EEL 5737 Principles of Computer System Design

Report for Final Project

on

Fault Tolerant File System Using Replicated Data Servers

Submitted to:

Prof. Renato Figeurado,
Department of ECE,
University of Florida

Submitted By:

Anumula, Sai Raghu Vamsi **UFID: 49939544**
Bopanna, Vamsi Kumar **UFID: 51671924**
Mattapalli, Jaivardhan **UFID: 53991381**

Contents

Table of Figures

Abstract

With increasing complexity in computer systems, rate of faults and failures increase proportionally. The efforts put on controlling these faults are not proving to be as useful as expected. Hence another approach to overcome the issues with faults and failures is making the systems fault-tolerant. With this view several fault tolerant systems have been developed with various approaches. In this project, we use principle of data redundancy and replicate the data among several servers to serve a client with correct data whenever it requires. We built this fault tolerant file system with hierarchical FUSEPY file system as base and replicated data server in number. The system supports quorum approach to resolve the data from multiple servers. As expected, results of our project show that a single server fault can not only be tolerated but can be corrected at server side transparent to the client.

1. Introduction and Background

Reliable performance of computer systems has been a requirement since the construction of computing systems. Improper functioning of functional units in a system is manifested as fault in the system. The art and science of building computing systems that continue to operate satisfactorily in the presence of faults is called fault-tolerant computing. It can be achieved by using redundancy in data i.e. replicating the data in multiple sites.

Replication is the process of copying and maintaining same data objects in multiple servers that make up a replicated file system. It creates multiple copies of data items and stores them at different sites. Main idea is to increase the reliability so that if a node fails at one site, data can be accessed from a different site. Replication can improve the performance and protect the availability of applications because alternate data access options exist. The purpose of replication is to prevent damage from failures or disasters that may occur in one location, or in case such events do occur, improve the ability to recover.

Bianca Schroeder and Garth A. Gibson [1] discusses component failure in large-scale IT installation systems. They argue that these IT systems need better system design and management to cope with more frequent failures and expect increasing the levels of redundancy can solve this issue. Many

With the concepts discussed in the above paper, we have implemented a replicated file system where in we store data into multiple replicated servers and it proved to be fault-tolerant in our tests. We have also incorporated Replication Transparency and Failure Transparency in the client-server model. Replication Transparency should be mainly incorporated for the client-server systems, which replicate the data at two or more sites for more reliability. The client generally should not be aware that a replicated copy of the data exists. The clients should also expect operations to return only one set of values. Failure Transparency enables the concealment of faults, allowing user and application programs to complete their tasks despite the failure of hardware or software components. Fault tolerance is provided by the mechanisms that relate to access transparency. The client server systems are more prone to failures as any of the component may fail which may lead to degraded service or the total absence of that service.

2. Assumptions

- Meta-server is fail proof
- *Md5 hashlib* generates same checksum every time a same data is fed.
- Q_R is given as a value greater than 3 to test fault tolerance.

3. Design

3.1. Concept

The replicated file system developed in this project can be divided into four major components:

1. Client,
2. Mediator,
3. Meta-server and
4. N data-servers, where N is the number of replicated data servers.

Whenever the client requires to store/retrieve the data, it requests the mediator which then stores/ retrieves the data from multiple data-servers in a transparent manner i.e. without client knowing anything about it replication. The communication between mediator and meta/data-servers takes place through XMLRPC protocol.

Error in data or a server crash is tolerated in this design. For every write checksum is calculated and stored in meta-server which is assumed to be fail proof and whenever a data is read from data servers, the checksum is calculated again and validated for errors.

The functioning of each of the components is explained below in detail.

3.1.1. Client

Client in this design is responsible for mounting the file-system and supporting all the file-system operations such as creating file/directory, writing data to a file, reading data from a file, changing meta-data etc. Unlike in base file system, where the file-system directly writes into server, here client just requests Mediator to write to the servers and requests Mediator to read the specific data from the servers. Thus Client is abstracted from what is happening at a lower level. Client is completely unaware of how mediator is writing to servers and how it is providing data from servers.

3.1.2. Reliable layer (Mediator)

A mediator acts as a bridge between the client and servers. It accepts requests from client and communicates with servers to write or read data over XMLRPC protocol. It writes data to all available data servers when a client requests it to put data. It reads from all the available data-servers and uses quorum approach to resolve data received when a client requests to get data.

