels of fail-safes. This added redundancy is useful to avoid data loss due to scenarios like data corruption, server failures or crashes or any other unseen cases. Two general techniques to utilize replication for fault tolerance are [2]:

- Primary backup replication.
- Active Replication.

### **Primary Backup Replication:**

This technique utilizes one primary server (called replica) and several other secondary servers. The role of the primary server is accept requests from the Client and send back the necessary responses. The secondary servers act just as a backup of the primary server. They have whatsoever no interaction with the Client.

### **Active Replication:**

In this implementation, N number of servers are utilized (all having the equal status) to serve requests from the Client. But as several servers now directly interact, a mechanism has to be established on the server side so as to ensure only the correct data is passed to the Client.

In our implementation of making our current FUSE File System fault tolerant, we have chosen the Active Replication technique. There are several drawbacks on has to keep in mind while using a single Server/Client model. To name a few – there is no system in place for restoring lost data, too much dependency on a single server, higher probability of no response and total system breakdown on an event of a Server crash. In the era where the success of a request from a server is crucial with respect to a particular job, we have to develop a highly reliable system which can provide accurate and consistent data. We utilize the Active Replication technique built around the quorum method [3]. The implementation of this method shall be discussed in detail in the next section. This architecture of having several Servers to

respond to requests from a client may seem redundant, but is highly effective as it spreads the data across several replicas which decreases the dependence on a single server.

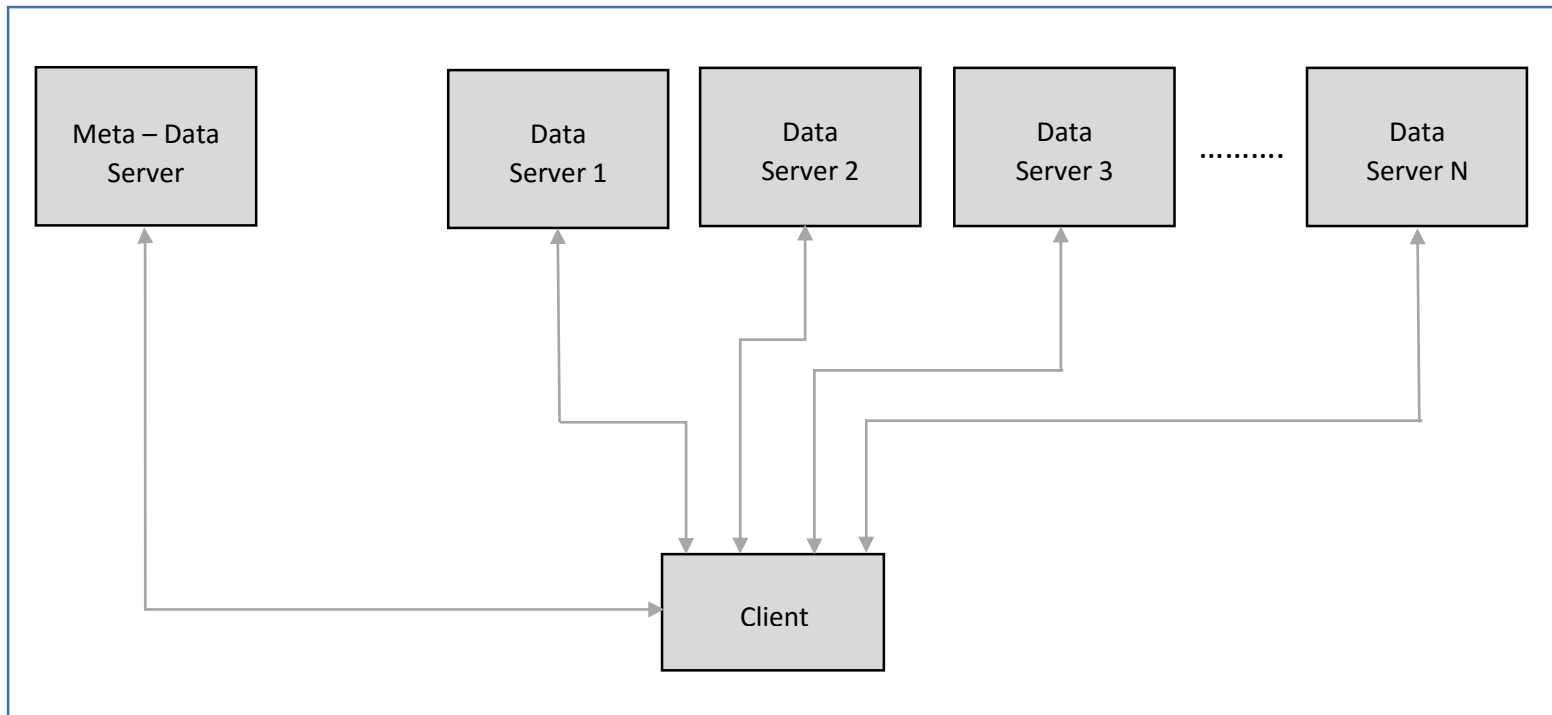


Fig. 1: Client – Server Architecture

## II. Implementation:

### Current Implementation:

We have been using the Filesystem in Userspace (FUSE) system which enables users to create their own virtual space. This kind of virtual file system gives users Root privileges and enables them to implement their own file system. We have developed a hierarchical file system and utilized the XML-RPC protocol to establish a Client – Server relationship and migrate all the data and meta-data onto the created server.

### Extended Implementation:

In order to extend the Client – Server FUSE File System into a fault tolerant system, we have utilized the quorum method to dictate the minimum number of servers required for reads and writes. This helps give us a standard to which we can conform in order to be assured of a successful read or write. Going ahead we state several mechanisms to detect corrupted data and rectify this erroneous data by copying that specific data from the servers that are healthy. The Majority Voter helps us recognize valid data.

Firstly, we shall talk about the implementation of the quorum method. While running the fault tolerant version of the File System program we pass multiple arguments, two of which being the ones that specify the read and write quorums. Qr and Qw define the read and write quorums respectively.

As per the quorum property,

$$Q_r + Q_w > N_{replicas}$$

where N is the number of servers. According to theory, a write will be said to be successful only on writing to a minimum of  $Q_w$  quorums (although the system will continue to write to the remaining servers as well). Similarly, a read will be said to be successful if at least a minimum of  $Q_r$  servers agree on a particular read content.

