EEL5737: Principles of Computer System Design

Design and Implementation of the project: Fault tolerant distributed remote file system.

Team members

Nirali Patel (Graduate student) UFID:36318593.
Shreyas Gaadikere Sreedhara (Graduate student) UFID:13144396.

INTRODUCTION

The project is an effort to build a distributed remote file system. The protocol used here to build the system is XMLRPC. Here the client is a FUSE file system that can contact the metaserver and data servers and perform the required actions. The system constitutes of a single meta server and four data servers according to the given project guidelines. The meta server is assumed to be robust i.e., never failing. The data is being distributed over all the data servers, that is a single block of data has one original copy in one of the data server and it has 2 replicas of the same data block in the next two data servers. Round robin policy will be followed for distributing the blocks of data in the data servers. The system will have a total of three copies for a given file or directory. The system also works against fault tolerance, while writing checksum is being allotted to check further if the data retrieved is corrupted or not. The data found corrupted is updated by the other neat replicas. If there are no neat replicas available either because of server crash or say data getting lost, it throws an error and the data cannot be retrieved back.

The data servers use shelve function to store the persistent storage of the data given by the user. This shelve function presents us with an opportunity to restore the data back when all the copies of the data in the different servers have been corrupted.

Here we have used a hash function to decide which dataserver is the starting point for storing the blocks of data. The blocks of data is being stored in the data server as a dictionary with key value pair. The key is the path of the block of data which is being stored in the data server. For corrupting the block of data in the file, a python program ‘corrupt.py’ has been written.

CLIENT SERVER ARCHITECTURE.

The client contacts the metaserver and data servers by using the XMLRPC protocol. This protocol requires the request going from the client to be in the Base64 format. So the client has to encode the data it is sending. For this purpose we have included the binary( ) inbuilt function. This helps to convert the string of bytes into Base64 format.

One more thing we have to take care of is the serialization of the data. Since we are using the python high level language to implement both the client and the servers we have to convert the python objects into strings of data. This

data will then be converted into the Base64 format. This can be achieved by 'pickle' python library. Thus the python objects have to be converted before converting them into Base64 format. The receiver deserializes the data using pickle.loads( ) function.

.The steps usually followed by the client while operating are as follows:

I. Send an RPC request to the metaserver for the metadata of the required file.
II. If the function is a data changing operation then send a RPC request to dataserver also. If it is a non data operation then the data from the metaserver is enough.
III. Make the required changes in the client and send back the new path back to metaserver and the new data back to the dataservers.

DATA STORAGE POLICY

The hash function gives a unique integer value for every path(file). This value is used for deciding where the data block has to be stored. The round robin storage policy is effectively implemented using the function

$$[x\%N, (x+1)\%N,(x+2)\%N,(x+3)\%N]$$

Here x is the hash value of the path of the file. N indicates the total number of data servers.

By appending one each time to the calculated hash value and then performing the modulus function we will be storing the replicas of the same data block on adjacent data servers.

ERROR DETECTING CODES

The client sends a checksum for the block of data along with the block of data it wants the data server to store. This checksum will be sent along with its corresponding data block. So when the client receives the checksum and the data, it can check whether the checksum received corresponds to the data or not. If not, it will know that either the checksum or the data has been corrupted. Then it can correct that block of data from the replicas.

To implement this functionality in the client we have used the md5 checksum module in the hashlib( ) library. This md5 function creates a 32 bit checksum for each 8 block of data. Both the 32 bit checksum and the 8 bit data will be considered as a tuple, converted into Base64 format and sent to dataserver for storing.

The checksum is calculated as

Checksum = md5(d).hexdigest( )
Data block value = (d , checksum)

RECOVERY PROCEDURE

There can be 2 ways in the failure of the dataserver. They are

I. Without failure of persistent storage
II. With failure of persistent storage.

Without failure of persistent storage.

Suppose the data server crashes and then restarts, its persistent storage is tried to open in the write only mode. If we are successful in opening the file then it means that the persistent storage is not harmed and can be used. If we are unable to open the file in the write only mode then it can mean two things. One, that we are starting the data server for the first time and the persistent storage hasn't been created yet. Two, the persistent storage has been lost.

With failure of persistent storage.

Suppose the persistent data storage fails, then we will not be able to open the file in write only mode. Then we will automatically create a new persistent storage file.

