Fault-Tolerance FUSE FileSystem

Group: Lalitha Mathi(UF ID : 29341966), Punam Mahato (UF ID : 68382764)

Fault tolerance is a crucial design consideration for mission-critical distributed real-time systems to meet their dynamic performance demand without failure. And since the data demands are increasing day by day the distributed file systems continue to grow more complex. Real large-scale systems have to deal with increasingly large amounts of data in the range of petabytes to exabytes which a single server is unable to store and process. So data is distributed across multiple servers to increase storage space and it also imparts flexibility.

Starting from HW2 we proceeded from in memory storage using memory.py to single client server design to multiple client server design. The following diagram represents it schematically:

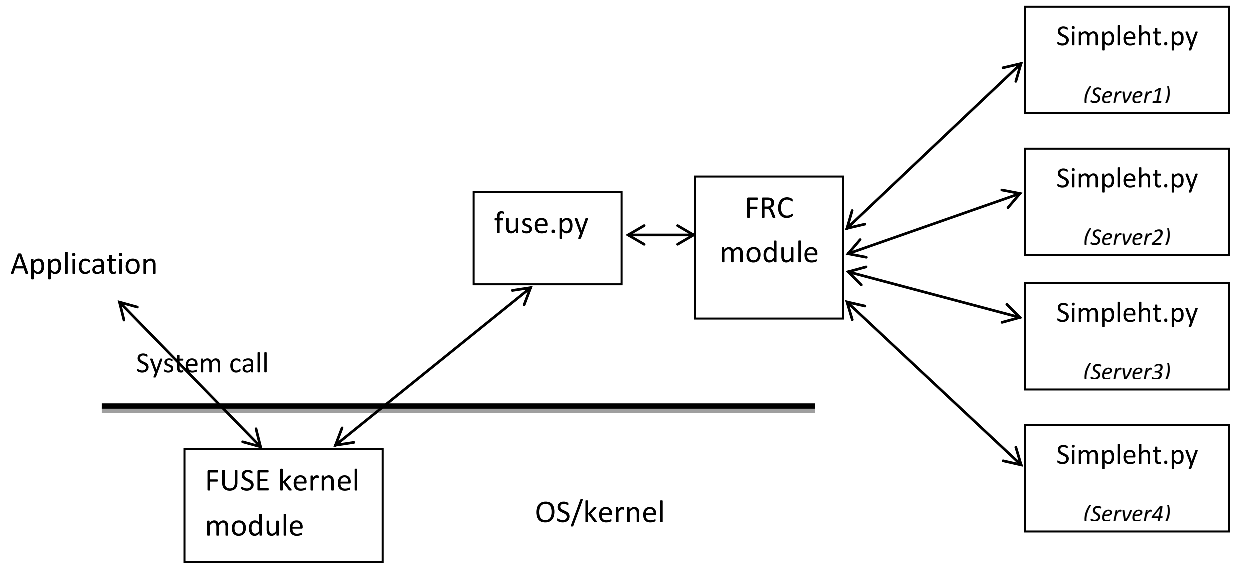


Figure1: Layers in the Implementation diagram of our project

Introduction:

Even though we apply error correction techniques, components can fail every now and then. So replicating data over multiple servers is a necessity in order to provide availability and consistency. Also, malicious intrusions can lead to error in storing and fetching of data. Software errors are a major cause of outages and they are increasingly exploited in malicious attacks. By the application of fault tolerant techniques we try to achieve successful performance of the distributed system even in the presence of such faults. We implemented Cyclic Redundancy Checksum for error detection and Advanced Encryption Standard(AES) algorithm for encryption and decryption. For the safety and availability of data even in case of data corruption and server faults we implemented redundancy by copying data to multiple servers and the concurrency issues were also dealt with. The general information and implementation of each of the mentioned concepts are described in details in the next sections.

Background:

Message Digest Algorithm(MD5):

The MD5 message-digest algorithm is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically expressed in text format as a 32 digit hexadecimal number. The algorithm can be used for cryptographic applications and also to verify data integrity. In our project MD5 hashing technique is used in AES cryptography and for distributing data among multiple servers in a round-robin fashion. MD5 was widely used before 1996 but after the detection of a design flaw, it was replaced by series of SHA algorithms.

Advanced Encryption Standard(AES):

AES is used to protect classified information in software applications, firmware and hardware implementations throughout the world. AES encryption transforms data in blocks of 128 bits into ciphered format. Decryption converts the ciphered text back to data. AES comprises of three block ciphers, AES-128, AES-192 and AES-256 that have cryptographic keys of 128-, 192- and 256-bits length, respectively. AES is basically used for symmetric or secret-key ciphers which use the same key for encrypting and decrypting. Both the sender and the receiver must know the key in order to decipher the data. There are 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys and each round consists of several pre-processing steps such as substitution, transposition and mixing of the input data and transform it into the ciphered data. The following substeps show one one round of AES encryption:
SubBytes() – Shuffles elements in each row in the matrix using a lookup table(substitution)
ShiftRows() – Shuffles the rows in the matrix(permutation)
MixColumns() – Combines elements in each column in the matrix
AddRoundKey()- Data is XORed with a key to encrypt

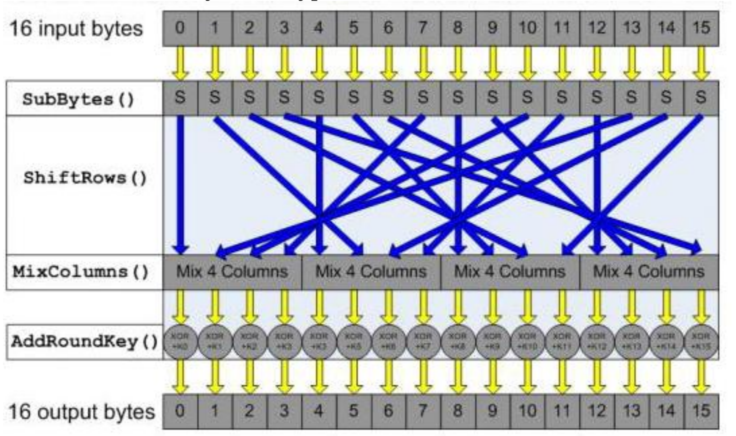


Figure2: AES encryption(Reference: E. Conrad. "Advanced Encryption Standard. CISSP. 2013.)

We have used cipher-block chaining (CBC) mode of operation for AES algorithm. In this mode, before encryption of each data block, it is XORed with the previously ciphered data block. Thus resulting cipher depends on all the data blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block. Decryption is done by XORing the decrypted block with previous cipher block.