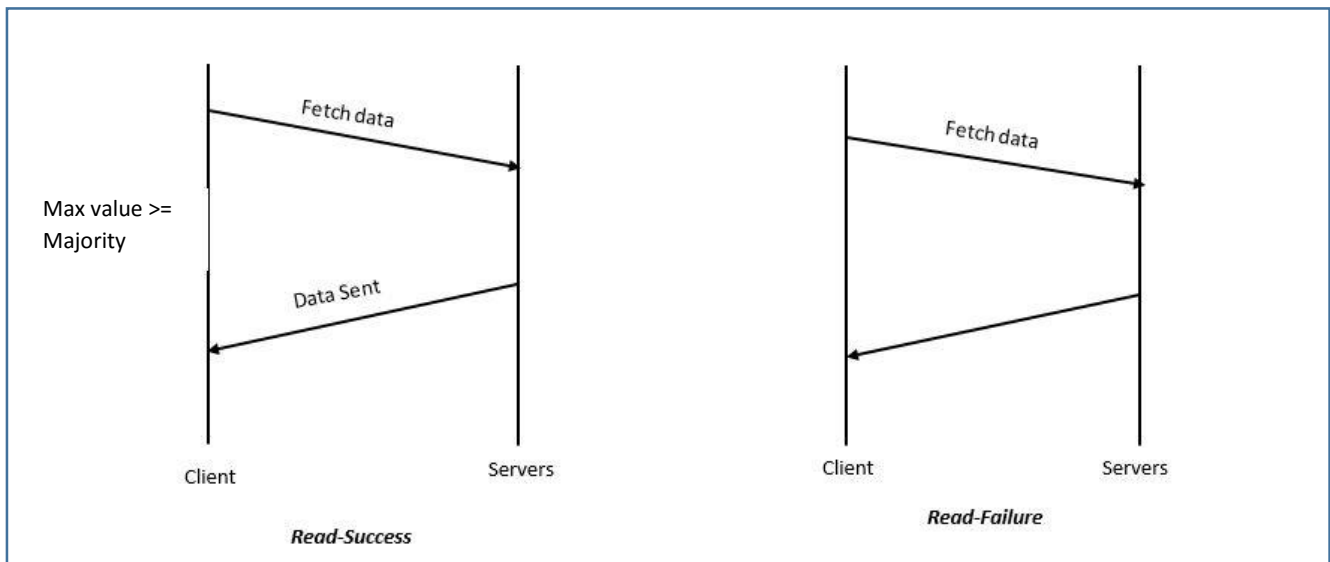


Fig. 2: Read Timing Diagrams

The Read - Success and Read - Failure (as defined by the Quorum property) timing diagrams are shown above. The client checks if the number of servers giving a response to a particular request is greater than the required majority. If its greater, then it accepts the response, else it discards it.

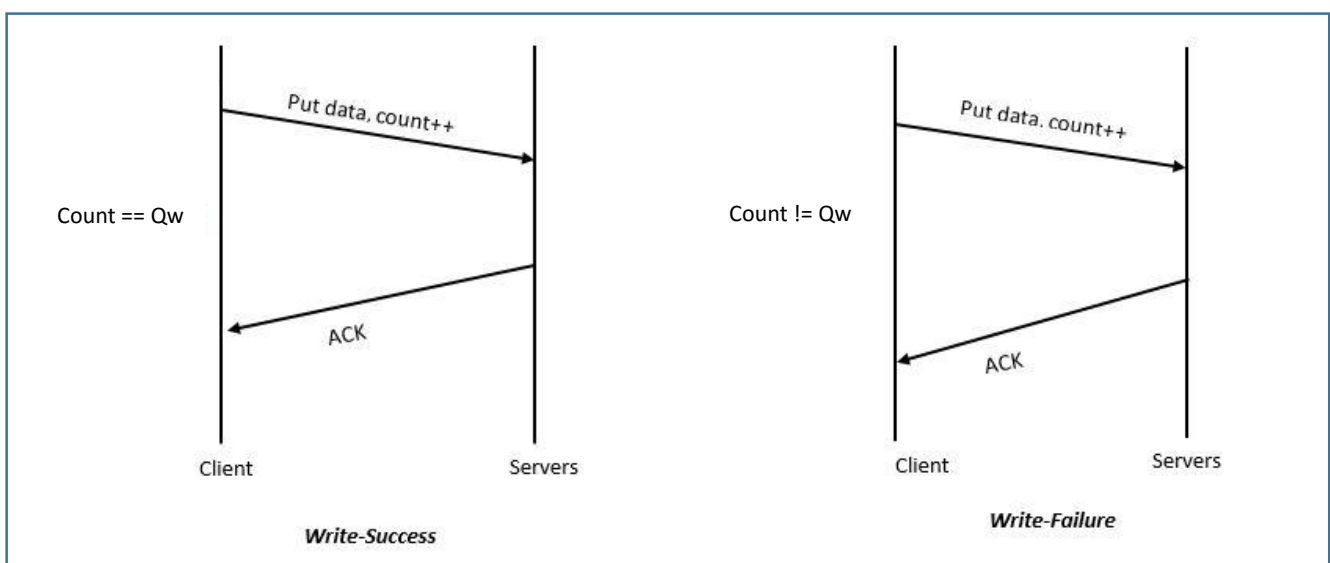


Fig. 3: Read Timing Diagrams

The Write - Success and Write – Failure (as defined by the Quorum property) timing diagrams are shown above. The client checks if the number of successful write to a server is at least equal to that specified by the Quorum property.

For example, if the number of servers is equal to 5, that is,  $N \text{ replicas} = 5$ . Generally the  $Q_w = N \text{ replicas}$ . Let us assume that  $Q_r$  is equal to 3 and hence at least three servers have to agree on the data which will be sent to the client. For a write to be successful, the minimum  $Q_w$  servers have to get data written in them. Hence, as shown in the timing diagram the write success only happens when the data is written in all the servers. In the same manner, a write - fail occurs because enough number of servers were not written in. A read - fail occurs when minimum  $Q_r$  number of servers do not agree upon the output, while a read - success happens only when we have a minimum  $Q_r$  number of servers agreeing on the data.

## **Modules:**

We have implemented the fault tolerant File System by segregating the system into 3 parts. This introduces modularity to the system which helps to prevent most or all errors from propagating beyond the scope of that particular module. It also makes managing each module straightforward. The three modules that we have implemented are:

- 1. Client:**

This module is concerned with the main underlying functionality of the FUSE File System. It is responsible for sending requests for meta - data and the data to the Meta Server and the Data servers respectively. The Client is oblivious to the functionality of the Servers. Each time an operation is performed on (something is written into or read from) the file system, the client uses the respective server.

- 2. Meta Data Server:**

This module mainly deals with the Meta data of the respective objects in the File System. There is only a single Meta Data Server available.

- 3. Data Server:**

The Data Server stores all the data of the respective objects of the File System. There can be  $N$  number of servers.

## **Client Module:**

The following are the additions and modifications made to the previous hierarchical FUSE implementation:

### **1. Putting and getting the Meta - Data and the Data from the servers:**

Meta - data and the data are integral while doing any basic functions with files. As per the problem statement, we were required to build one meta - data server and multiple data servers. Hence no failures or corruption of the data is considered in the meta - data server. In other words, it is considered to be ideal. Instances of the meta - data server and data servers are defined while mounting the file system. The `put_meta_server()` and `get_meta_server()` functions are defined to put and get meta data into a sever, respectively. There are methods within the `simpleht` function can directly be used, but the demerit of this method would be usage of complex command which would finally lead to syntax errors. Hence to reduce coding errors we defined these two functions to put and get the metadata to the server.

```

Majority = 0
If Qr % 2 == 0:
    majority = (Qr/2)+1
else:
    majority = (Qr+1)/2

```

Fig. 4: Majority checking algorithm.