It also uses a '*md5 hashlib* **checksum calculator** which is used to store the check sum in meta-server when a data is written to data-servers and re calculates checksum when data is read from data-servers and compares it with that stored in meta-server. A difference in checksum indicates that the data read from particular data-server is corrupted and should be discarded.

Mediator makes use of quorum approach where in it decides which data to be returned to client based on number of data-servers that respond with similar and correct data. Only if the number is above Q_R (minimum number of servers that should respond with correct data), the data is assumed to be correct and will be returned to client.

3.1.3. Meta-Server

There is single Meta-server in the system which stores meta-data for all the files and directories in the file system. It also stores the list-nodes data for all the file system contents. Since meta-server is assumed to be fail-proof, we used it to store checksum for the data that is stored in data servers.

3.1.4. Data-Servers

Data-server cluster consists of replicated N servers which have copies of same data. Mediator writes to each of them when the client requests to put data onto servers. All these servers respond to mediator with their own data which will be validated by the mediator

3.2. Detailed design

To make the file system replication transparent and failure transparent, we created a reliable layer, a mediator which isolates client from data servers and thereby creates abstraction and makes the system modular. Client only communicates with this Mediator for storing data or reading the data.

Mediator takes the responsibility to write the contents to meta/data servers and return the contents to client from meta/data servers.

We have used '*md5*' checksum generator from '*hashlib*' library for generating Checksum while writing data into data servers and while validating the received data by comparing with checksum stored during write operation.

The control flow for different operations is explained below.

Storing Contents:

When application wants to store some data, client invokes reliable layer with contents it wants to store and type of contents i.e. meta data or list nodes or data etc. Reliable layer then checks the type of contents and writes the contents to appropriate server. If the contents to be written are meta-data or list nodes, then they are written to meta server otherwise, they are written to all the data servers.

The writing to data servers is not exactly same as writing to meta-server. Whenever the client requests a *put*, mediator calculates checksum for the contents and writes it to meta-server and writes data and its corresponding checksum to data-servers. To maintain reliability, Mediator tries to make sure that data is written to at least Q_W number of data-servers.

For writing to a data server, mediator tries to connect to server for a small fixed number of times. If not connected, it enters into a list which contains details of servers to which write failure was observed and proceeds with attempting write to next data-server. If the total number of servers to which data is successfully written is less than Q_W, then Mediator tries to write to failed servers once again. If the write to these failed servers still fails, then Mediator doesn't attempt to write them again.

Client Layer –

```
put(key,value):
      reliable_put(path,key,value) # Call reliable put
```

Reliable Layer –

```
Reliable_put(path, key, value):
      if META_DATA or LIST_NODES:
            Write to Meta_Server.
      else
            Update_Checksum() #Calc. Checksum and write Checksum to Meta_Server

            for N Servers:
                  Try to connect to server
                  if connected:
                        Write data to that server
                        Add Server name to LIVE_SERVERS list
                  else:
                        wait for short period and try atleast 5 times.
                        if still not connected:
                              Add server name to FAILED_SERVERS.

            if Number of LIVE_SERVERS < Qw:
                  Re-attempt write to failed servers (as in above loop).

Update_Checksum():
      Calculate Checksum using HASHLIB library.
      Write Checksum along with Path and Key to Meta_Server.
```

Reading Contents:

When application wants to read some data, client invokes reliable layer with contents it wants to read and type of contents i.e. meta data or list nodes or data etc. Reliable layer then checks the type of contents and reads the contents from appropriate server. If the contents to be read are meta-data or list nodes, then they are read from meta server otherwise, they are fetched from all the data servers.

The fetching of contents from data servers is not exactly same as reading from meta-server. Whenever the client requests a *get*, Mediator reads from all the replicated data-servers and validates each returned data against the checksum stored in meta-server. To maintain reliability, Mediator tries to make sure that data is returned by at least Q_R number of data-servers.