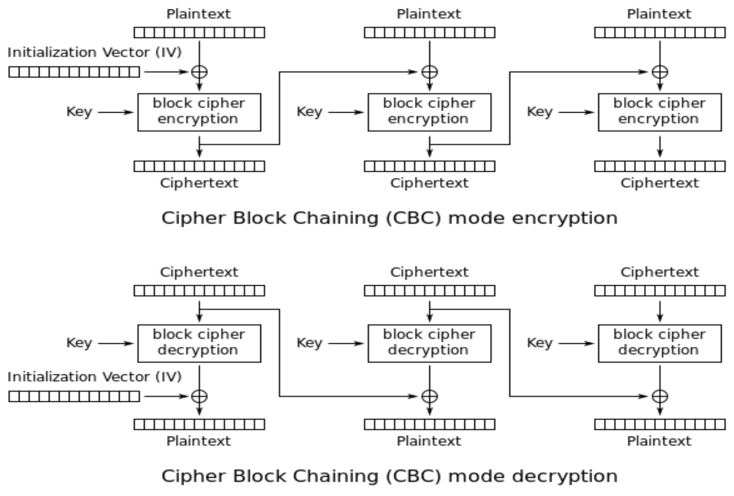

Plaintext
Initialization Vector (IV)
Key
block cipher encryption
Ciphertext
Cipher Block Chaining (CBC) mode encryption
Ciphertext
Key
block cipher decryption
Initialization Vector (IV)
Plaintext
Cipher Block Chaining (CBC) mode decryption

Figure3: CBC mode Encryption and Decryption(Source: Wikipedia)

AES is quite reliable because side-channel attacks(implementation weakness rather than weakness in the algorithm) are the only successful attacks against it. This is possible only if there is a weakness in the implementation or key management of certain AES-based encryption products. AES is more secure as it can use longer keys and is faster which makes it popular in applications that require either low-latency or high throughput.

Redundancy:

Redundancy is the ability of the system's infrastructure to provide additional servers at runtime for backup during server failure or temporary halting of primary server for maintenance. This provides backup of data in case of server faults like crashes and also in the case of corruption of the data stored on a particular server. Redundant servers can be used for fault detection, fault location, fault containment, fault recovery. Fault detection can be done by comparing the data in one server with its replica in another server. Similarly we can locate the faulty server by

N-Modular Redundancy (NMR) is a scheme where n independent servers replicate the same file. Triple-modular redundancy (TMR) is a specific case of NMR where 3 servers perform a process and output is produced by majority-voting scheme. If all servers are working fine the output given is same and the voting produces the correct result. If any one of the three server fails, the other two systems can mask the faulty server's output and hence give the correct result. This can be extended to n level redundancy.

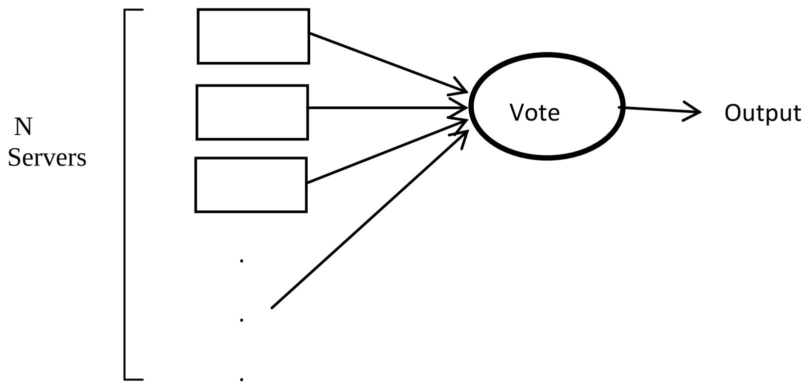


Figure4: N Modular Redundancy

Advantage of redundancy over encryption techniques is that in the case that data gets corrupted in the server and we cannot recover data through error detection and correction techniques we can fetch the correct data again from the redundant servers by resending requests.

Error Detection And Correction:

CRC is primarily used to detect errors that occur while transmission of data. A certain number of check bits called checksum are appended to the data being transmitted. The receiver can determine whether or not an error occurred in transmission by seeing whether or not the check bits agree with the data. If an error occurred, the receiver can send a negative acknowledgement back to the sender requesting it to retransfer. The simplest is to append a single bit, called the parity bit to detect 1 bit errors.

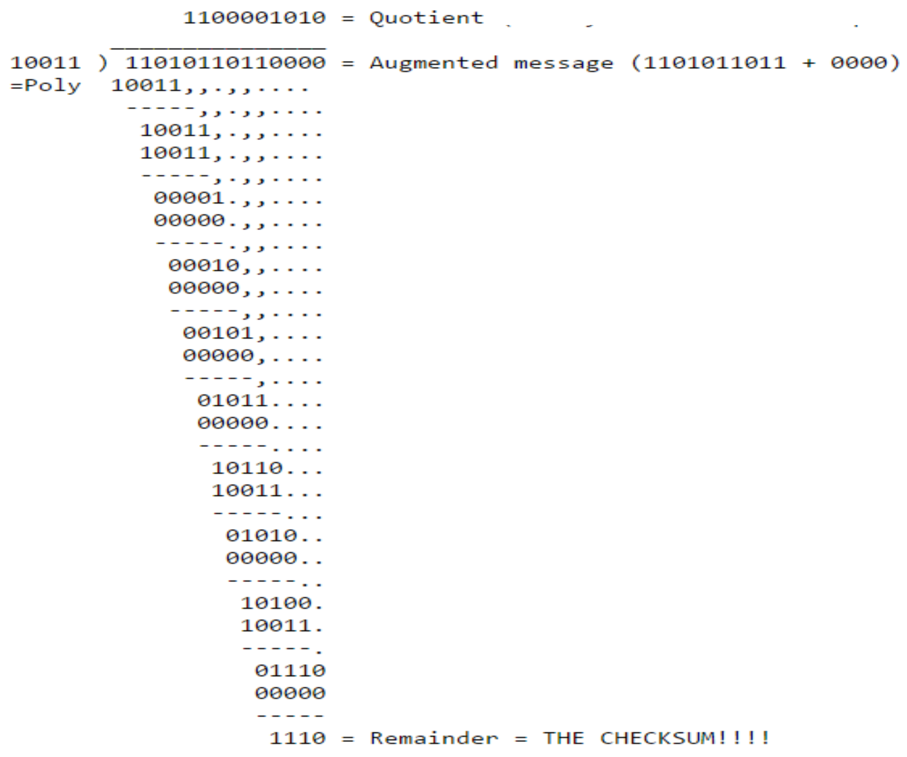
We used the built-in binascii module in python to find out CRC-32 of the data. The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations. To generate the same numeric value across all Python versions and platforms we used crc32(data) & 0xffffffff.