- **get\_data\_server()**

This function is defined to get data from the servers, which return multiple responses, on which voting is done in order to choose and pass a valid response back to the client and an error if there is no majority. Majority voting is done taking Qr into account. Qr and Qw are used from the arguments passed by the user and their description is given below.

```

get_data_server():
initialize empty list d = [ ]
majority = value(even or odd)
while(# of data blobs) < Qr:
    append the list 'd' with data blobs
    except 'server down'
    increment counter i
if (counter i == Qw) then:
    all the servers are down
    Break ;
unique = make a unique set of data blobs got
unique_count = counting the occurrences of the unique data blobs in
the original 'd' list and incrementing the corresponding count values
in this list
max_value= max of unique_count
if max_value > majority then pass the data client:
else return 'False'

```

Fig. 5: Getting data from the server algorithm.

- **put\_data\_server()**

This function is used to write the data into the servers and the count variable is used in the function to keep a count of successful writes done into the servers. At the end, the count is compared to Qw to send the corresponding acknowledgment to the user. All four functions take path as an argument, which is then passed to the servers; where the path in binary format is stored as the key and the data or metadata is stored as the corresponding value to that path.

The former two functions are the foundations of the code which add fault tolerance to our basic client and server model. The concept is called the Quorum approach which is an extension to the modular redundancy technique which uses replication to add some fault tolerance to a naturally Non-Fault tolerant system.

```

put_data_server():
Initialize a count
for all servers:
    Put the data blob in servers one by one
    Increment count
if count == Qw then: return 'True'
else return 'False'

```

Fig. 6: Putting data into the server algorithm.

## **2. Qr & Qw variables:**

These variables are used in the code to implement the Quorum functionality in a single client and multi-server model. The Client generates a request which is passed to an array of servers and the responses of all these servers are collaborated and one final majority response is given back to the client and this is where the Qr and Qw comes into the picture. Qr is used in the `get_data_sever()` function where data is read from the servers, that is, responses of all the servers are compared and Qr specifies the minimum number of servers that have to agree on the responses, to send back a valid response to the client. Similarly, Qw is used in the `put_data_server()` function where data is written to the servers and we count the number of writes to the servers, transfer it to the count variable and check it if is at least equal to the Qw or not. If not send back 'False' and if it matches then send a 'True' acknowledgement.

## **3. Finding the Parent path and the Child path from the given path: `path_parent()` and the `path_child()`**

`path_parent()` and the `path_child()` are two small but integral functions to the code. A path is passed as an argument to the `path_parent()` function and the parent is computed and returned back to the calling function. The `path_child()` function takes the path as an argument and returns just the child of that path to the calling function.

## **4. Deleting the metadata and the data in the servers: `del_all()`**

The `del_all()` function takes care of deleting the meta - data and the data of the file corresponding to the path which is passed as an argument to that function. To enable this functionality of deleting the contents and the meta - data of the file, we have to add a function at the server side as the `simpleht.py` provides us the putting and getting the data but not deleting. Hence, we have created a function called `rmv()` which can be called from the client side to get rid of the contents and the metadata of the file, that is, it essentially takes a path as the key and deletes the value corresponding to it from the hash table of the servers.

## **6. Creating files: `create()`**

Creating files will require putting the data into the data servers and putting the corresponding meta - data of the newly created file into the metadata server. Along with putting the correct data types to servers, `path_parent()` function is called which returns the path of the parent to the `create()` function. After getting the parent path, the metadata of the parent is fetched from the servers. The `path_child()` is called to get the name of the child and finally the 'list\_nodes' field of the metadata of the parent is appended by the name of the child. The `list_nodes` field of the parent contains the list of names of all the children under it. The updated meta - data is overwritten in the meta - data server for this file.

## **7. Making a directory: `mkdir()`**

Creating a directory in the file system involves similar steps as used in creating a file. Firstly, the meta - data of the newly created directory is placed into the meta - data server by using the `put_meta_server()` function. Then the parent path is found by using the `path_parent()` function and the 'list nodes' field of the parent is appended by the newly created child directory. The updated meta - data is then overwritten in the meta - data server for the file.

## **8. Removing a directory: `rmdir()`**

To make sure that the chosen object for removal is a directory, we first make sure that the object is a directory type by checking the 'st\_mode' field of the metadata of the directory. If it is not a directory then an exception is raised. Secondly, we have to make sure that the directory is empty. This can be done by checking the `list_nodes` of the directory. If it is not empty then raise an exception. If the object passes both the tests, the meta - data of the directory

is removed from the meta - data server and the parent path is fetched using parent\_path function and the child is removed from the list nodes of the parent.

### **9. Renaming a directory or a file: rename()**

This functionality is made up of two functions, the rename() function and the rename2() function. The rename() function calls the rename2() function which can in turn call itself recursively. The function checks the meta – data to take appropriate action depending if the object is a file or a folder. We utilize the path\_parent() and the path\_child() functions along with other functions like the rmv() function in the process to transfer the meta – data and the data from the old named file or folder to the new named file or folder. A couple of drawbacks were encountered in the creation of this problem and they shall be stated in a later section.

### **10. Creating Sym-link: symlink()**

This function involves the placement of the meta - data of the source to the meta\_data\_server() and the parent path is fetched using path\_parent () function by passing the target as the argument. We fetch the metadata of the parent and then append the list nodes of the parent and update it back to the meta - data server.

### **11. Truncating the length of the file: truncate()**

The truncate function is used to truncate the length of the file to a user defined value. After the data of the file is fetched and the length of the data is limited to the value given by the user, the data is updated back to the servers. The meta - data of the file is also fetched and the 'st\_size' field of the meta - data is modified and then updated back to the meta - data server.

### **12. Unlinking the file from the parent directory: unlink()**

This function involves firstly the removal of the meta - data of the file from the meta - data server and then parent path is fetched using path\_parent() function and meta - data of the parent is fetched and the child is removed from the list nodes of the parent.

### **13. Writing to a file: Write()**

The data servers give responses for the value corresponding to the path which is passed as argument to this function. The data is written from the offset and the data is updated back to the file and is put back to the data servers. The meta - data of the file is fetched and the 'st\_size' field of the file is edited and is saved back to the metadata server.

### **14. Made the necessary change to accept more arguments to accommodate for the ports quorums specified. Added the functionality to check for cases when Qw & Qr are less than 1:**

The main() of the file system code has been edited to take care of the passed quorum formation parameters and the initialized meta port and the data port. This functionality is implemented in the main() of the file system code where the inputs from the user are taken for the Qr and Qw ,the meta port and the data ports. While processing these arguments if the Qw and Qr is less than 1 then we throw an error stating that the values given are not correct and the action should be tried once again.

## Server Module:

The SimpleXMLRPCServer module provides us with the elementary framework for XML – RPC servers that is necessary to setup a Client – Server configuration.