For reading from a data server, mediator tries to connect to server for a small fixed number of times. If not connected, it enters server name into a list which contains details of servers to which read failure was observed and proceeds with attempting read from next data-server. If the total number of servers from which data is successfully read is less than Q_R, then Mediator returns NONE to client, otherwise, it validates checksum for received data. If no server responded with correct data, then Mediator returns NONE to client. Otherwise, it tries to write correct data to faulty servers and returns good data to Client

Client Layer -

```
get(key):
      reliable_get(path,key) # Call reliable get
```

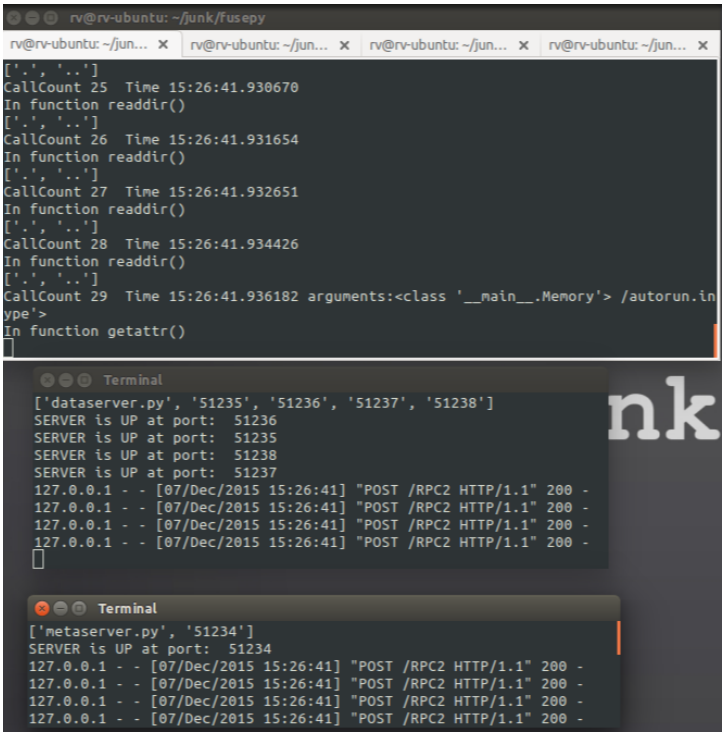
Reliable Layer -

```
reliable_get(path, key):
      if META_DATA or LIST_NODES:
            Get data from Meta_Server.
      else
            Update_Checksum() #Calc. Checksum and write Checksum to Meta_Server

            for N Servers:
                  Try to connect to server
                  if connected:
                        Read data from that server
                        Add Server name to LIVE_SERVERS list
                        Add Data to RECEIVED_DATA list
                  else:
                        wait for short period and try atleast 5 times.
                        if still not connected:
                              Add server name to FAILED_SERVERS list.
                              Add NONE in corresponding RECEIVED_DATA list

            if Number of LIVE_SERVERS < QR:
                  return NONE to Client

            Data_To_Client = Validate_Checksum (path, key, RECEIVED_DATA)
            #Check how many servers returned correct data and repair faulty
            Server

            return Data_To_Client to Client


Validate_Checksum(path, key, RECEIVED_DATA):
      Read Checksum using Path and Key from Meta_Server.
      for all elements in RECEIVED_DATA:
            Calculate Checksum using HASHLIB library
            Compare with Checksum from Meta_Server
            if matched:
                  Increment GOOD_DATA_CNT.
            else:
                  Increment BAD_DATA_CNT.

      if GOOD_DATA_CNT = 0:
            return NONE
      else:
            Write correct data to faulty Data_Server/s #REPAIR HAPPENS HERE
            return (Good element from RECEIVED_DATA)
```