The data is treated as an enormous binary number and is divided by another fixed binary number. The remainder from this division is the checksum. At the receiver end the receiver can perform the same division and compare the remainder with the transmitted checksum. This can find out if there were any alterations while transmission.

We use a generator polynomial as the divisor which is the key parameter of any CRC algorithm. The width (the actual bit position of the highest bit) of the poly is very important as it dominates the whole calculation. For example, the width of 10011 is 4, not 5. For the purposes of example, we will chose a poly of 10011 (of width W of 4). W zero bits are appended to the message before the CRC is calculated.

Original message : 1101011011
Polynomial : 10011
Message after appending W zeroes : 11010110110000

Then the augmented message is divided by the polynomial using CRC arithmetic:



```
            1100001010 = Quotient
       _______________
10011 ) 11010110110000 = Augmented message (1101011011 + 0000)
=Poly  10011,,.,,....
        -----,,.,,....
         10011,.,,....
         10011,.,,....
          -----,.,,....
          00001.,,....
          00000.,,....
          -----.,,....
           00010,,....
           00000,,....
           -----,,....
            00101,....
            00000,....
            -----,....
             01011....
             00000....
             -----....
              10110...
              10011...
              -----...
               01010..
               00000..
               -----..
                10100.
                10011.
                -----.
                 01110
                 00000
                 -----
                  1110 = Remainder = THE CHECKSUM!!!!
```

The checksum is then appended to the message and the result transmitted. In this case the transmission would be: 11010110111110.

CRCs can provide quick and reasonable assurance of the integrity of messages delivered. However they are not suitable for protecting against intentional alteration of data. It doesn’t have any significant overhead of space and time. CRC is widely used because of its power of detecting burst errors. Burst errors are contiguous sequences of erroneous data symbols in messages etc.

Concurrency:
In case of multiple clients, there can be race condition when both clients are trying to read or write the same file. To prevent such situations we use locks. And any read write operation can be done atomically by the client who has the lock.

When data is stored in multiple servers, it is not necessary to fetch data from all the server, so only the primary server is in use. When the primary server fails the requests are sent to the backup servers and are processed by them. We can detect when the primary server becomes live again and the next requests are sent to it instead of the backup servers.

When the data is written or updated in a primary server the update should be reflected on all the redundant servers. Otherwise a client fetching data from a server(other than primary) can get un-updated data.

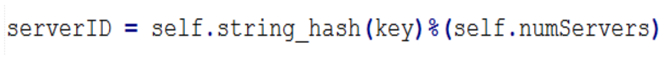
Implementation:

A. MD5:

MD5 hashing has been used in our project for evenly allocating the files to multiple servers and to encrypt the key used in AES cryptography. Data is allocated to multiple servers to provide a flexible extended memory system. Servers can be added without much effort and without the client even noticing but multiple server system are fraught with many issues such as availability of servers, server failures, data corruption in particular servers, concurrency issues, load balancing etc. One of the most important issue is how to distribute data evenly among the servers available. A significant part of the data is stored in each server, hence loss of one server's data may have significant effect on the entire data. In our project we have used MD5 hashing to determine which files are handled by which servers. The MD5 hashing was performed on the pathname of the file to obtain a 128 bit hash value.

The MD5 object can be fed with arbitrary strings using the update() method, and *digest* of the strings fed to it so far can be fetched with the digest () method which is a 16-byte string which may contain non-ASCII characters including null bytes. Hexdigest() method returns the digest as a string of length 32, containing only hexadecimal digits.

The modulus of this hash value and the number of servers available is used to determine in which server the file is stored in. This code is given by:



```
serverID = self.string_hash(key)%(self.numServers)
```

string_hash method is used to provide the 128 bit key while serverID provides a server alias that can be mapped to each file. Any file that is set up by the serverID will be allocated to a single server at all times. The key will generate the same hashed key no matter how many times the string_hash method is called. Hence this ID can be used to allocate the file to a specific server as long as number of servers does not change. serverID is used for get, put, readfile, writefile calls to establish a link with the particular server.

B. Advanced Encryption Standard

We have imlemented AES (128 bit) encryption in CBC(Cyclic Block Cipher) to encrypt the data and protect it from hackers/malicious intruders. For this we have installed the python PyCrypto package which has an AES module with encrypt and decrypt methods. PyCrypto AES package is a standard and widely used for AES implementations. The encryption method uses a initialization vector and a key.

The metadata and data which are in the value of the dictionary 'ht' are encrypted before we invoke a put call to copy the value into server and decrypted after we get the value from the get call. AES performs encryption and decryption on blocks of data of a particular size. For our project, we have chosen the block size to be 16.

In cases where the data is not a multiple of this block size we pad the data with zeros to achieve a block size that is a multiple of 16(in put call). Hence this requires unpadding on the other side(in get call). For this we have written two methods called pad and unpad that carry out the respective operations.
In padding we use 0x80 as first byte and add 0x00 bytes till it makes a size that is a multiple of 16.

The pad function we used is:

```
def pad(self,data):
  # return data if no padding is required
  if len(data) % BS == 0:
    return data
  # subtract one byte that should be the 0x80
  # if 0 bytes of padding are required, it means only a single \x80 is required.
  padding_required = (BS - 1) - (len(data) % BS)
  data = '%s\x80' % data
  data = '%s%s' % (data, '\x00' * padding_required)
  return data
```

The initialization vector is obtained by creating a new object and reading out the first 16 bytes of the random number generated by it. For this we import Random module. The key is obtained by MD5 hash digest as stated earlier in MD5 section for the password we selected. This password is located at the top of the python code document.

```
password = "FAULT_TOLERANT"
md5obj = md5.new()
md5obj.update(password)
key = md5obj.digest()
```

This is then used to initialize the AES object as:

```
ivp = Random.new().read(BS)
aes_encrptr = AES.new(key, AES.MODE_CBC, ivp)
```

the iv is used to encrypt padded data and obtain the cipher which is then attached to the iv and sent through the communication link buffer invoked in the put call. This implementation had given some errors initially and produced undesired outputs. Upon investigation, we found that in order to store cipher text we need an additional ASCII ARMOUR. This is done using base64 encoding. A simple base64 encoding converts everything to binary which can be obtained using a get call and then converted back. We also added a header"0x41" to the cipher to indicate ASCII ARMOUR. If base64 encoding is not present then a header of 0x00 is added to the cipher for purpose of indication. The entire string is then encoded using base64 and stored in the server.

```
cipher = '\x41' + base64.encodestring(cipher)
```