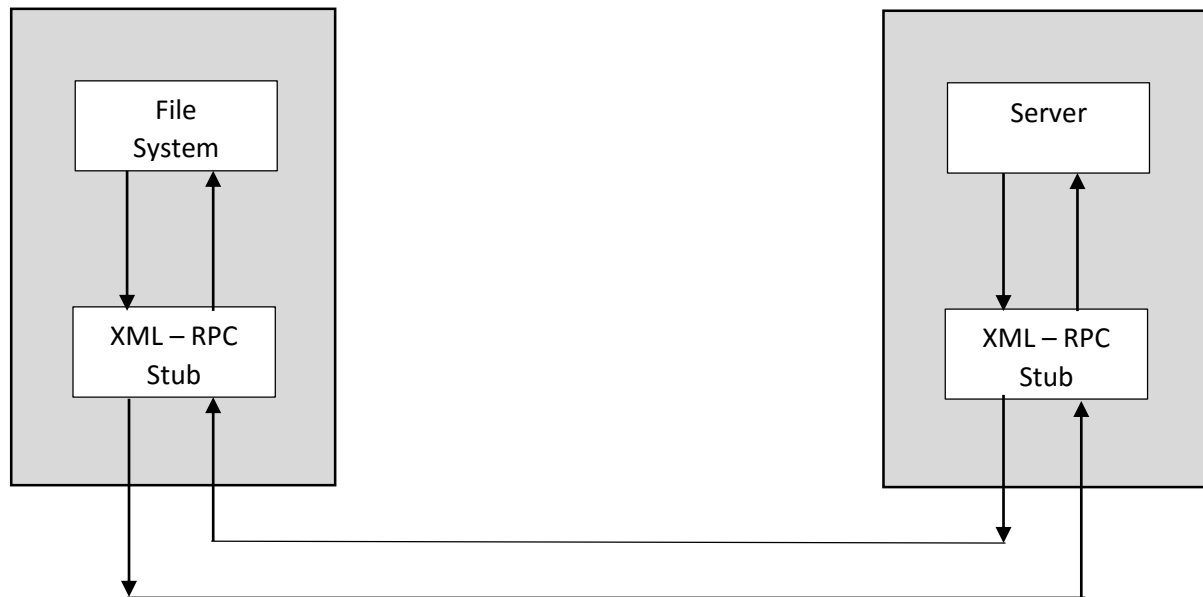


Fig. 7: XMLRPC Configuration

The following are the additions and changes made to the previous SimpleHT code:

### 1. List contents: `list_contents()`

It is used to return all the keys in a particular data server.

### 2. Corruption of data: `corrupt()`

It uses the `list_contents()` function to get the necessary keys and will use those keys to corrupt the values corresponding to that key.

### 3. Creating a new class: `Class Server(SimpleXMLRPCServer)`

We have created this class as a co - existing class to the SimpleHT class. It is useful in the terminate function.

### 4. Terminate a server remotely: `terminate()`

A class `Server` is defined in the `simpleht.py` code which enables the opportunity to terminate the server when the terminate function is called remotely from another terminal.

### 5. Restart the server: `restart()`

This function enables the opportunity to remotely connect to an already established server and restart it. All the contents of that particular server will be wiped out. A couple of drawbacks were encountered in the creation of this problem and they shall be stated in a later section.

### 6. Remove the contents :- `rmv()`

It takes the key and removes the value corresponding to that key from the hash tables.



## 7. Added the new function to the serve function: serve()

All the functions which are being used in the simpleht.py code must be registered in the serve() function and hence all the newly created functions, that are registered here, can be used on the server as well as the client side.

### Mounting Procedure:

The project setup consists of three basic components:

1. The Hierarchical FUSE File System. (tolFS.py)
2. The Meta – Data server. (simpleht.py)
3. The Data server. (simpleht.py)

The procedure to run the files in order to mount the developed file system are as per the given instructions and in the order stated:

1. The Meta-Data server and the Data server are mounted using the following commands:
  - a. For running a Meta-Data server we use the following commands:

*python simpleht.py <Meta-Data port>*

- b. For running a Data server we use the following commands:

*python simpleht.py <Data port> <Remaining Data ports>*

2. The Hierarchical FUSE file system is mounted using the following commands:

*python tolFS.py <mount-point> <Qr> <Qw> <Meta-Data port> <Data ports>*

Here <mount-point> is the folder where the FUSE file system will be mounted.

Once this is completed, the file system will have been successfully mounted at the specified mount point.

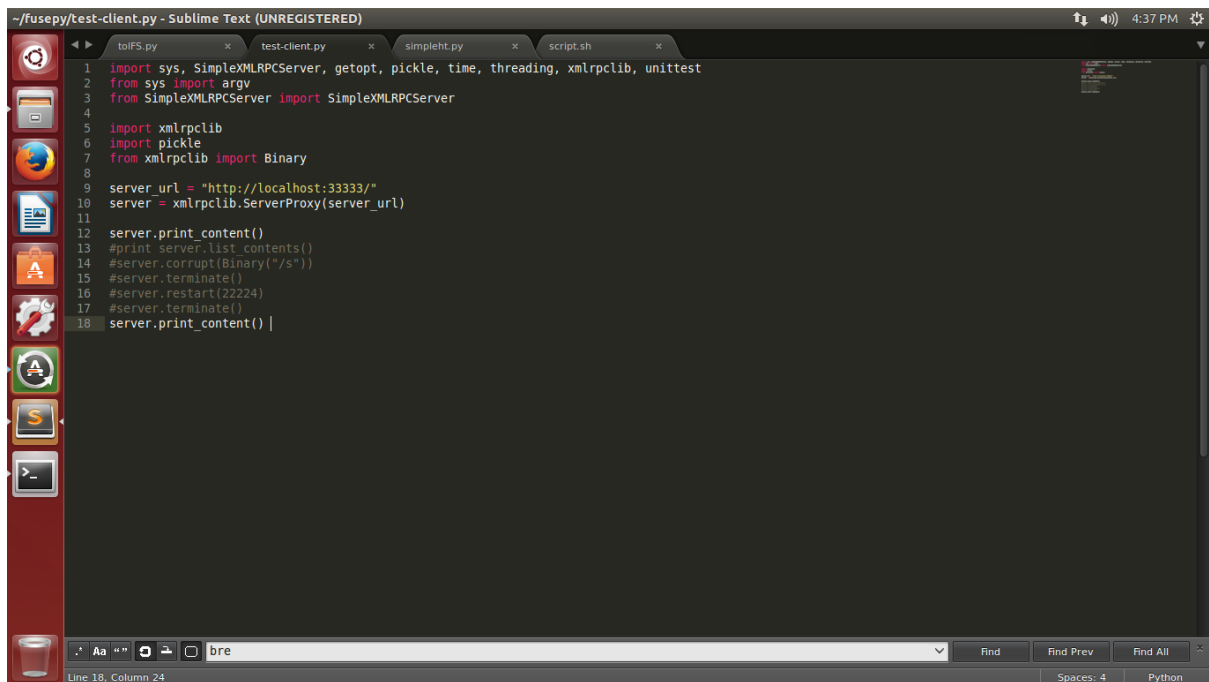
## III. Testing Mechanisms:

We have tested out implementation by utilizing an automated test script called *test-client.py* in order to evaluate the execution of various scenarios.

The test script can be modified to accomplish the following functions:

1. Remotely kills servers and checks if the writing of data will be blocked.
2. Corrupts the data in a particular server remotely and checks if the data is getting corrected on the next read.
3. Remotely restarted a server.
4. Remotely terminate a server.

