Principles of Computer System Design
Project Report

Rajesh Alwar 11110165
Aditya Kanawade 28287415

1. INTRODUCTION

In this project there is one client, one meta-data server and multiple data servers. So this is an example of implementation of RAID1 where fault tolerance is provided by replicating data in multiple servers. Our implementation uses quorum approach to specify minimum number of data servers that must be working to successfully complete read operation and also all the data servers should be working to successfully complete write operation[1].

2. BACKGROUND

This is a fault tolerant technique in which redundancy is obtained by storing the same data on multiple servers. The redundant servers provide fault tolerance by providing backup if any server fails. Data is lost only if all the servers fail. The downside of replication is that the capacity of the storage system is reduced by the factor of number of replicated servers. So, this is a very simple approach to implement at the risk of consuming capacity for replicated servers[2].

There can be two types of failures, one due to server crash and restart and other due to data corruption on single server. This failures need to be addressed because they occur in all our day to day applications which we use. In financial applications when we want to keep an account of money in bank account or transfer money between accounts, we need to take care that none of the data stored in server is corrupted and also there must be consistency with data stored on various data servers providing the service. In online shopping applications, during thanksgiving and black friday there is lot of traffic on servers, so even if one server crashes due to excessive load care must be taken to revive it and also update its contents. In file systems, all the files needs to be protected from getting corrupt. In Hadoop Distributed File System, each file is made up of several equal sized blocks and to implement fault tolerance each block is replicated on multiple data nodes for high availability[3]. All these factors leads to realization and appreciation of this project.

3. IMPLEMENTATION

Project can be divided into three modules:

1. Client
2. Mediator
3. Data servers and Meta servers

The block diagram of our project implementation is

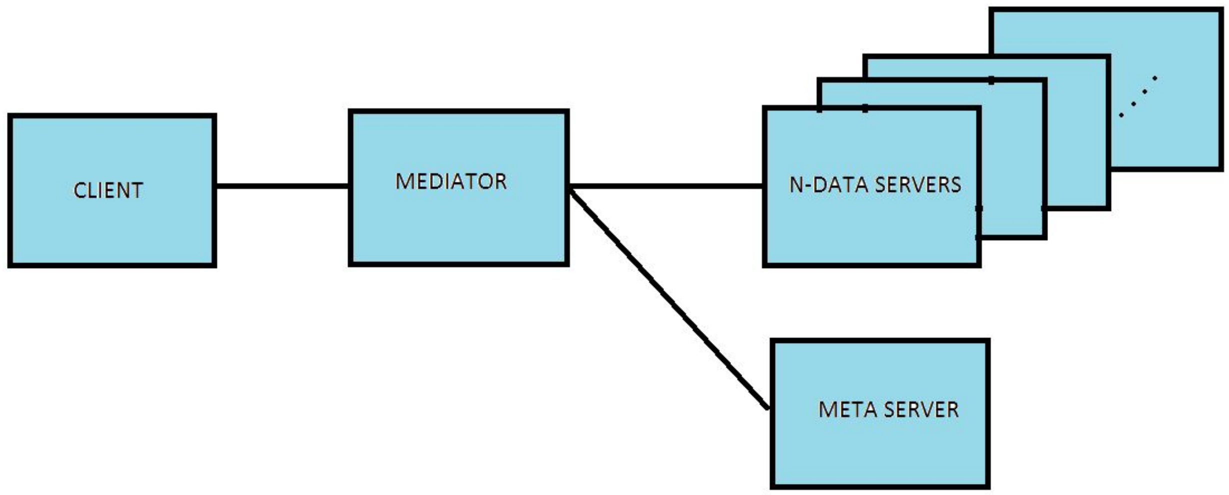


Figure 1. Block diagram of our implementation.

Client:

Firstly, we are developing our project on Fuse Filesystem. So whenever we run FileSystem.py python script in client the mediator starts working and all the arguments passed to client are transferred to it. Client just mounts the mediator in filesystem. So all the other things including fault tolerant techniques for data servers are implemented by mediator and it is transparent to client.

Mediator:

In order to invoke mediator, we will have to execute the following command:

```
python mediator.py <Qr> <Qw> <meta_port> <data_port1>....<data_portN>
```

Here, Qr represents the number of servers required to read from dataservers.
Qw represents the total number of dataservers required.
meta_port represents the port address of meta server.
data_port represents the port addresses of N data servers.