When the get call receives the cipher, we separate the header and determine whether ASCII ARMOUR is present. After determining the fact, we proceed to decode it using base64.decodestring. We arrive at the original cipher. The cipher has an iv that is determined by the first 16 bits. Using the iv and ads.decrypt, we decrypt the cipher to obtain the original blocks of data. After decrypting the cipher, we check for any bit errors or discrepancies in the data by using CRC or checksum function.

```
if data[0] == '\x00':
    data = data[1:]
elif data[0] == '\x41':
    data = base64.decodestring(data[1:])
```

```
def finddecrpt(self,data): #decrypts the AES and returns the unpadded data(CRClen + data + chksum)
  # remove ascii-armouring if present
  if data[0] == '\x00':
      data = data[1:]
  elif data[0] == '\x41':
      data = base64.decodestring(data[1:])

  iv = data[:BS]
  data = data[BS:]
  decipher = AES.new(key, AES.MODE_CBC, iv)
  data = decipher.decrypt(data)
  return self.unpad(data)
```

The data is then unpadded to provide the actual data + checksum information.

```
def unpad(self,data):
  if not data:
    return data
  data = data.rstrip('\x00')
  if data[-1] == '\x80':
    return data[:-1]
  else:
    return data
```

C. Cyclic Redundancy Check

To implement CRC we have imported the inbuilt binascii module and used the method crc-32 on the data to find the checksum. To facilitate the same crc across all python versions and platforms, we have used 0xffffffff. This gives the same crc for any python version.

Here the checksum is evaluated and then appended to the data at the end. The length of the crc is determined and added at the beginning of the data using delimiter "|" to saperate the length from actual data with crc. The final data is the length of crc + "|" +data+ crc.

```
def findcrc(self,val) : #finds crc and appends it to the end of data. Also appends length of
crc to start of data
  crcval = binascii.crc32(val) & 0xffffffff
  lencrc = len(str(crcval))
  finalval = str(lencrc) + "|" + val + str(crcval)
    #print 'finalval after crc appending: ', (finalval)
  return finalval
```

This final value is given as input to AES encryption function AES.encrypt….and is encrypted and passed to server. In the get method, the received data is decrypted and crc is retrieved. We obtain the data by saperating the last bits of length equal to the previously extracted length. crc32 is used on extracted data and if it matches the crc extracted from final value previously, we conclude that there are no errors during transmission.

```
crclen = int(decryp_data.split("|", 1)[0])
crc_cksum = decryp_data[-crclen:]
```

```
recv_data = decryp_data[len(str(crclen))+1 : -crclen]
  #compute checksum of recv_data
crc_recv_data = binascii.crc32(recv_data) & 0xffffffff
if str(crc_recv_data) == crc_cksum:
    #no error in data
  return recv_data
```

This gives an idea of whether the data has been compromised. If it is not compromised, the data is returned. If it is compromised and crc values do not match, the get call is performed again till it gets a value. This process is repeated 3 times because we have set the number of retries as 3.

D. Redundancy:

We have done two major implementations for redundancy. First one is to ensure that the same piece of data is copied onto given number of servers by replicating it and storing it. We have used the argument Level of redundancy to give the number of servers that the same data is stored in. This is given to a variable temp.LR. We used MD5 to determines which servers are going to store the data for a particular serverID. By creating a loop for temp.LR times, we can allocate the same data to temp.LR servers. This is executed both in the get and put methods.
Here size_test = Number of Servers/ temp.LR

```
 serverID = (this_key%(self.size_test))+(pending_servers*self.size_test)
```

size_test denotes number of non redundant units formed by the servers. AS denotes the total number of servers initialized in the command line. For our code to work we need the self.AS to be a multiple of level of redundancy. The pending _servers variable is used to copy the data into the given number of servers namely temp.LR. This is done to create redundant servers, where each server receives data during put operation. This creates redundant copies of the same data in different servers.

This implementation does not copy the initialized file paths and attributes on startup to the redundant servers. This means that while redundant servers will only act as mirrors to key-values (data) stored but not act as mirrors to initialized file paths for corresponding key values. Thus this setup only provides redundant data file dumps (fault) and does not provide fault masking or fault recovery.

Second implementation in redundancy is the N-modular redundancy or majority voting implementation which is built on top of the first implementation. We initialize a dictionary of the values obtained from each server response during the get call. It chooses the value that is redundant the maximum number of times by keeping a track of how many times each value has been recorded. By returning this value, the mechanism allows for fault masking when data on one or a fewer number of servers is corrupted.

```
if "value" in res:
  datainside = res["value"].data
  if datainside in dict1:
    dict1[datainside] += 1
  else:
    dict1.update({datainside : 1})
```

This value is then decrypted using AES decrypt function and the CRC is checked to determine if it is an error free data.

```
res_enc_value = datakeys[dataval.index(max(dataval))]
```

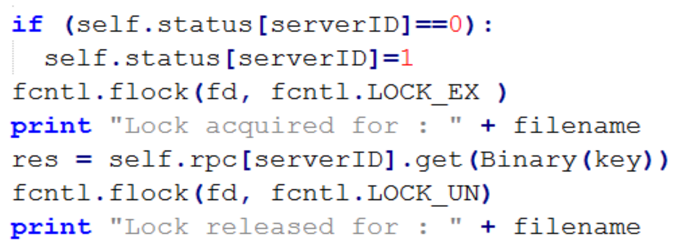
When considering multiple servers or multiple clients the major problem faced is concurrency. Since redundancy uses multiple servers. We have also developed an implementation for concurrency of multiple servers and clients for our fault tolerance implementation.

E. Concurrency:

For concurrency of multiple servers and multiple clients, we have implemented it in two parts: oncurrency in case of multiple servers, concurrency in case of multiple clients and integration of both. For this we have imported inbuilt fcntl module that executes locks (flock) on files.