The test script that we use to test the cases mentioned above is done by modifying the code shown below:

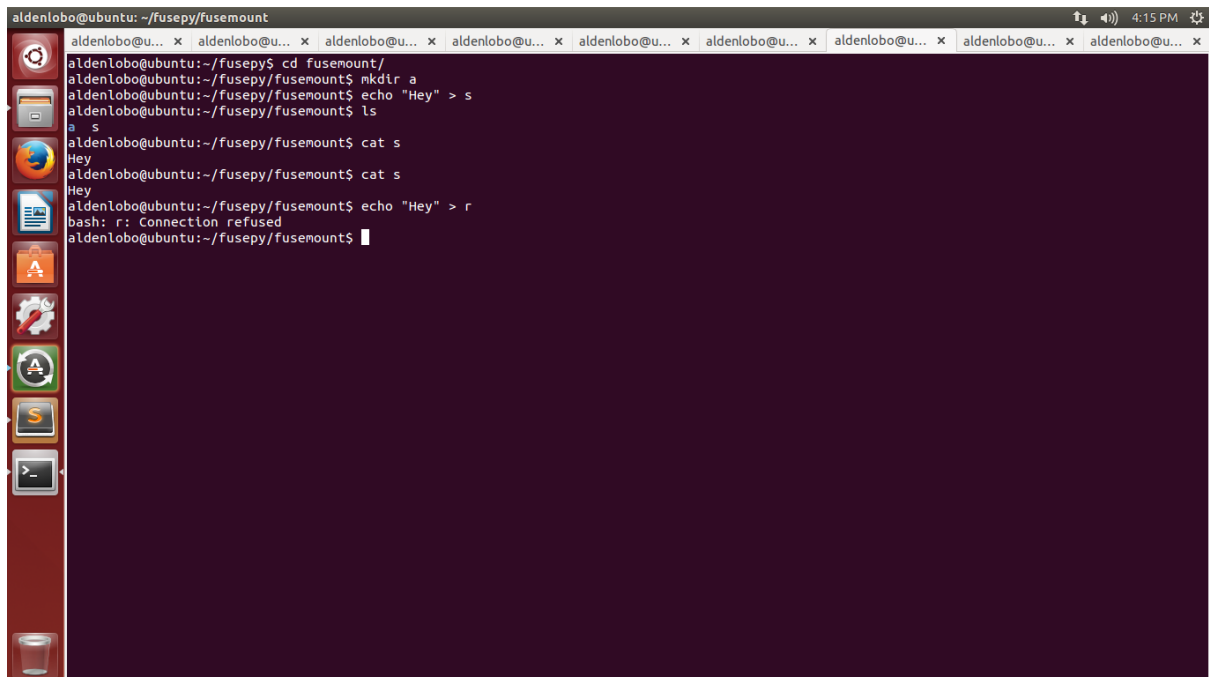


```
1 import sys, SimpleXMLRPCServer, getopt, pickle, time, threading, xmlrpclib, unittest
2 from sys import argv
3 from SimpleXMLRPCServer import SimpleXMLRPCServer
4
5 import xmlrpclib
6 import pickle
7 from xmlrpclib import Binary
8
9 server_url = "http://localhost:33333/"
10 server = xmlrpclib.ServerProxy(server_url)
11
12 server.print_content()
13 #print server.list_contents()
14 #server.corrupt(Binary("/s"))
15 #server.terminate()
16 #server.restart(22224)
17 #server.terminate()
18 server.print_content() |
```

Fig. 8: Test script to observe the results of various cases.

The following are screenshots of the test outputs:

1. Remotely killed a server and then tried to write a file to the file system.



```
aldenlobo@ubuntu: ~/fusepy/fusemount
aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x
aldenlobo@ubuntu:~/fusepy$ cd fusemount/
aldenlobo@ubuntu:~/fusepy/fusemount$ mkdir a
aldenlobo@ubuntu:~/fusepy/fusemount$ echo "Hey" > s
aldenlobo@ubuntu:~/fusepy/fusemount$ ls
a s
aldenlobo@ubuntu:~/fusepy/fusemount$ cat s
Hey
aldenlobo@ubuntu:~/fusepy/fusemount$ cat s
Hey
aldenlobo@ubuntu:~/fusepy/fusemount$ echo "Hey" > r
bash: r: Connection refused
aldenlobo@ubuntu:~/fusepy/fusemount$
```

Fig. 9: Fusemount directory error.

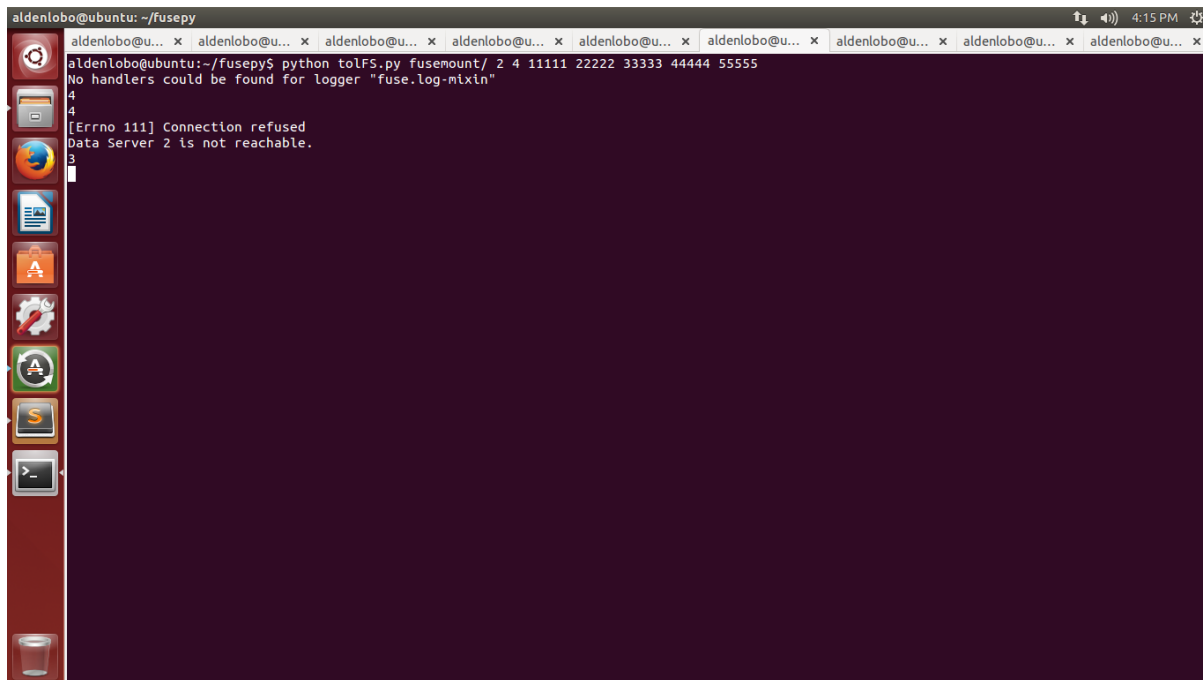


Fig. 10: Error shown on the server terminal.

As can be seen in the screenshots above, on killing a server, new data cannot be written to any of the servers if they do not satisfy the Qw value set.

2. Here we corrupt data in a file already written to the file system. The file system (on the server side) then automatically corrects this data by comparing it with the data in the data in other servers and then over writing the obtained correct data into the corrupted file in the respective server.