The function which we have implemented will take the list of all the data server ports which were given by the user at the starting. The list of data server ports are important to update the corrupted file from the replica data.

DATA CORRUPTION.

To corrupt the data in the persistent we have written a 'corrupt.py' python file. This function takes the argument as the path of the file to be corrupted. 'Random' python module has been used here. It randomly selects any data block and corrupts it.

The steps followed while corrupting the data is

I. Get the data from the persistent storage by using the getDataStore( ) function.
II. Pick a random block of data to corrupt.
III. The data which we have specified will be copied in the place of that random block.
IV. The new data will be stored into the persistent storage using the putDataStore( ) function.

TEST CASES

The distributed file system is tested for the following cases.

S.no	Test Cases	Output
1.	metaserver.py ran with no arguments	metaserver should not start

2.	metaserver.py ran with exactly one argument	metaserver started
3.	dataserver.py ran with less than two arguments	dataserver should not start
4.	dataserver.py ran on another port that is already used	dataserver should not start
5.	dataserver.py ran with index number that is greater the number of ports	dataserver should not start
6.	dataserver.py with exact number arguments with all port numbers	dataserver should start
7.	project.py ran with incorrect dataserver port	FUSE client should not start
8.	project.py with correct number of arguments	FUSE client should start

The above cases should work for the following file system working:

Sl no	Test cases	Code executed	Results and remarks
1.	Writing in a file “one.txt”	echo “abcdef” > one.txt	Text file was created with data “abcdef”, output verified through cat command and GUI.
2.	Copying file “one.txt”	cp one.txt two.txt	The contents of one.txt is copied to new file two.txt, output verified through cat command and GUI.
3.	Renaming file “two.txt” to “three.txt”	mv two.txt three.txt	The text file “two.txt” is renamed to “three.txt”
4.	Reading the renamed file	cat three.txt	output verified through cat and GUI.
5.	Append data to file “one.txt”	echo “ghijklm” >> one.txt	Data in the text file is appended and it now reads “abcdefghijklm”, output verified through cat command and GUI.
6.	Checking hardlink before creating a directory	ll	Number of hardlinks is showed as 2.
7.	Create a folder “folder_one”	mkdir folder_one	Creates a folder “folder_one” in the present path, output verified by GUI, ls and ll command.
8.	Checking the hard links of the parent directory	cd .. ll	The number of hardlinks will have increased by one after the folder

			creation.
9.	Move file “one.txt” to folder “folder_one”	mv one.txt folder_one/one.txt	Moves the file one.txt inside the folder_one.
10.	Check files inside “folder_one”	ls ll	Result verified through GUI, ls and ll command.
11.	Read “one.txt” from “folder_one”	cat folder_one/one.txt	Reads the file “one.txt” inside the folder “folder_one”.
12.	Create symlink	ln -s two.txt folder_one/z.txt	Creates a symlink for file “two.txt” inside the folder “folder_one” with file “z.txt”.
13.	Create a folder “folder_two” inside the folder “folder_one”	mkdir folder_one/folder_two	Creates a folder inside another folder. Output verified through GUI, ls and ll command.
14.	Check hardlink of the parent	cd .. ll	The number of hardlinks will be increased by one after the folder is created.
15.	Check harlink of the newly created file	cd folder_two ll	The number of hardlinks of the newly created file will be 2.
16.	Remove folder_one	rmdir folder_one	Error is thrown that the directory is not empty.
17.	Check number of hardlinks after the remove.	ll	There will be no change in number of hardlinks after this command has been executed.
18.	Recursive remove the folder_one	rmdir -r folder_one	Deletes the whole folder_one and its contents.
19.	Check number of hardlinks after the remove	ll	The number of hardlinks will be decreased.
20.	Copy folder_one to folder_one_copy	cp folder_one folder_one_copy	Copies the contents of folder_one to folder_one_copy
21.	Remove the directory “folder_one_copy”	rmdir folder_one_copy	Throws an error that the directory is not empty.
22.	Recursive delete the directory “folder_one_copy”	rmdir -r folder_one_copy	Deletes the folder “folder_one_copy” and everything inside it. Verified the output through GUI, ls and ll command.