Read
So for read operation to execute successfully, there must be at least Qr servers working.
So we will first check as to how many servers are working and which ones are working. Then we will compare this value with Qr and only if the number of servers working is greater than Qr we will execute read operation else our program will wait for atleast Qr servers to work to complete read operation.

If any server recovers from a crash then it will be in blank state so all its contents need to be restored along with all the contents that changed in other replicated servers when it was down. So all this is done during read operation.

This project implements triple modular redundancy (TMR) fault tolerant approach for detecting failed server and to retrieve data from other servers. In TMR technique checks three outputs and check their correctness if at least two outputs are matching. Initially it finds out all the working servers and storing their list in an array and the number of entries will be at least equal to Qr as it is the pre-condition to start read operation. Then out of these entries, first three servers are used and if the data available in all the server matches then the data to be read is correct and it is received from the first dataserver. However, if one of the server was in blank state or its data is corrupted then the data of two dataservers will match and we copy the contents from one of this server to the server which gave different output[4].

Write
For write operation to work successfully, all the dataservers which is equal to Qn should be working. Mediator checks whether all the servers are working and if any of the server is not working then it stalls the write operation. When all the servers are working then only write operation is performed.

Metaservers and Dataservers
The code for metaserver and dataserver is exactly the same. Meta and listnodes keys are stored on Metaserver while data is stored on dataserver. The mediator checks for “data” string in each key and if it is found only then value for that key is stored on data server or else all the meta and listnode keys are stored on metaserver. The data on dataserver and metaserver is stored in different format and hence while reading key from dataserver into metaserver it gave us an error. Hence, listnodes are stored in metaserver.

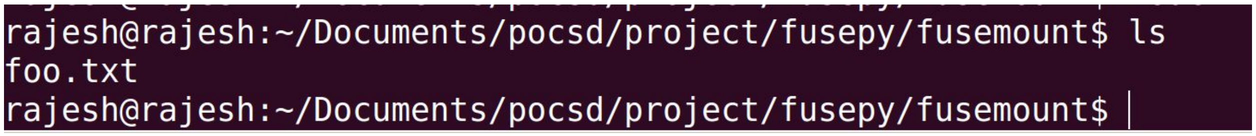
4. TESTING AND EVALUATION MECHANISMS:

We present a fault tolerant file system which handles server crashes and writes only if all servers are available. The write operation is an all or nothing operation, it either writes to all the servers or writes to none of them. To perform tests we have used listing.py, servkill.py and testbench.py files which performs listing keys, terminating server and corrupting keys respectively.

Testing for Qr = 3 and Qw = 5

Case i. When all 5 servers are available

Both Read and Write operations are successful.

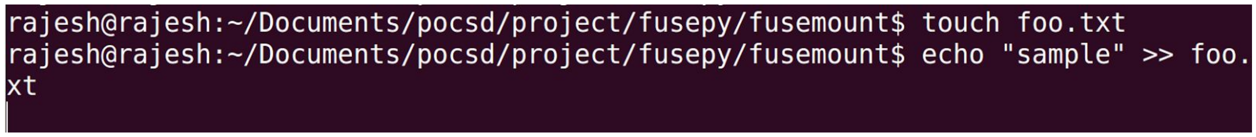


```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ ls
foo.txt
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ 
```

Figure 2. Screenshot showing successful read and write operation when all servers are working.

Case ii. When one server has crashed.

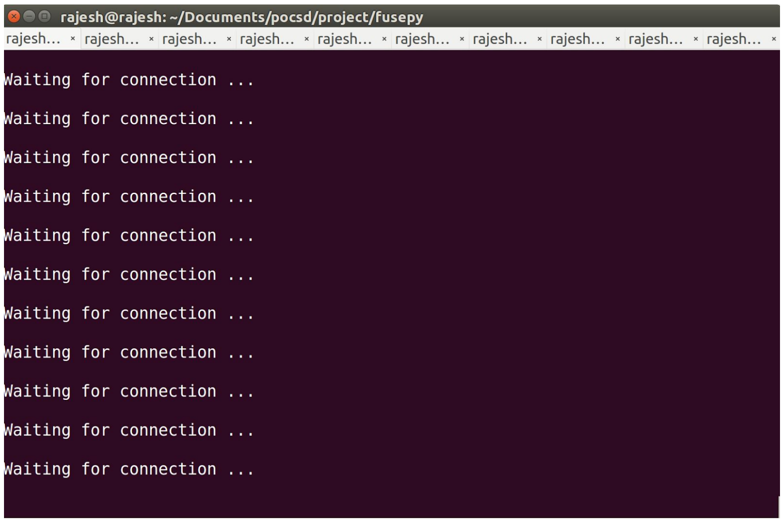
Write operation will be stalled.