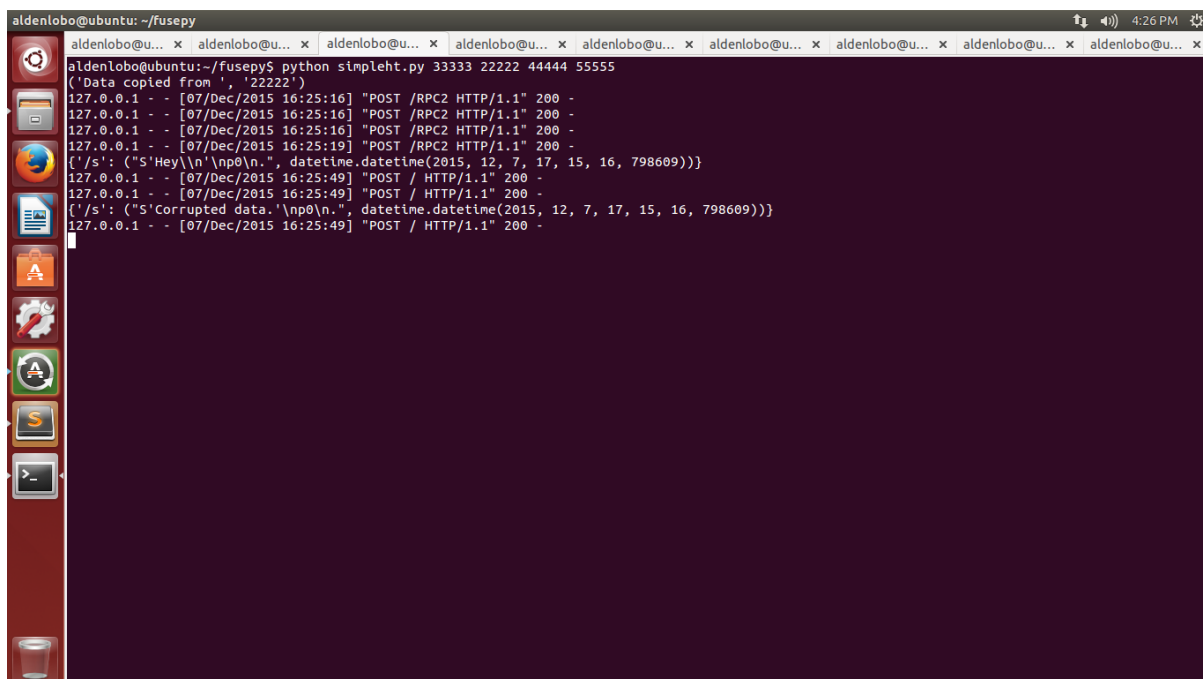
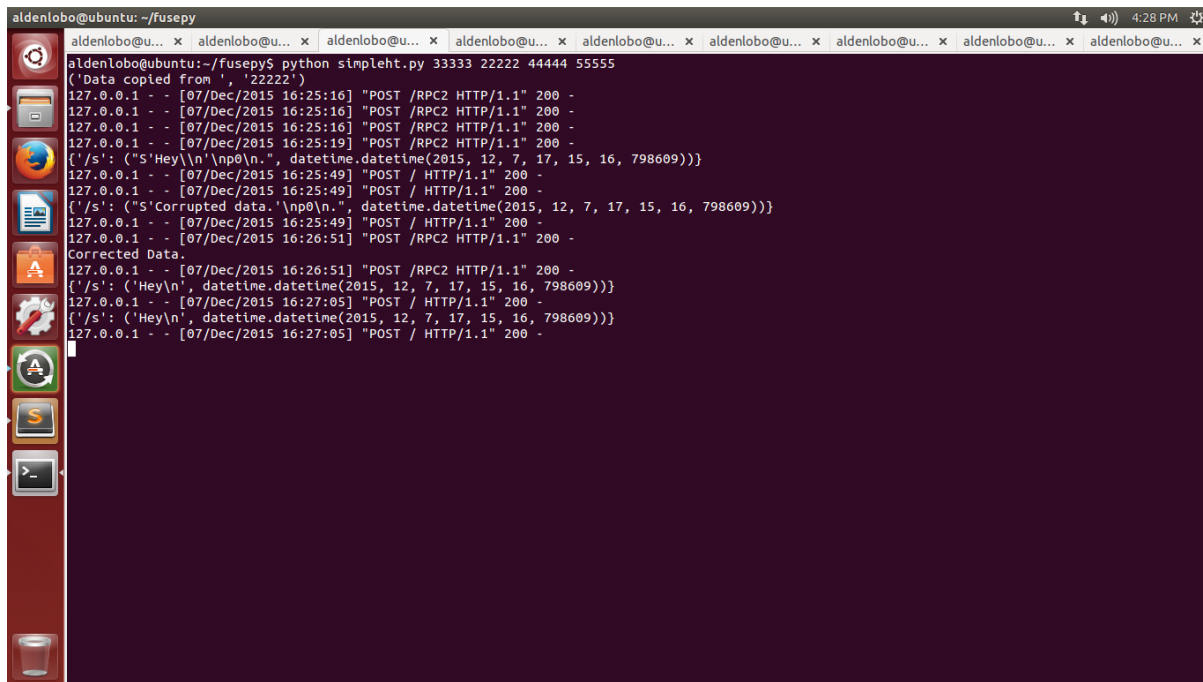


Fig. 11: Server terminal before data corruption.

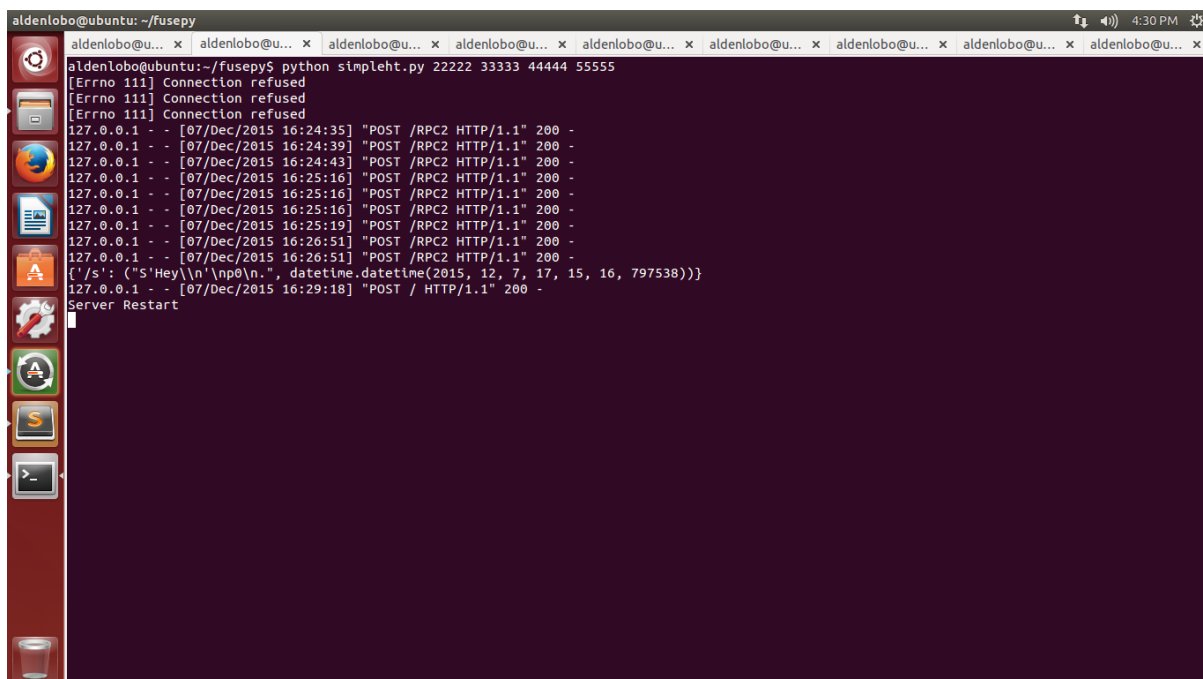


```
aldenlobo@ubuntu: ~/fusepy
aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x
aldenlobo@ubuntu:~/fusepy$ python simpleht.py 33333 22222 44444 55555
('Data copied from ', '22222')
127.0.0.1 - - [07/Dec/2015 16:25:16] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:16] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:16] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:19] "POST /RPC2 HTTP/1.1" 200 -
{'/s': ('S'Hey\\n'\\np0\\n.", datetime.datetime(2015, 12, 7, 17, 15, 16, 798609))}
127.0.0.1 - - [07/Dec/2015 16:25:49] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:49] "POST / HTTP/1.1" 200 -
{'/s': ('S'Corrupted data.'\\np0\\n.", datetime.datetime(2015, 12, 7, 17, 15, 16, 798609))}
127.0.0.1 - - [07/Dec/2015 16:25:49] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:26:51] "POST /RPC2 HTTP/1.1" 200 -
Corrected Data.
127.0.0.1 - - [07/Dec/2015 16:26:51] "POST /RPC2 HTTP/1.1" 200 -
{'/s': ('Hey\\n', datetime.datetime(2015, 12, 7, 17, 15, 16, 798609))}
127.0.0.1 - - [07/Dec/2015 16:27:05] "POST / HTTP/1.1" 200 -
{'/s': ('Hey\\n', datetime.datetime(2015, 12, 7, 17, 15, 16, 798609))}
127.0.0.1 - - [07/Dec/2015 16:27:05] "POST / HTTP/1.1" 200 -
```