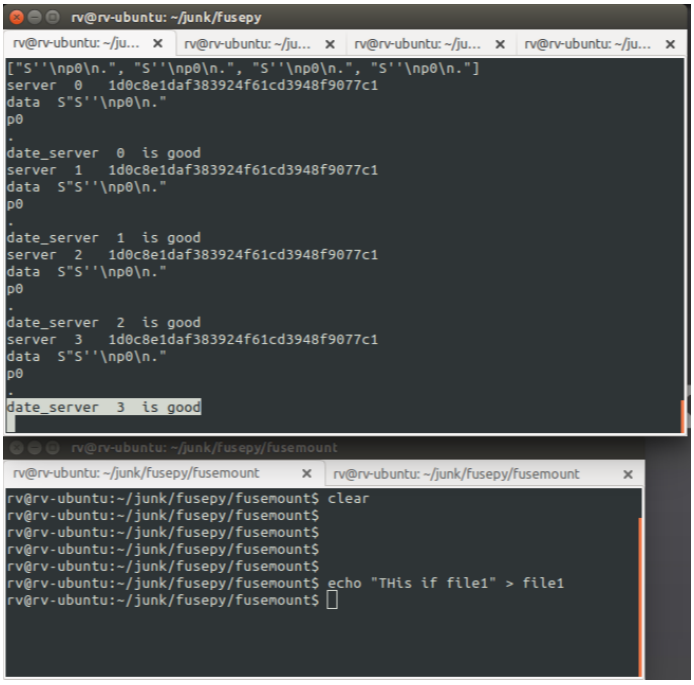
4. Testing

TEST 1

STEPS -1: Running 4 data servers and 1 meta server. with QR=2 and QW=2

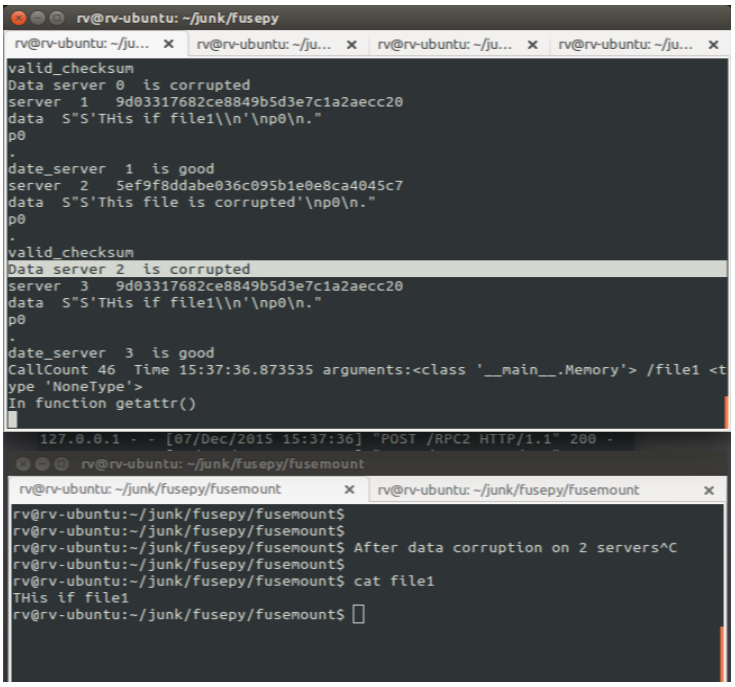


STEP-2: Creating File1. File System reports that good data has been written.

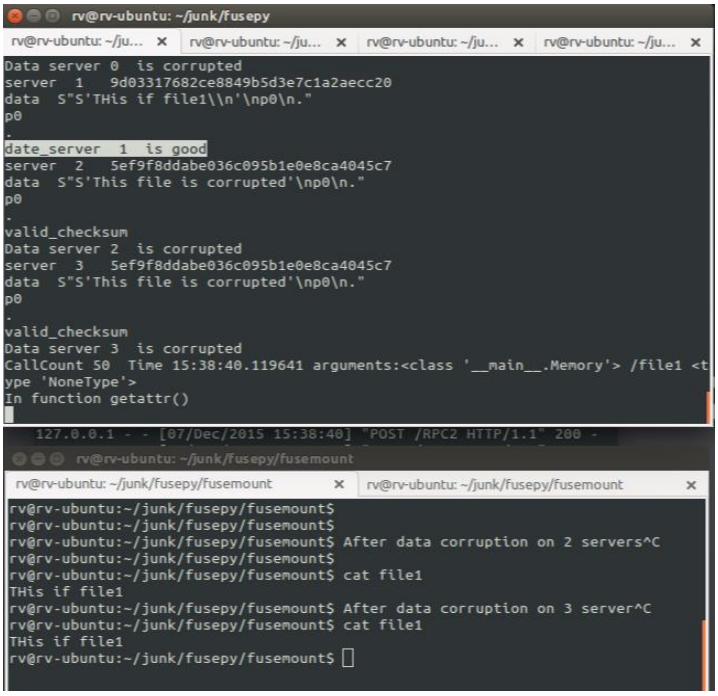


STEP-3:

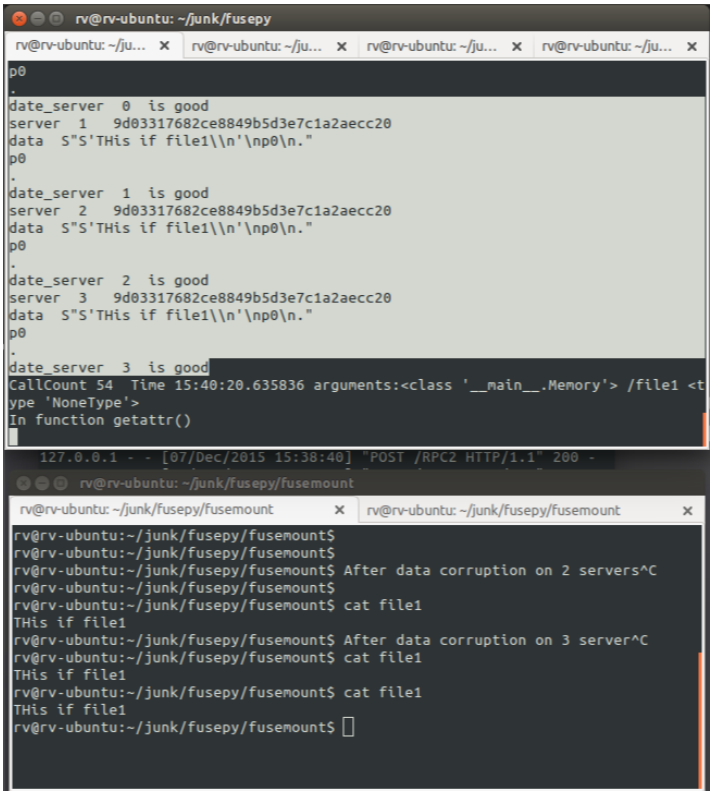
Here the data has been corrupted on server 0 [51235] & server 2 [51237. Hence the subsequent "cat file1" in the bottom terminal still prints out correct data. The log file shows in which servers data has been corrupted and corrects them using good data at the end of the step.



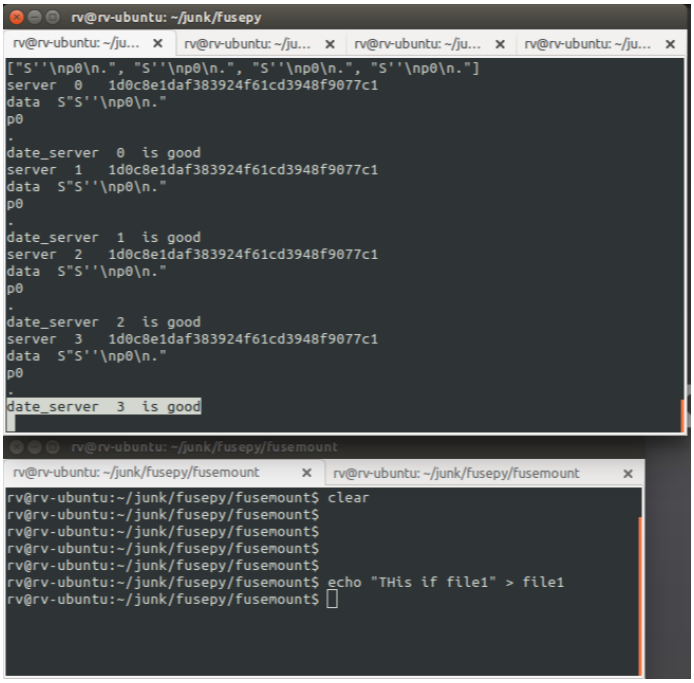
STEP 4: Here the data is corrupted on 3rd server, so it reports that data has been corrupted on 3 servers. Still in this case if uses the good data on the 4th server to correct rest of them.