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ touch foo.txt
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ echo "sample" >> foo.
xt
```

Figure 3. Screenshot showing stalled write operation when one server is not working.

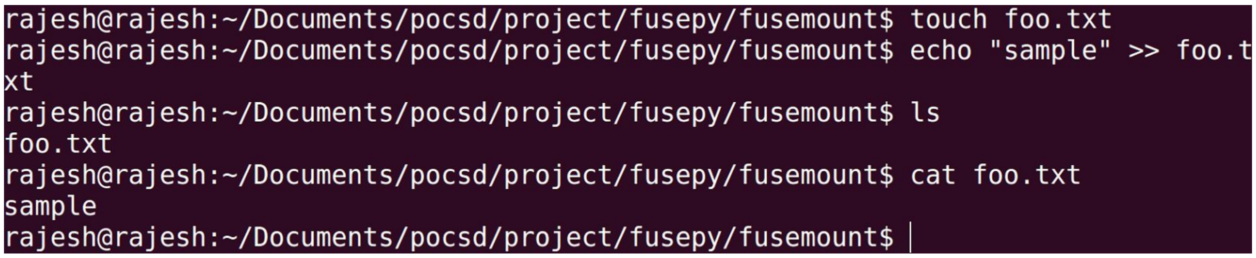
Mediator prints...



```
rajesh@rajesh: ~/Documents/pocsd/project/fusepy
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
Waiting for connection ...
```

Figure 4. Screenshot showing contents displayed in Mediator console.

The write operations finishes as soon as the server becomes available.



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ touch foo.txt
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ echo "sample" >> foo.t
xt
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ ls
foo.txt
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.txt
sample
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$
```

Figure 5. Screenshot showing completed write operation when crashed server is restarted.

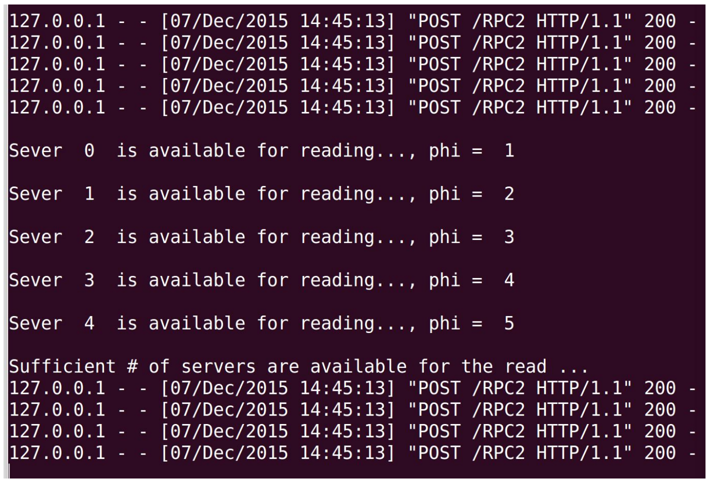
When a server becomes unavailable, and the mediator waits for the server to become available. In order to prevent excessive number of requests from server to mediator the mediator checks if the server has become available after every 1 second.

The following code takes care that the server becomes available:

```
while connected != True:
    try:
      connected = s[i].tryconnect()  #checks if i is connected or not
    except:
      print "Waiting for connection ... \n"
      time.sleep(1)
```

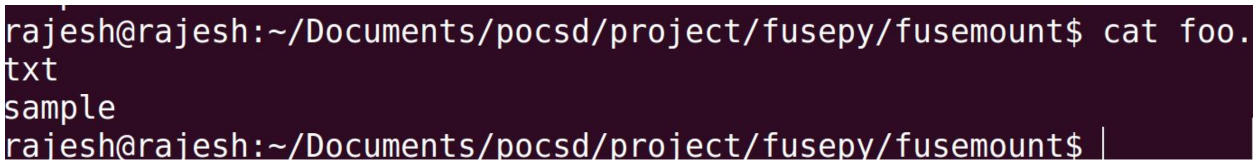
Case iii. Testing for reads: Read operation finishes when at least Qr servers are available.