Fig. 12: Server terminal after data correction.

As we observe in the screenshots, the first image shows the corrupted data. The second image shows that the corrupted data has now been replaced with the appropriate correct data.

### 3. Remotely restarted a server.



```
aldenlobo@ubuntu: ~/fusepy
aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x
aldenlobo@ubuntu:~/fusepy$ python simpleht.py 22222 33333 44444 55555
[Errno 111] Connection refused
[Errno 111] Connection refused
[Errno 111] Connection refused
127.0.0.1 - - [07/Dec/2015 16:24:35] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:24:39] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:24:43] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:16] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:16] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:16] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:25:19] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:26:51] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:26:51] "POST /RPC2 HTTP/1.1" 200 -
{'/s': ('S'Hey\\n'\\np0\\n.", datetime.datetime(2015, 12, 7, 17, 15, 16, 797538))}
127.0.0.1 - - [07/Dec/2015 16:29:18] "POST / HTTP/1.1" 200 -
Server Restart
```

Fig. 13: Server terminal once it is restarted.

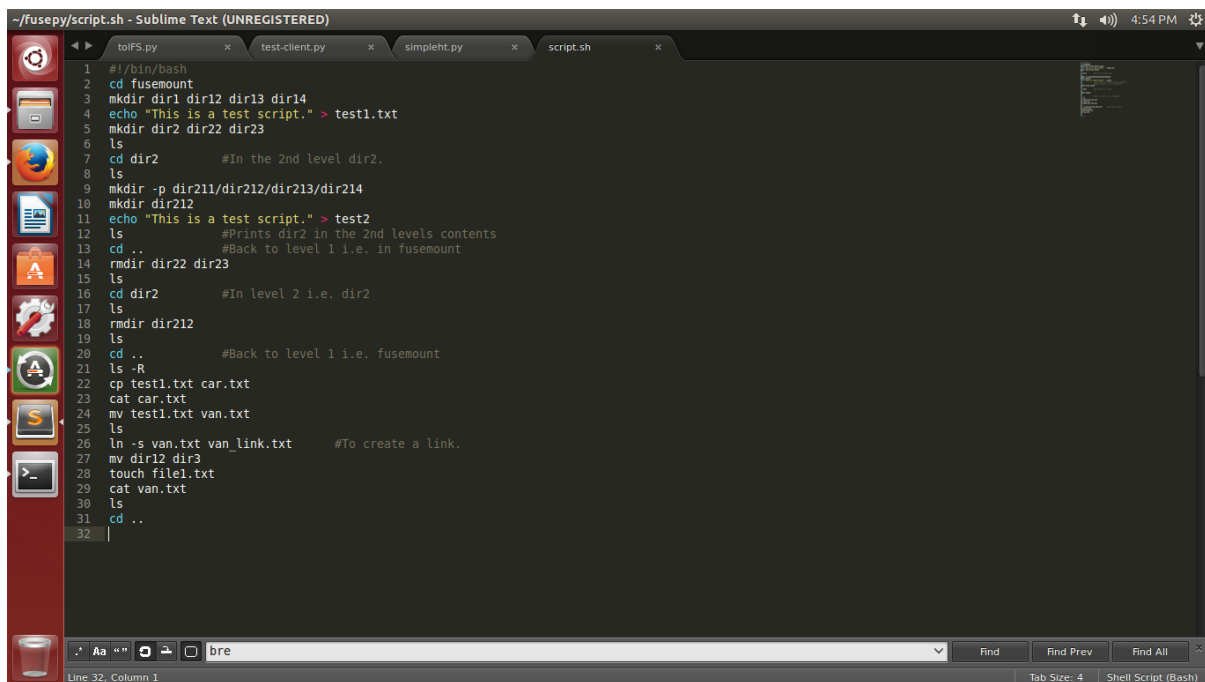


## IV. Evaluation Mechanisms:

A system which consists of replicated servers will have a slightly greater latency than a system with a single server. A file system which works on a local system has a latency that is lesser than even that of a single server.