23.	Make a folder and move “folder_one” to it	mkdir folder_four mv folder_one folder_four	Moves the whole folder “folder_one” to folder_four.
24.	read one.txt from the folder “folder_one” which is inside “folder_four”	cd folder_four/folder_one cat one.txt	Reads the file one.txt from the given folder. Output seen as “abcdefghijklm”.
25.	Remove the file “three.txt”	rm three.txt	Removes the file “three.txt”, verified the result using GUI, ls and ll commands.
26.	Create a directory with the name as “st_mode”	mkdir st_mode	Created the directory “st_mode”, verified using GUI, ls and ll command.
27.	Truncate 5 bits of data from the data of the text file which has more than 5 bits	truncate -s 5 c.txt	The data will be truncated leaving only 5 bits from the original data.
28.			

Suppose there is a text file “fileone.txt” which contains the data “abcdefghijklm” inside it. We assume that the data in the file will be divided into blocks of 8 characters, that is the data in the file will be divided as “abcdefgh” and “ijklm” and the first block will be stored starting from the dataserver 0 according to the output of the hash function. Also this file will have the replicas in next 2 servers, that is in dataserver 1 and dataserver 2.

The next block of data “ijklm” will start to store from the next server. That is from the dataserver 1 and its replicas will be stored in dataserver 2 and dataserver 3. This pattern continues for the number of servers we have and then the data will start storing from the dataserver 0 again in a round robin fashion as mentioned above.

Cases for testing various scenarios of data server being down, the data in it being corrupted and data servers having neat copy of data are shown below. Here we are explaining the example of a block data that has the original copy in the dataserver 1 and two replicas in dataserver 2 and dataserver 3.

Sl no.	Test case	Output
1.	One data server is crashed and other two data servers are working fine.	The data will be retrieved and the warning will be displayed that one data server is not working. When the crashed data server restarts, it will get the data from the

		persistent storage.
2.	Two servers are crashed and one data server is working fine.	The data will be retrieved from the clean copy and once the two crashed servers restart they will recover the data from persistent storage.
3.	All three data servers are crashed.	The data will not be retrieved and when the data servers restart they will get the data back from the persistent storage.
4.	One data server is corrupted and other two data servers are working fine.	The data will be retrieved from the clean copy of data and the corrupted data server will be repaired.
5.	Two data servers are corrupted and one data server is clean.	The data will be retrieved from the clean replica and also the corrupted file will be repaired.
6.	All three data servers are corrupted.	The data can't be retrieved and the data is lost forever.
7.	One data server is corrupted, one data is crashed and the other data server is correct.	The data will be retrieved from the clean copy of the replica, the corrupted copy of data will be repaired from the clean copy and the crashed data server gets back the data from its persistent data storage once it restarts.

CONCLUSION.

The data in the data servers have been successfully implemented to be fault tolerant. The file system is a distributed remote file system which communicates with the different servers using the XMLRPC protocol. Here the metaserver has been assumed to be non failing, which is not true in the real scenarios. As a continuation of this project, meta servers that are fault tolerant can be designed.

DIVISION OF LABOR.

Both of us have worked on the code in our individual capacity and came up with different implementations of the modules in the code. After this was done, we integrated both our code and tested it afterwards. The best possible implementation was followed for the final project.