STEP 5: data corrected on all the servers



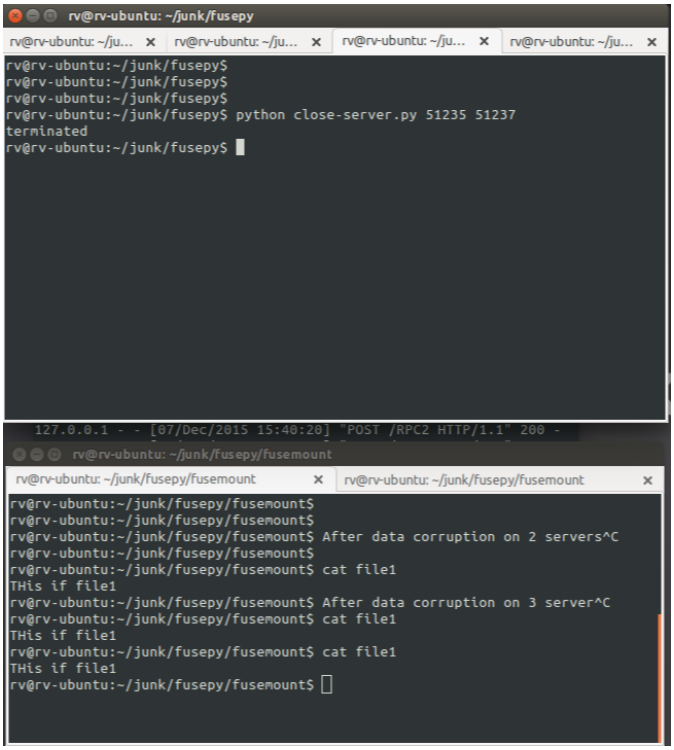
STEP6 : data corrected on all the servers



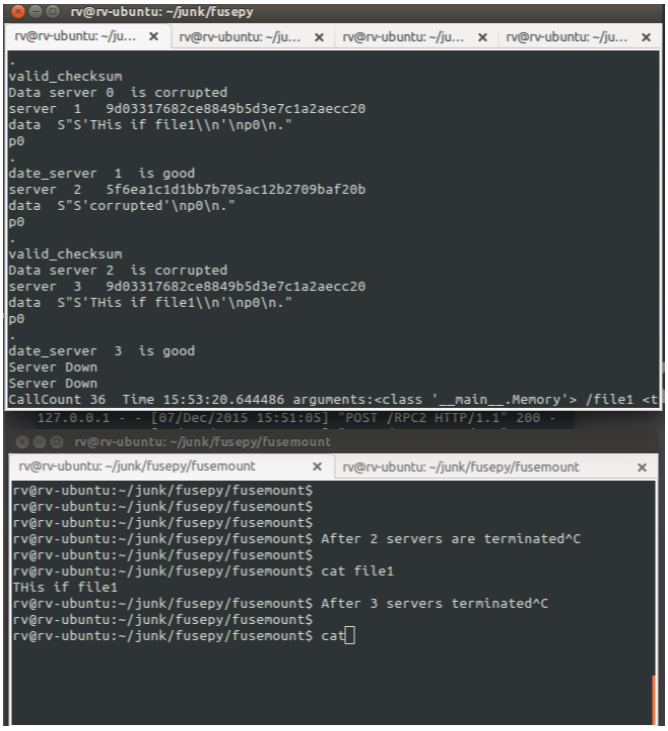
TEST- 2

Step1: Terminate two servers and try to read the data. The data is succesfully returned even in this case as it is available in other servers.