When all servers are working i.e. Number of servers available(5) > Qr(3)



```
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -

Sever  0  is available for reading..., phi =  1

Sever  1  is available for reading..., phi =  2

Sever  2  is available for reading..., phi =  3

Sever  3  is available for reading..., phi =  4

Sever  4  is available for reading..., phi =  5

Sufficient # of servers are available for the read ...
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:45:13] "POST /RPC2 HTTP/1.1" 200 -
```

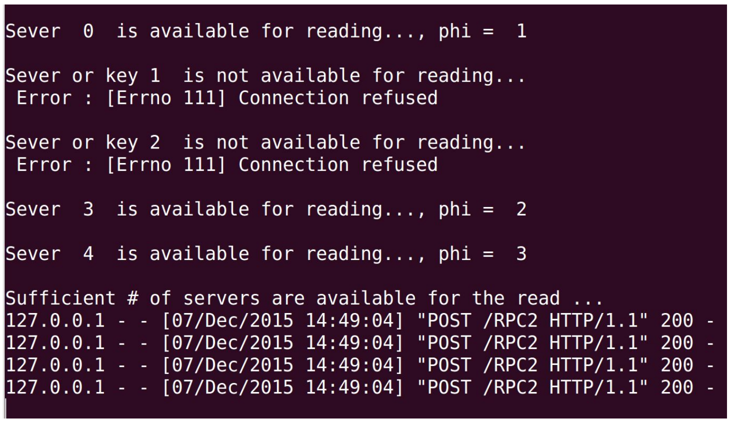
Figure 6. Screenshot displaying number of servers working and condition for read operation is satisfied.



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
sample
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ 
```

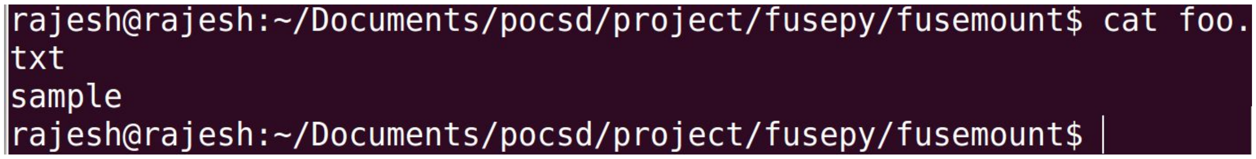
Figure 7. Screenshot showing successful read operation.

After killing 2 servers: Number of servers available(3) = Qr(3)



```
Sever  0  is available for reading..., phi =  1

Sever or key 1  is not available for reading...
 Error : [Errno 111] Connection refused

Sever or key 2  is not available for reading...
 Error : [Errno 111] Connection refused

Sever  3  is available for reading..., phi =  2

Sever  4  is available for reading..., phi =  3

Sufficient # of servers are available for the read ...
127.0.0.1 - - [07/Dec/2015 14:49:04] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:49:04] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:49:04] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 14:49:04] "POST /RPC2 HTTP/1.1" 200 -
```

Figure 8. Screenshot displaying number of servers working and condition for read operation is satisfied.

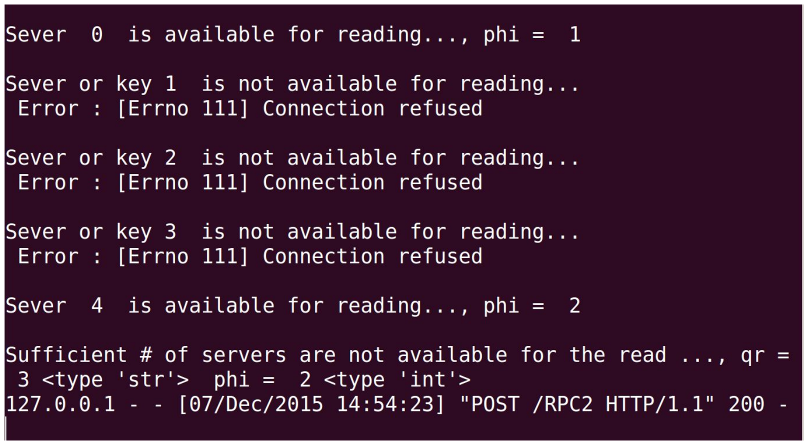


```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
sample
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ 
```

Figure 9. Screenshot showing successful read operation.

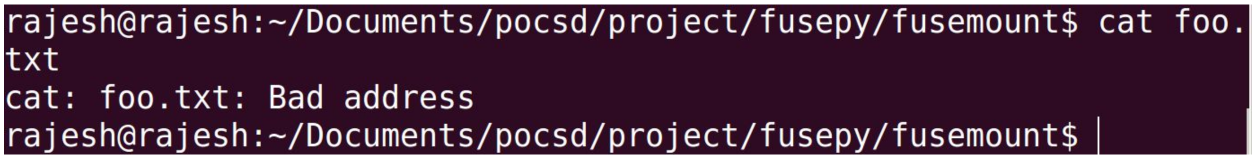
The reads were successfully completed even without the availability of two servers since 3 servers were available for read.

Case iv. Now further killing one more server. The number of available servers goes below 3. As this does not fulfil our condition of read, the read should fail.