For multiple servers concurrency, in the get call we have created and opened a file with name "i" added to the key to facilitate locking on atomic processes. We have also initialized a dictionary called self.server_info that accepts server name as key and provides the number of files in the server, and current status of the server. Status is initialized to'0'.Whenever data is written into the file status of all the redundant server is changed to '1', then a normal put call is executed to update the data in the redundant servers. Initially we initialize server status as 0. In case of a get call, when each server is accessed inside the while temp.LR loop we change the status in the self.server_info dictionary to 1 and once the status is changed we lock the process using flock… and the new file created with "i" appended at the end (For example: if filename is /hello.txt the new file used for locking is hello.txti). Once the value in the corresponding key-value pair is retrieved, we unlock the file using flock.
Also we implement the locking and unlocking in case of faults like server-client communication faults like xmlrpclib.fault, socket.fault. For these faults we use lock the file for a period of time and unlock it while returning none. The pending_servers loop present executes the get or put call for the next server in line.

For multiple clients concurrency, in the get call we have created and opened a file with name "i" added to the key to facilitate locking on atomic processes. We have also initialized a dictionary called self.server_info that accepts server name as key and provides the number of files in the server, and current status of the server. We have considered that if status of server is '0', then server is free and if status of server is '1', then server is busy. Initially we initialize server status as 0. In case of a get call, when each server is accessed inside the while temp.LR loop we change the status in the self.server_info dictionary to 1 and once the status is changed we lock the process using flock… and the new file created with "i" appended at the end (For example: if filename is /hello.txt the new file used for locking is hello.txti) . Once the value in the corresponding key-value pair is retrieved, we unlock the file using flock….This allows for multiple clients reading and writing at the same time into same file to be concurrent. This has not been completely implemented and tested.



```
if (self.status[serverID]==0):
  self.status[serverID]=1
fcntl.flock(fd, fcntl.LOCK_EX )
print "Lock acquired for : " + filename
res = self.rpc[serverID].get(Binary(key))
fcntl.flock(fd, fcntl.LOCK_UN)
print "Lock released for : " + filename
```

<u>TESTING AND EVALUATION MECHANISMS:</u>

The project was designed, implemented and tested in the following levels:
1.fault tolerance: MD5, AES and CRC implementations
2.Fault tolerance+Redundancy: Simple Redundancy and NMR have been implemented on top of fault tolerance code.
3. Fault tolerance + redundancy+concurrency: Concurrency is implemented for get and put methods on top of the previous code. We have done two implementations for this. One is a simple lock implementation in get and put calls. The other encompasses all of the other mechanisms mentioned in the implementation. We have implemented and testes the simple lock implementation(FRC.py). For the second implementation we have a partial working and tested code(FRC_multipleseverclient.py).

The above implementation has been tested in the following ways:

1. Basic testing: The codes for fault tolerance, redundancy and concurrency have been checked by using shell commands at each level.

```
cd  fusemount
echo “Hello world!” > fusemount/hello.txt
cat fusemount/hello.txt
rm fusemount/hello.txt
```

2. Introducing errors into the system:

a. An error is introduced with the help of test_client.py which proxies into a given server. Using get and put calls, it changes the corresponding value for the key. Corresponding output is checked for only a fault tolerance system without any redundancy or concurrency components. It invokes the retry method described in CRC but still shows an error because the actual value has been changed. It shows that there is an error in the obtained value when the FRC.py code calls the get method. Thus fault tolerance system is tested.

b. Fault tolerance, redundancy and concurrency are checked: If Level of redundancy is 1or 2, then the data is simply replicated but we do not know which one is right. The data in primary server is returned which is fraught with error. Hence none value is returned. It also has 3 retries based on number of retries, but it still shows error. However, if the error value will be corrected based on N-modular redundancy, values are compared against one another. Data can be recovered based on which one is repeated most number of times.

c. Using a random value generator we increment the checksum by 1 randomly. When testing is '1' chanceOferror can be varied from 0 to 1. This is the probability of a '1' occuring in the random variable selected from the range of 0 to 1. It is implemented such that if a '1' occurs is the random number generated, then crc is incremented by '1'. This shows that there is an error. Thus chanceOferror is varied and the respective percentage of errors is recorded.

3. Crash Failures:

When a server crashes, then the other redundant servers still work and they can be accessed to obtain the data. For this we simply kill off one of the servers in a terminal using shell commands. If server that has filesystem path is killed(crashes), error recovery is not possible although data recovery can be done.

4.Using different servers and clients:

a. time.sleep(1000) is added in the lock that performs the get function while the server in question is accessed by a different client(test_client). The server will be locked for that amount of time. Hence test_client will not be able to access the server for the given amount of time with the same key or filename.

b.When value is updated to one server time.sleep(1000) is added in the put call and other redundant servers are accessed by the test_client similar to above. The test_client will not be able to access the servers with same key because the value in the corresponding key-value pair is being updated.

EVALUATION:

A. Installation and Usage:

For level1 we have imported the inbuilt packages md5 and binascii. Cryptographic modules for python, PyCrypto is taken from https://pypi.python.org/pypi/pycrypto . Download the tar.gz file from this site address and install the package.

For level2 we need to initialize many servers. Along with the simpleht.py and fuse.py that are in the same folder there is also the fault tolerant system named FRC.py in the same folder. All the servers that are intended to be used have to be initiated in a terminal. For example:

```
$python simpleht.py --port=8000
```

8000 is the port number, it is unique for each server.
The terminal line command to execute FRC.py is as follows:

```
$python FRC.py <mountpoint> <remote hashtable> <# of Servers> <Level of Redundancy>
```

<mountpoint> is the file directory to mount the file system to(located in the same directory as the .py files)
<remote hash table> is the list of ports ex. http://localhost:8000 each listed in full and separated by a blank space
< Servers #> is the integer number of servers
<Level of Redundancy> is the integer number of redundant backups to make for each file.

The servers provided to the <# of Servers > must be a multiple of <Level of Redundancy>
modulus of <# of Servers> and <Level of Redundancy> should be 0 or FRC.py will throw usage statement and exit without executing.

For level3 we need to import fcntl inbuilt package to use the flock method in it. The flock module is used for locking and unlocking.

B. Error Evaluation:

When there is an error the system can detect and correct the error. Detection is done by the fault tolerance system, and NMR. In case of the fault tolerance system, it is fast and reliable for error detection while not taking up much memory. Error correction however takes much more memory and time. For NMR, due to redundant data, the memory constraints are increased N-fold where N is level of redundancy, latency is increased and requires more computation time to handle errors that occur. But if any of the servers were to fail the redundancy allows for other servers to provide the required data. Concurrency further increases the latency but provides a reliable data and avoids possible failures. Each section below describes the performance tests that were done and the results obtained.