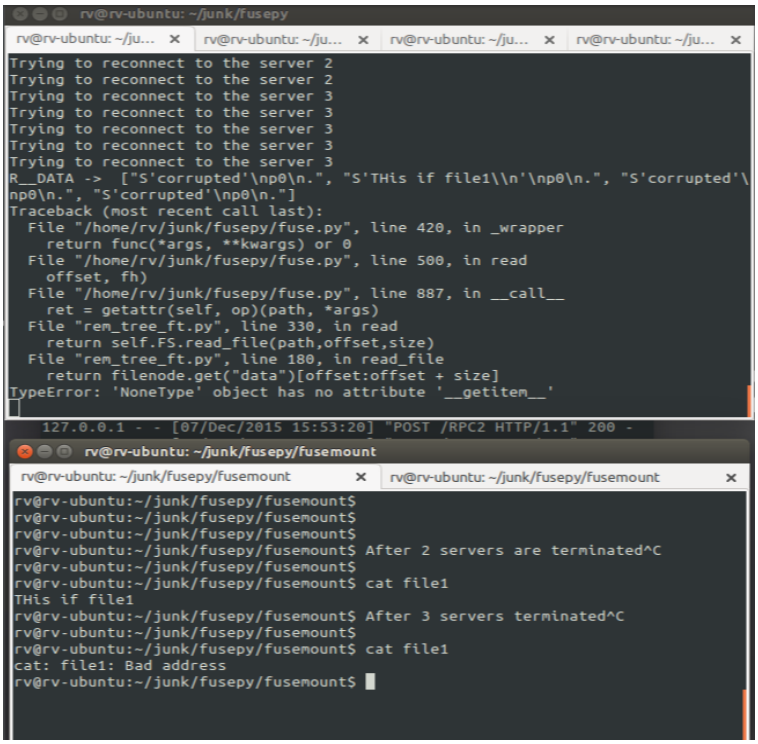
However when 3 servers are terminated, no data is returned.