```
Sever  0  is available for reading..., phi =  1

Sever or key 1  is not available for reading...
 Error : [Errno 111] Connection refused

Sever or key 2  is not available for reading...
 Error : [Errno 111] Connection refused

Sever or key 3  is not available for reading...
 Error : [Errno 111] Connection refused

Sever  4  is available for reading..., phi =  2

Sufficient # of servers are not available for the read ..., qr =
 3 <type 'str'>  phi =  2 <type 'int'>
127.0.0.1 - - [07/Dec/2015 14:54:23] "POST /RPC2 HTTP/1.1" 200 -
```

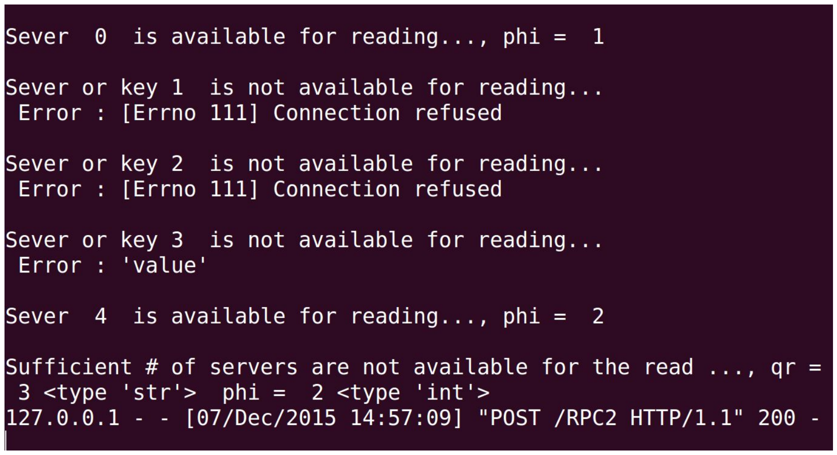
Figure 10. Screenshot displaying number of servers working and condition for read operation is **not** satisfied.



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
cat: foo.txt: Bad address
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ 
```

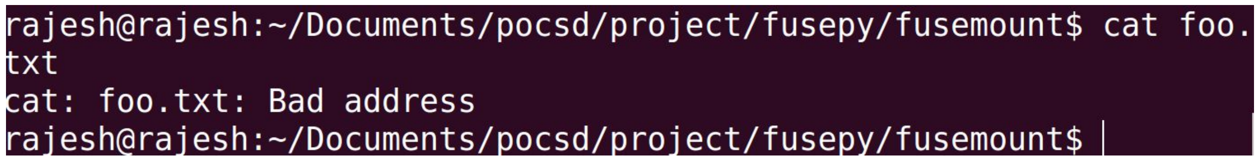
Figure 11. Screenshot showing unsuccessful read operation

Case v. Now restarting the server 3 which had failed earlier and then trying to read,



```
Sever  0  is available for reading..., phi =  1

Sever or key 1  is not available for reading...
 Error : [Errno 111] Connection refused

Sever or key 2  is not available for reading...
 Error : [Errno 111] Connection refused

Sever or key 3  is not available for reading...
 Error : 'value'

Sever  4  is available for reading..., phi =  2

Sufficient # of servers are not available for the read ..., qr =
 3 <type 'str'>  phi =  2 <type 'int'>
127.0.0.1 - - [07/Dec/2015 14:57:09] "POST /RPC2 HTTP/1.1" 200 -
```

Figure 12. Screenshot displaying number of servers working and condition for read operation is again satisfied.