In order to demonstrate this, we run the following script using the command,

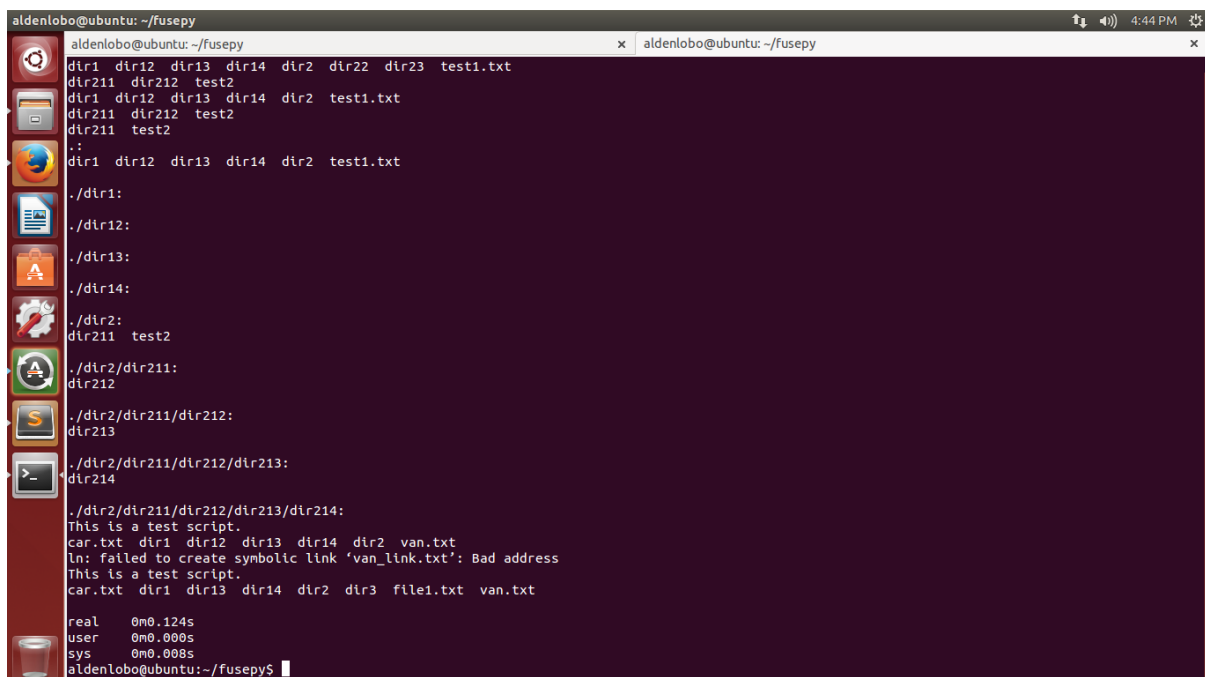
`time ./script.sh`



```
1 #!/bin/bash
2 cd fusemount
3 mkdir dir1 dir2 dir3 dir4
4 echo "This is a test script." > test1.txt
5 mkdir dir2 dir23
6 ls
7 cd dir2      #In the 2nd level dir2.
8 ls
9 mkdir -p dir211/dir212/dir213/dir214
10 mkdir dir212
11 echo "This is a test script." > test2
12 ls          #Prints dir2 in the 2nd levels contents
13 cd ..      #Back to level 1 i.e. in fusemount
14 rmdir dir22 dir23
15 ls
16 cd dir2    #In level 2 i.e. dir2
17 ls
18 rmdir dir212
19 ls
20 cd ..      #Back to level 1 i.e. fusemount
21 ls -R
22 cp test1.txt car.txt
23 cat car.txt
24 mv test1.txt van.txt
25 ls
26 ln -s van.txt van_link.txt    #To create a link.
27 mv dir12 dir3
28 touch file1.txt
29 cat van.txt
30 ls
31 cd ..
32 |
```

Fig. 16: Test Script

The following are the completion times of the various systems:



```
aldenlobo@ubuntu: ~/fusepy
dir1 dir12 dir13 dir14 dir2 dir22 dir23 test1.txt
dir211 dir212 test2
dir1 dir12 dir13 dir14 dir2 test1.txt
dir211 dir212 test2
dir211 test2
.:
dir1 dir12 dir13 dir14 dir2 test1.txt
./dir1:
./dir12:
./dir13:
./dir14:
./dir2:
dir211 test2
./dir2/dir211:
dir212
./dir2/dir211/dir212:
dir213
./dir2/dir211/dir212/dir213:
dir214
./dir2/dir211/dir212/dir213/dir214:
This is a test script.
car.txt dir1 dir12 dir13 dir14 dir2 van.txt
ln: failed to create symbolic link 'van_link.txt': Bad address
This is a test script.
car.txt dir1 dir13 dir14 dir2 dir3 file1.txt van.txt

real    0m0.124s
user    0m0.000s
sys     0m0.008s
aldenlobo@ubuntu:~/fusepy$
```

Fig. 17: Completion time of a simple hierarchical file system.

```

aldenlobo@ubuntu: ~/fusepy
aldenlobo@ubuntu:~/fusepy x aldenlobo@ubuntu:~/fusepy x aldenlobo@ubuntu:~/fusepy
dir1 dir12 dir13 dir14 dir2 dir22 dir23 test1.txt
dir211 dir212 test2
dir1 dir12 dir13 dir14 dir2 test1.txt
dir211 dir212 test2
dir211 test2
.:
dir1 dir12 dir13 dir14 dir2 test1.txt
./dir1:
./dir12:
./dir13:
./dir14:
./dir2:
dir211 test2
./dir2/dir211:
dir212
./dir2/dir211/dir212:
dir213
./dir2/dir211/dir212/dir213:
dir214
./dir2/dir211/dir212/dir213/dir214:
This is a test script.
car.txt dir1 dir12 dir13 dir14 dir2 van.txt
ln: failed to create symbolic link 'van_link.txt': Bad address
This is a test script.
car.txt dir1 dir13 dir14 dir2 dir3 file1.txt van.txt
real    0m0.318s
user    0m0.004s
sys     0m0.004s
aldenlobo@ubuntu:~/fusepy$

```

Fig. 18: Completion time of a Client – Server hierarchical file system.

```

aldenlobo@ubuntu: ~/fusepy
aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u... x aldenlobo@u...
aldenlobo@ubuntu:~/fusepy$ time ./script.sh
dir1 dir12 dir13 dir14 dir2 dir22 dir23 test1.txt
dir211 dir212 test2
dir1 dir12 dir13 dir14 dir2 test1.txt
dir211 dir212 test2
dir211 test2
.:
dir1 dir12 dir13 dir14 dir2 test1.txt
./dir1:
./dir12:
./dir13:
./dir14:
./dir2:
dir211 test2
./dir2/dir211:
dir212
./dir2/dir211/dir212:
dir213
./dir2/dir211/dir212/dir213:
dir214
./dir2/dir211/dir212/dir213/dir214:
This is a test script.
car.txt dir1 dir12 dir13 dir14 dir2 van.txt
This is a test script.
car.txt dir1 dir13 dir14 dir2 dir3 file1.txt van_link.txt van.txt
real    0m0.725s
user    0m0.004s
sys     0m0.004s
aldenlobo@ubuntu:~/fusepy$

```

Fig. 19: Completion time of a fault tolerant Client – Server hierarchical file system.

Thus we can see the increase in completion time of the script with the increase of a server and then multiple servers.

Simple hierarchical file system: 0.124 seconds

Client – Server hierarchical file system: 0.318 seconds

Fault tolerant Client – Server hierarchical file system: 0.725 seconds

## **V. Potential Issues:**

- The rename() function used in the code works only while within one level. This is an issue as rename is one of the basics functions that a file system requires.
- Network related issues may come in the practical implementation as the server connection is not checked first and the data is directly put into a server. A problem can occur when packet is lost while putting the data into the server. The output data might get corrupted or might face burst errors while traveling through the network.

## **VI. Conclusion:**

A fault tolerant system eases the strain on the user from worrying about data loss to a very large extent. Although having multiple server replicas to save your data seems a good approach, the cost factor involved with setting up such a system is steep. However, in applications where data reliability, availability and security are vital, such system are essential. This project has given us a good insight to the requirements of file systems as well as the advantages and disadvantages of a fault tolerant file system.

## **VII. References:**

- [1] Adler, R. M, "Distributed Coordination Models for Client/Server Computing," Computer, Vol. 28, No.4, April, 1995, pp.14-22.
- [2] Guerraoui Rachid, Schiper André, "Software-Based Replication for Fault Tolerance", Computer, Vol. 30, Issue: 4, April 1997, pp. 68-74.
- [3] J. H. Saltzer, Principles of Computer System Design: An Introduction, 2009.