2_stop

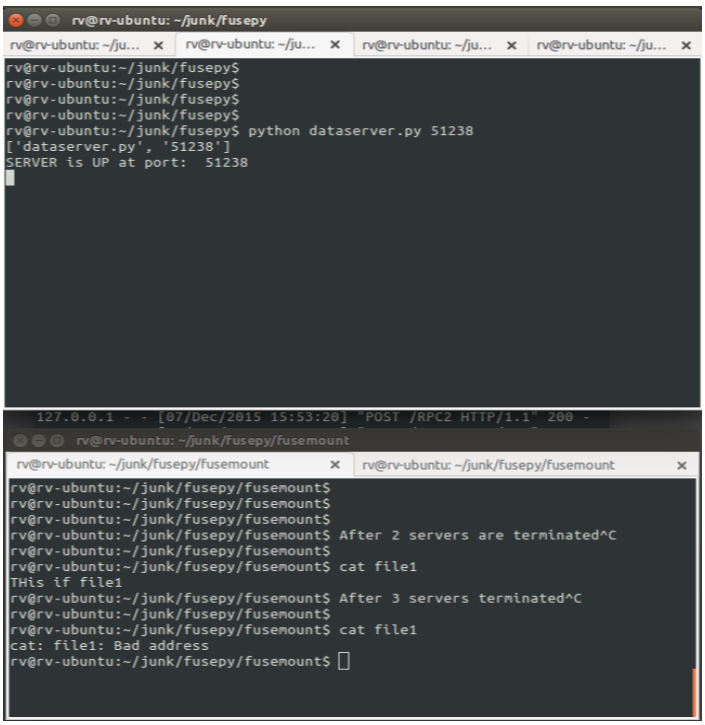


3 stop

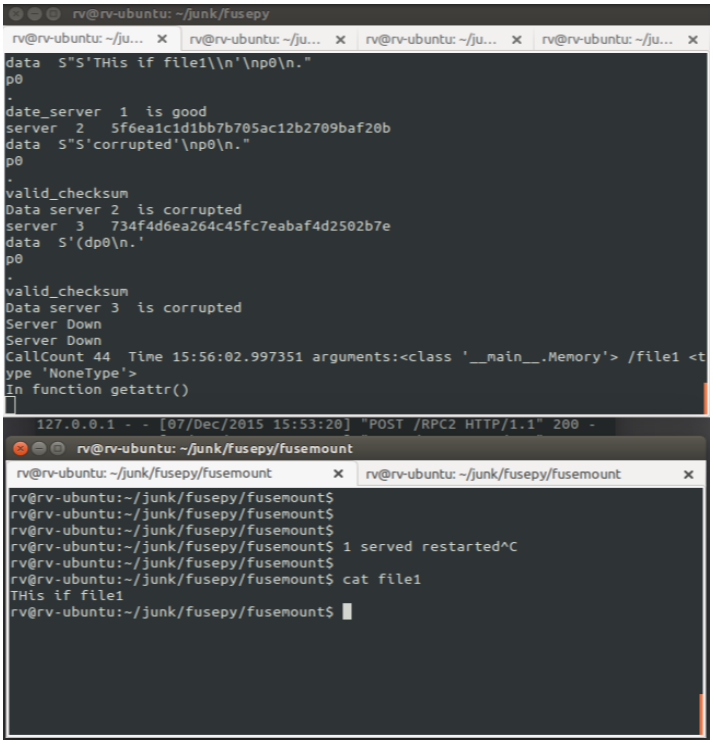


TEST- 3

1_restart



1s_restart



5. Evaluation

We have implemented a file system which is fault tolerant by virtue of redundancy in data. We achieved it by replicating data into multiple servers and checking all of them when read. Using the inputs provided by User, we connect to the data servers and meta-servers. Q_R and Q_W indicates at least how may servers should respond with correct data and at least to how many servers data has to be written.

We found that our file system can tolerate errors in a server or even a server cache i.e. client receives correct data even if a server crashes. Even better thing is that the faulty server is corrected using the correct data from other servers. We have made it more robust by correcting all other faulty servers even if single server responds with correct data.

The mechanism of how this works is explained below:

- If Number of servers responded to a read request is less than Q_R, return NONE to Client.
- If Number of servers responded to a read request is greater than or equal to Q_R then,
    - If at least one Data server responded with correct data, repair all other servers and return correct data to client
    - Otherwise, return NONE to client

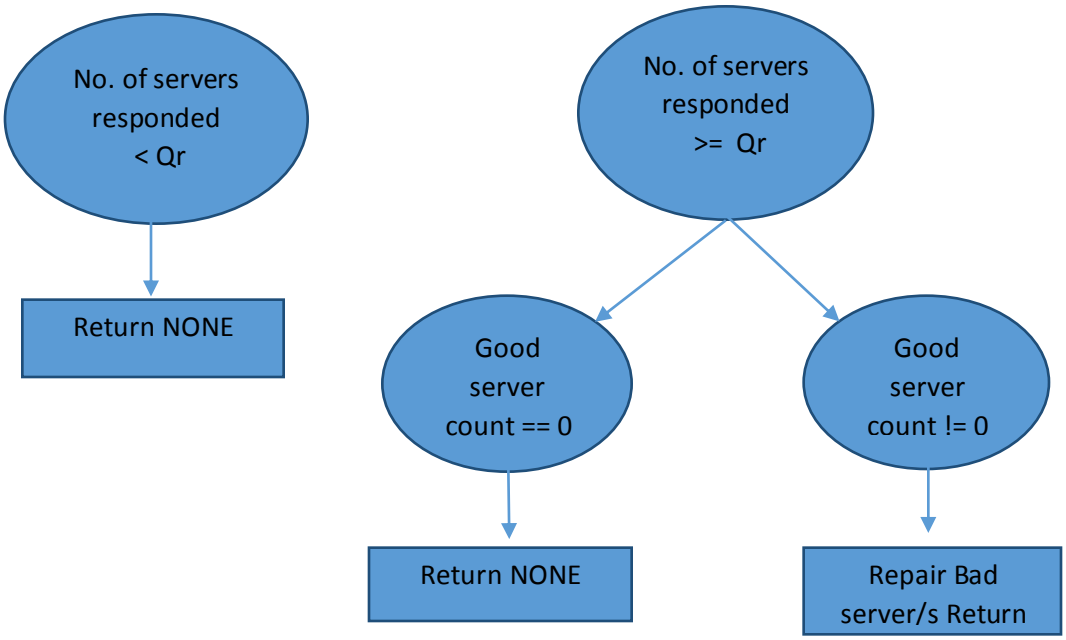


Figure: Return values for GET under various conditions

id	Operation	QR	QW	# of servers with good data before operation	# of servers with good after operation	No.of server running out of 4 servers	result
1.	GET	2	2	4	4	4	ok
2	stop servers1,2	2	2	4	4	2	-
3	PUT	2	2	2	2	2	OK - Logs as PUT fail but doesn't give out error
4	server1 restarted	2	2	2	2	3	-
5	PUT	2	2	2	3	3	OK - Logs as PUT fail but doesn't give out error
6	server2 restarted	2	2	3	3	4	-
7	GET	2	2	3	4	4	**ok** - uses good server data to correct the data on 4th server

If Number of servers to which write was successful is less than Q_W, then a Warning message is printed on screen. The data in servers to which it is properly written still stays and can be read when needed.