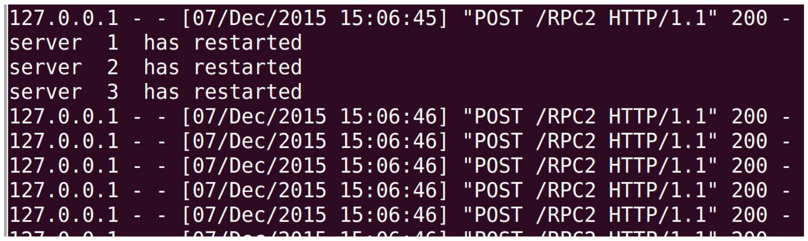
```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
cat: foo.txt: Bad address
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ 
```

Figure 13. Screenshot showing successful read operation

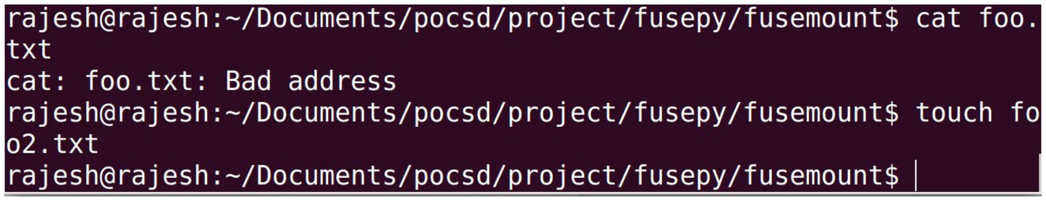
Even though the server is available (after restart) the read still does not finish as it does not have the key/value from which we have to read from. Therefore this read operation also does not finish although all the servers are available.

Case vi. Restarting the unavailable servers and performing a write operation:

On issuing a write after restarting all the unavailable servers, the number of keys are checked in each server if there are no keys present in any server then, it indicates that server has crashed and restarted.
When a server restarts and has no keys, we copy all the keys and values to this restarted server from the adjacent server.



```
127.0.0.1 - - [07/Dec/2015 15:06:45] "POST /RPC2 HTTP/1.1" 200 -
server  1  has restarted
server  2  has restarted
server  3  has restarted
127.0.0.1 - - [07/Dec/2015 15:06:46] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 15:06:46] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 15:06:46] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 15:06:46] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 15:06:46] "POST /RPC2 HTTP/1.1" 200 -
```

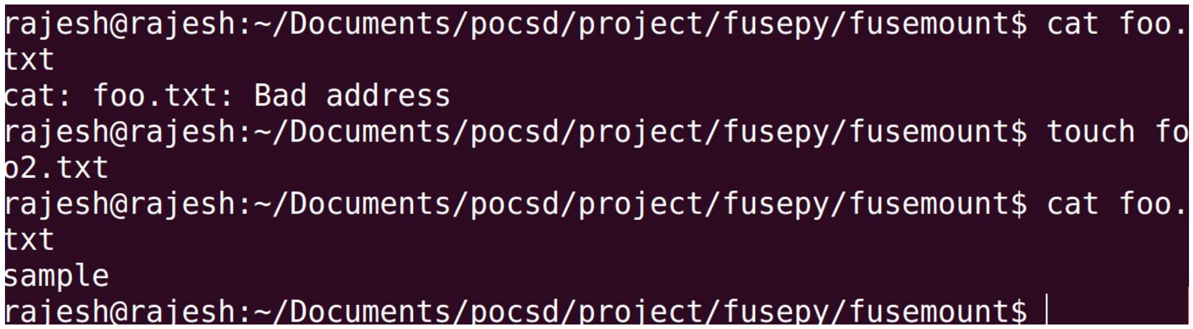


```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
cat: foo.txt: Bad address
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ touch fo
o2.txt
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ 
```

Figure 14. Screenshot showing successful write operation after restarting servers

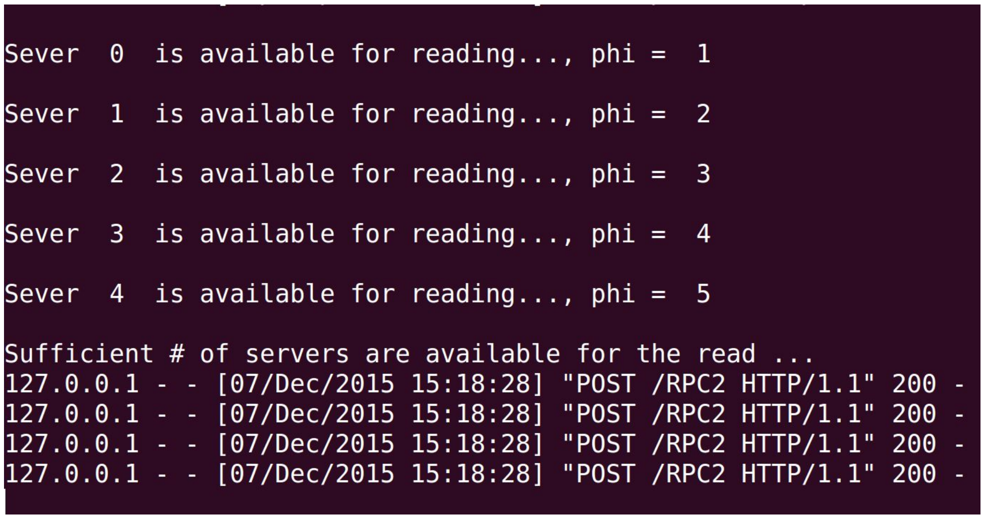
Also the write operation finishes.