C. Error Percentage:

Error probability of unmodified FUSE file system has no measures against possible errors. Our modified FUSE file system detects errors and handles them appropriately. Thus overall error probability can occur at a lower rate than an unmodified FUSE file system. By changing the 'chanceoferror' variable in our fault tolerance system we can calculate the probability of system failure. A testingScript.sh linux shell script file that does the following three commands in a loop 1000 times is executed:

```
echo "Hello world!" > fusemount/hello.txt
cat fusemount/hello.txt
rm fusemount/hello.txt
```

TestingSript.sh is executed with the following command: $ksh TestingScript.sh

Each loop consists of 14 linux calls. Hence in total the above shell script has about 14000 commands.

Chance of Bit Error	**50%**	**25%**	**10%**	**1%**	**0.1%**
FUSE modified	28.57%	3.31%	0.12%	0%	0%
FUSE unmodified	50%	25%	10%	1%	0.1%

Table1: Comparision of chance of bit errors in modified vs unmodified fuse file system

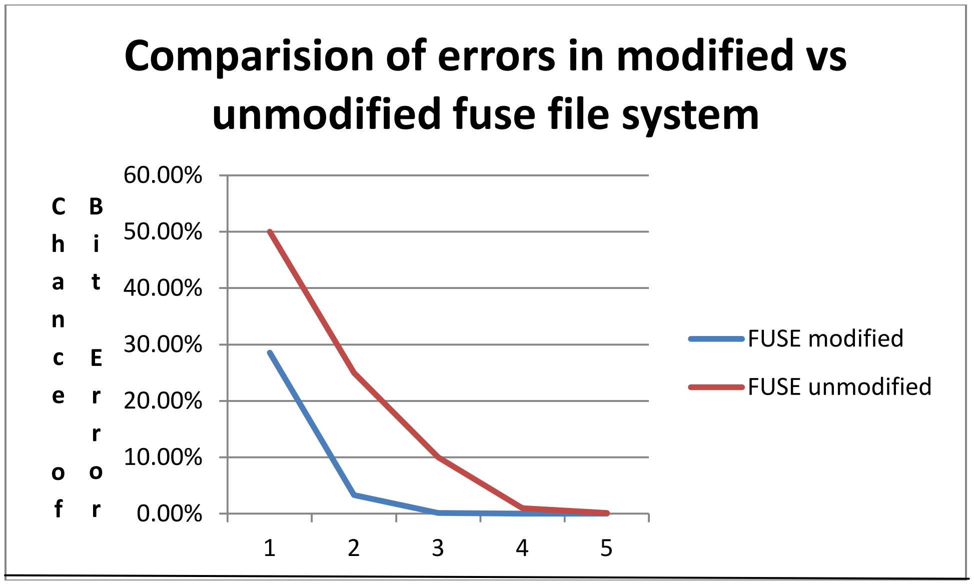


Figure 6: Comparision of chance of bit errorthroughput for the unmodified and modified FUSE file systems.

Thus our fault tolerance system handles errors better than an unmodified FUSE filesystem. Usually the chance of error is less than 1% and only increases more than 1% due to sudden increment in noise or burst traffic. So the probability of error is very low in the modified fuse file system or the fault tolerant system.

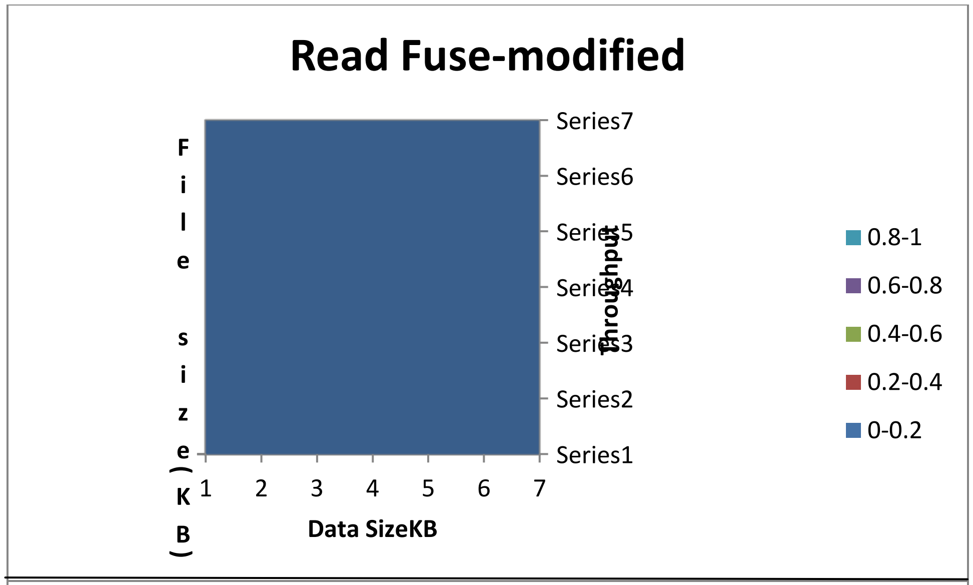
A. Overall Performance of System

By adding redundancy and concurrency, our modified FUSE file system will be able to detect and handle errors as shown above. This comes at the cost of sacrificing the speed of the overall system. For the performance analysis of the two systems and their comparisions, we use Iozone

IOzone is a filesystem benchmark tool that can measure the throughput of read and write for any directory. IOzone package was downloaded as tar.gz file from http://www.iozone.org/ . It is installed and the 'iozone' program was placed in the same directory simpleht.py, FRC.py, fuse.py. IOzone reads and writes a range of data sizes to a temporary targeted file within the FUSE file system and measures the throughput for a specified range of file sizes. The command used to obtain the benchmarks was:

```
$iozone –Rb result.wks –a –y 4 –n 4 –f ./fusemount/test1 –i 0 –i 1
```

We modify this command for the iozone to generate the throughput starting from 64KB to 1024KB file sizes inside fusemount/test. It reads and writes data of size 64KB to 1024KB in for each file size (if allowed) and writes the data to a spreadsheet called result.wks. However, for our tests FUSE has size limit of 64K, so the files sizes that were tested were 4KB, 8KB, 16KB, 32KB and 32KBwith read and write sizes of 4KB, 8KB, 16KB, 32KB and 32KBwith As for the chances of reading and writing errors to occur, the same values from the previous section were used. The following graphs show the comparison of read and write throughput in both the modified and unmodified FUSE file systems. For easier understanding of the graph, the legend on the side of each graph show the color for each throughput range.