Now since all the servers are available and have the latest contents trying to read from the file (foo.txt) which we were unable to read from earlier should be possible.



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
cat: foo.txt: Bad address
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ touch fo
o2.txt
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
sample
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ 
```

Figure 15. Screenshot displaying successful read from crashed server.



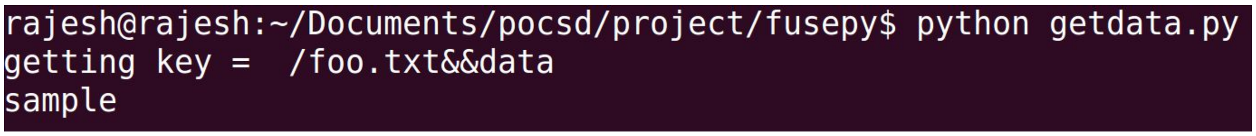
```
Sever  0  is available for reading..., phi =  1

Sever  1  is available for reading..., phi =  2

Sever  2  is available for reading..., phi =  3

Sever  3  is available for reading..., phi =  4

Sever  4  is available for reading..., phi =  5

Sufficient # of servers are available for the read ...
127.0.0.1 - - [07/Dec/2015 15:18:28] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 15:18:28] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 15:18:28] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 15:18:28] "POST /RPC2 HTTP/1.1" 200 -
```

Figure 16. Screenshot displaying number of servers working and condition for read operation is again satisfied.

Now evaluating for corruption:
getdata.py is a file which just reads the value from the key on a particular server.

testbench.py is a file which corrputs and displays the key which it has corrupted

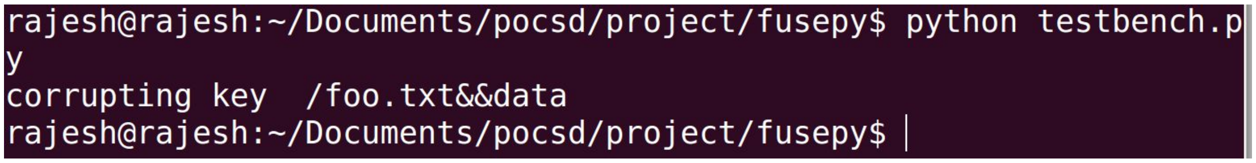
Before corruption:



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy$ python getdata.py
getting key =  /foo.txt&&data
sample
```

Figure 17. Screenshot displaying original contents of key.

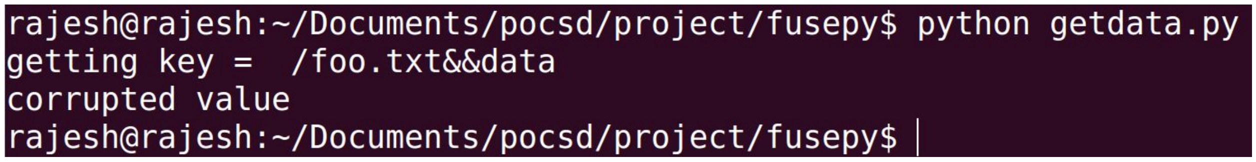
Corrupting the key:



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy$ python testbench.p
y
corrupting key  /foo.txt&&data
rajesh@rajesh:~/Documents/pocsd/project/fusepy$
```

Figure 18. Screenshot displaying /foo.txt&&data key has been successfully corrupted.

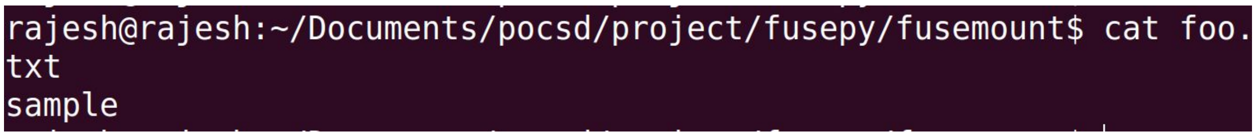
After corruption:
Running the same getdata.py as before, to see the corrupted value. The value of the key has changed from “sample” to “corrupted value”



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy$ python getdata.py
getting key =  /foo.txt&&data
corrupted value
rajesh@rajesh:~/Documents/pocsd/project/fusepy$
```

Figure 19. Screenshot displaying corrupted contents of /foo.txt&&data key.

When the client now performs a read operation the values should be restored to the correct ones:

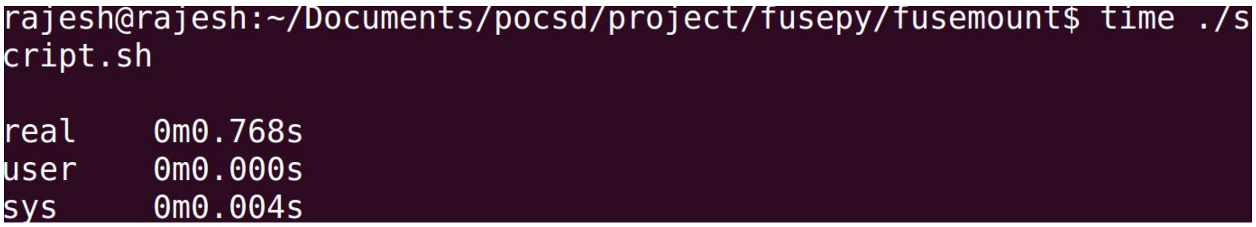


```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ cat foo.
txt
sample
```

Figure 20. Screenshot displaying corrected contents of /foo.txt&&data key during read..

The read operation thus restores the correct value.

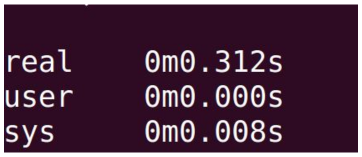
Writing a files with more than 10 lines of text.
When all servers are available for write:



```
rajesh@rajesh:~/Documents/pocsd/project/fusepy/fusemount$ time ./s
cript.sh

real	0m0.768s
user	0m0.000s
sys	0m0.004s
```

Figure 21. Screenshot displaying time required for writing.

Reading it from available servers:



```
real	0m0.312s
user	0m0.000s
sys	0m0.008s
```

Figure 21. Screenshot displaying time required for reading..

It takes lesser time for the read as only fewer servers need to be read from.

Now reducing the number of available server to the bare minimum 3.

5. POTENTIAL ISSUES

1. Availability of the servers is checked serially, if a already checked server fails after the check and before the write, it can lead to potential issues.
2. Keys present in the data server even after the files have been removed using rm on command line.

References:

1. About RAID levels and ClearCase, IBM Corporation.
2. Paul Krzyzanowski, "*Distributed File Systems ieee", Rutgers University.* https://www.cs.rutgers.edu/~pxk/417/notes/16-dfs.pdf
3. Uehara, M., "RAIDv: Extensible RAID1 Based on Voting," in *Network-Based Information Systems (NBiS), 2012 15th International Conference on* , vol., no., pp.128-133, 26-28 Sept. 2012 doi: 10.1109/NBiS.2012.41
4. Pham, H., "Optimal cost-effective design of triple-modular-redundancy-with-spares systems," in *Reliability, IEEE Transactions on* , vol.42, no.3, pp.369-374, Sep 1993 doi: 10.1109/24.257819

APPENDIX:

README
To run the code:
Since the guidelines did not mention anything about mediator port the mediator is fixed at:
http://127.0.0.1:51234

Sample Arguments:

```
python FileSystem.py fusemount http://127.0.0.1:51234
```

```
python mediator.py 3 5 http://127.0.0.1:51200 http://127.0.0.1:51235 http://127.0.0.1:51236
http://127.0.0.1:51237 http://127.0.0.1:51238 http://127.0.0.1:51239
```

```
python dataserver.py 51235 &
python dataserver.py 51236 &
python dataserver.py 51237 &
python dataserver.py 51238 &
python dataserver.py 51239 &
python metaserver.py 51200
```

Tested successfully for Qr = 3 Qw = 5