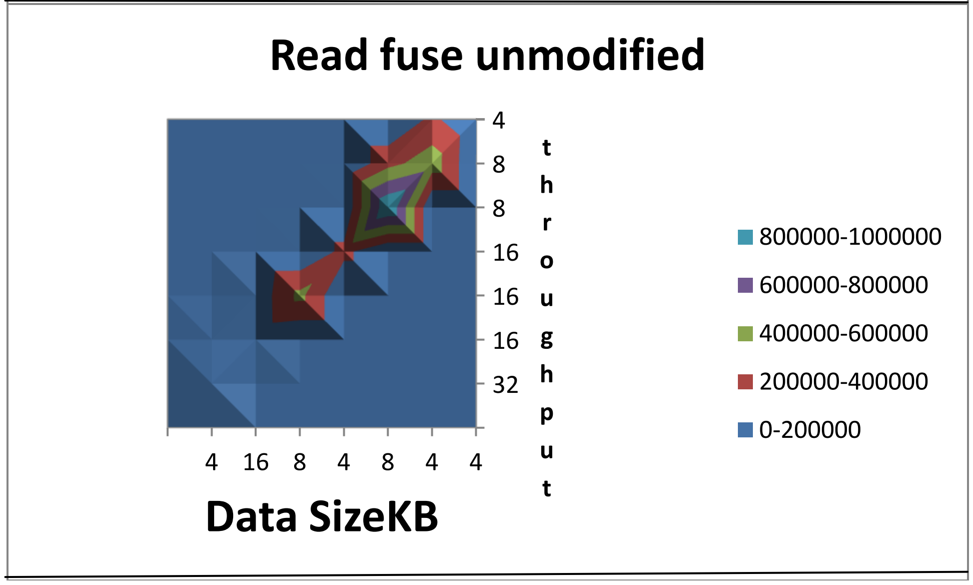
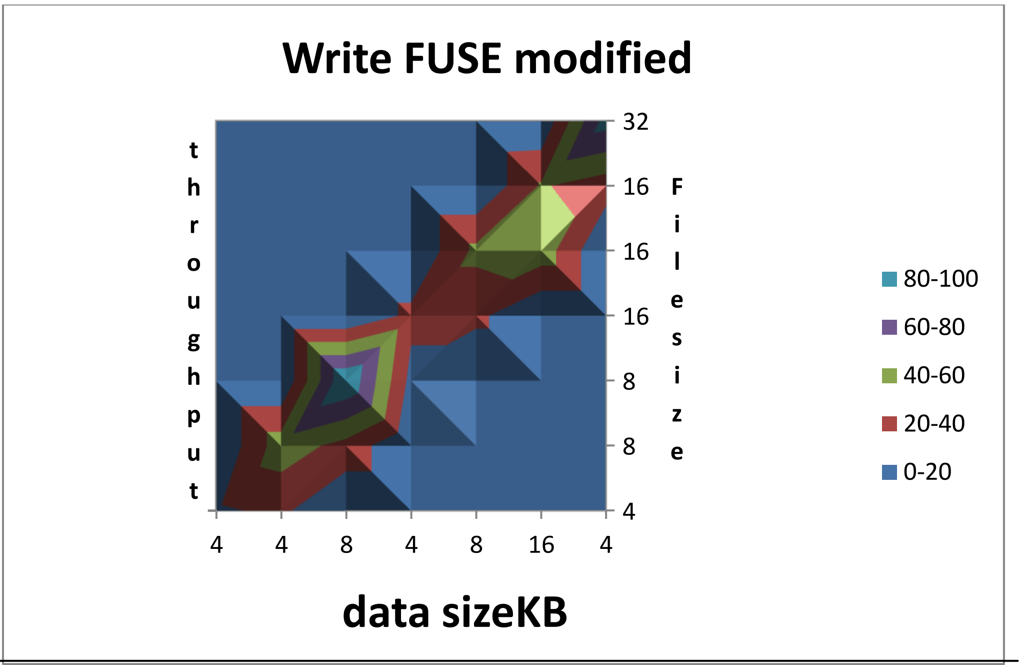


Figure 7. Reading throughput for the unmodified and modified FUSE file systems.



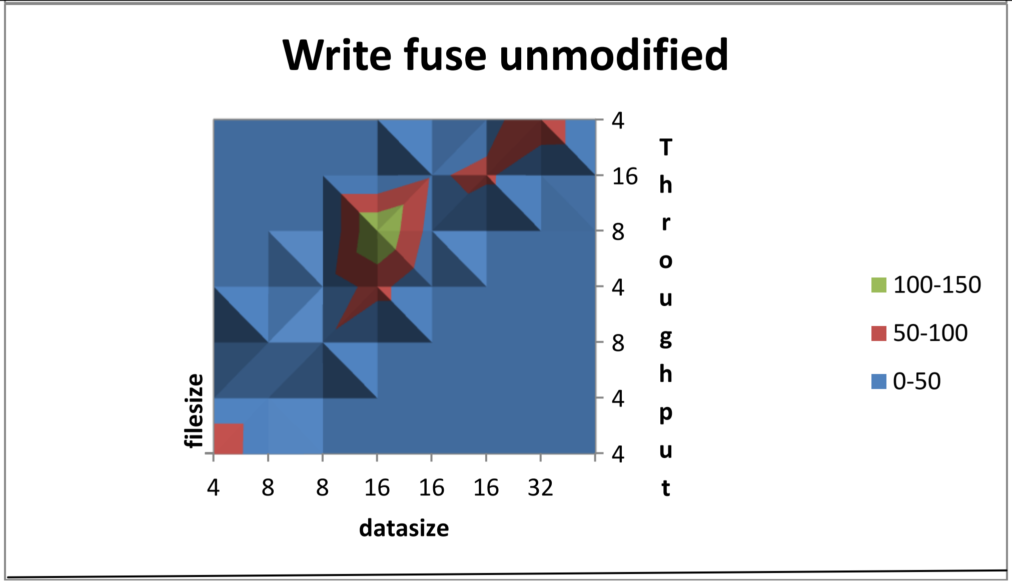
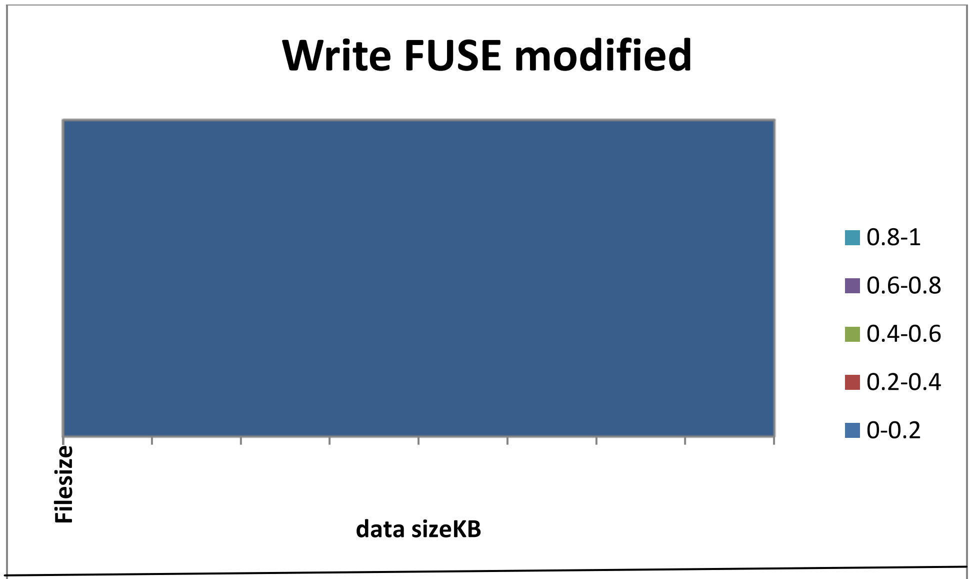


Figure 8. Writing throughput for the unmodified and modified FUSE file systems.

Since limit of the file size was 64KB, the analysis is done again with 1MB. The resulting graphs for writes are as follows. From these graphs we know that when file sizes increase, the difference in the throughputs for the modified and unmodified implementations change. Hence we can say that for a large data files using fault tolerance, redundancy and concurrency mechanisms is actually useful because there is only a small difference in the throughput



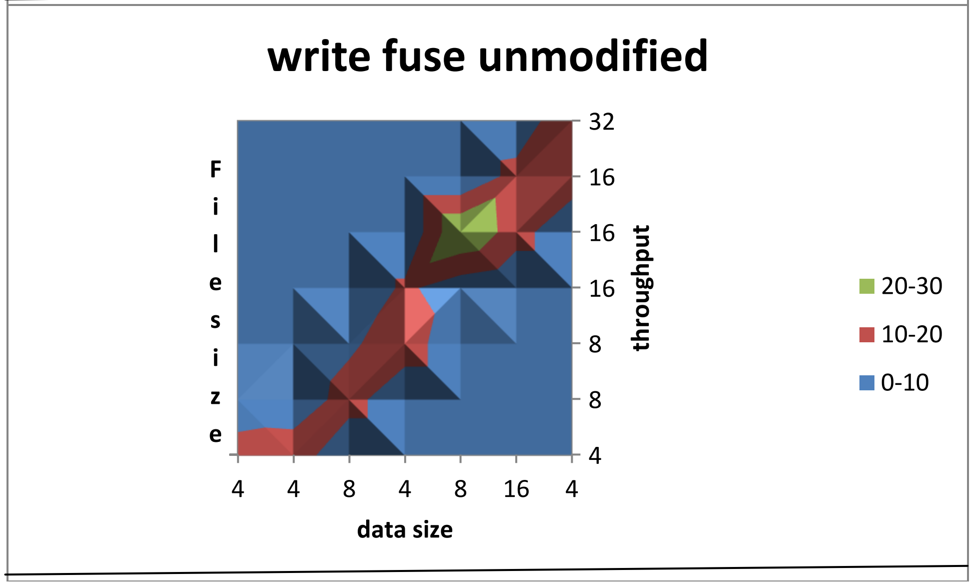


Figure 9. Writing throughput for the unmodified and modified FUSE file systems.

<u>POTENTIAL ISSUES:</u>

A. Fault tolerance:

In this project, although errors can be detected using fault tolerance mechanism(1st level), it does not correct them. In order to correct them, the redundancy mechanism(2nd mechanism) has to be used. We have chosen CRC only to detect errors. If we had used error correction codes, multiple rpc calls to the server in case of redundancy can be avoided. This can also be faster and cheaper. However, we haven’t implemented any error correction mechanism. In our project we chose CRC over Hamming code because it is had sensible decode and encode mechanisms.
AES encryption can be cracked using brute force method. The key can easily be obtained was an MD5 digest of a given password. Hence a key generation functions would serve better.
The fault tolerance is implemented for only the get and put calls, therefore for only updating or creating new file in the mounted directory.

B. Redundancy and Majority Voting:

Redundancy was not correctly implemented to attain the file paths from the fuse.py attributes necessary to maintain file system operations in case of server failure. Also the reads and writes have to be balanced on each server to ensure maximum performance. If one server is used more than the others, then it will have a shorter life span and throughput reduces due to the heavy load on the server. It also causes problems like server faults in the future. Load balancing can be done by using self.server_info[servername][1] which gives number of file systems in a server. This can be used to load balance and give the next file to the server with minimum load. However, this has not been done due to time constraints. We can also use the rsync function in the shell after the initial mount of FRC.py. This function helps maintain redundant servers which were initialized to separate mount points. For example:

```
$python FRC.py mountpoint 1 <address1> 1 1
$python FRC.py mountpoint 2 <address2> 1 1
$rsync –avh mountpoint1/* mountpoint2
```

When rsync was implemented as a subsystem function in FRC.py source filesystem became unusable.
Also in N modular redundancy scheme, if the servers are corrupted to the same value, this value will be chosen through voting scheme despite it being the corrupted data. When user corrupts the values or when voting schemeis hacked, this might be the case.

C. Concurrency:
A simple lock is placed on the file when it is being processed by one server. If the status of server changes before lock can be called, then race conditions still exist. We have based the status on the value of a dictionary, this can be corrupted or might not reflect the actual status or it may also take a very long time to update.

References:

- J. Saltzer, M. Frans Kaashoek, , “Principles of Computer System Design,” Morgan Kaufmann, 2009, 1955. *(references)*

- Dwayne C. Litzenberger. Crypto Software Package. “PyCrypto V.2.6.1. https://pypi.python.org/pypi/pycrypto

- E.Conrad. “Advanced Encryption Standard. CISSP. 2013. http://www.giac.org/cissp-papers

- Advanced Encryption Standard." *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

- Cyclic Redundancy Check." *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/Cyclic_redundancy_check

- GUIDE TO CRC ERROR DETECTION ALGORITHMS http://www.repairfaq.org/filipg/LINK/F_crc_v32.html

- MD5 Hashing." *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/MD5

- Triple ModularRedundancy." *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/Triple_modular_redundancy