PYTHON CODE

```
#!/usr/bin/env python

from __future__ import print_function, absolute_import, division
import logging
import xmlrpclib,pickle,hashlib,socket
from collections import defaultdict
from errno import ENOENT
from errno import ENOTEMPTY
from stat import S_IFDIR, S_IFLNK, S_IFREG
from sys import argv, exit
from time import time,sleep

from fuse import FUSE, FuseOSError, Operations, LoggingMixIn

if not hasattr(__builtins__, 'bytes'):
    bytes = str

class Memory(LoggingMixIn, Operations):
    'Example memory filesystem. Supports multilevel of files.'

    def __init__(self,metaserverport,dataserverport):
        #self.files = {}
            self.size_block=8
            self.data = defaultdict(list)
        self.fd = 0
            self.metaserverport=metaserverport
            self.dataserverport=dataserverport
            print(self.metaserverport)
            print(self.dataserverport)
            self.metaserveradd=xmlrpclib.ServerProxy('http://localhost:' + str(self.metaserverport) +
'/')
            self.dataserveradd=[]
            for i in range (0,len(self.dataserverport)):
                    self.dataserveradd.append(xmlrpclib.ServerProxy('http://localhost:' +
str(self.dataserverport[i]) + '/'))
            print (self.metaserveradd)
            print (self.dataserveradd)
```

```
        now = time()
            self.metaserveradd.put('/',pickle.dumps(dict(st_mode=(S_IFDIR | 0o755),
st_ctime=now,st_mtime=now, st_atime=now, st_nlink=2, files = [])))
            print(pickle.loads(self.metaserveradd.get('/')))

    def chmod(self, path, mode):
            metadata = pickle.loads(self.metaserveradd.get(path))
            metadata['st_mode'] &= 0o770000
        metadata['st_mode'] |= mode
            self.metaserveradd.put(path,pickle.dumps(metadata))
        return 0

    def chown(self, path, uid, gid):
            metadata = pickle.loads(self.metaserveradd.get(path))
        metadata['st_uid'] = uid
        metadata['st_gid'] = gid
            self.metaserveradd.put(path,pickle.dumps(metadata))


    def create(self, path, mode):
            metadata = dict(st_mode=(S_IFREG | mode), st_nlink=1,st_size=0, st_ctime=time(),
st_mtime=time(),st_atime=time(),files=[],blocks=[])
            self.metaserveradd.put(path,pickle.dumps(metadata))
            parent, child=self.dividepath(path)
            metadata = pickle.loads(self.metaserveradd.get(parent))
            metadata['files'].append(child)
            self.metaserveradd.put(parent,pickle.dumps(metadata))
            self.fd += 1
        return self.fd

    def dividepath(self, path):
            child = path[path.rfind('/')+1:]

            parent = path[:path.rfind('/')]
            if parent == '':
                    parent='/'
            return parent,child

    def getattr(self, path, fh=None):
```

```
        if self.metaserveradd.get(path) == -1:
            raise FuseOSError(ENOENT)
            return pickle.loads(self.metaserveradd.get(path))

    def getxattr(self, path, name, position=0):
            if self.metaserveradd.get(path) == -1:
            return ''
            metadata = pickle.loads(self.metaserveradd.get(path))
            attrs = metadata.get('attrs', {})
            try:
            return attrs[name]
        except KeyError:
            return ''                       # Should return ENOATTR

    def listxattr(self, path):
            if self.metaserveradd.get(path) == -1:
            return ''                       # Should return ENOATTR
        metadata = pickle.loads(self.metaserveradd.get(path))
        attrs = metadata.get('attrs', {})
        return attrs.keys()

    def mkdir(self, path, mode):
        metadata = dict(st_mode=(S_IFDIR | mode), st_nlink=2,st_size=0, st_ctime=time(),
st_mtime=time(),st_atime=time(),files=[])
            self.metaserveradd.put(path,pickle.dumps(metadata))
            parent,child=self.dividepath(path)
            metadata=pickle.loads(self.metaserveradd.get(parent))
            metadata['st_nlink'] += 1
            metadata['files'].append(child)
            self.metaserveradd.put(parent,pickle.dumps(metadata))

    def open(self, path, flags):
        self.fd += 1                                                    # increment
the self.fd
        return self.fd


    def read(self, path, size, offset, fh):
            metadata = pickle.loads(self.metaserveradd.get(path))
```

```
        data = self.readdata(path,metadata['blocks'])
      return data[offset:offset+size]

   def readdata(self,path,blocks):
      output = ''
         #storage1 = storage2 = storage3 = 0
      for i in range(0,len(blocks)):
         try:
            storage1 = self.dataserveradd[blocks[i]].get(path + str(i))

         except socket.error:
            print('WARNING READ - link to Data Server at port ' +
str(self.dataserverport[blocks[i]]) + ' is lost.')
            storage1 = -1
         try:
            storage2 = self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].get(path +
str(i))

         except socket.error:
            print('WARNING READ - link to Data Server at port ' +
str(self.dataserverport[(blocks[i] + 1) % len(self.dataserverport)]) + ' is lost.')
            storage2 = -1
            try:
            storage3 = self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].get(path +
str(i))
                  print("localblock 3" + str(storage3))
         except socket.error:
            print('WARNING READ - link to Data Server at port ' +
str(self.dataserverport[(blocks[i] + 2) % len(self.dataserverport)]) + ' is lost.')
            storage3 = -1

         # All three copies are lost
         if (storage1 == -1) and (storage2 == -1) and (storage3 == -1):
            output = 'ERROR - Three adjacent servers lost their respective persistance storage
blocks. Cannot RECOVER.'
            break
            # All three copies are present
         elif (storage1 != -1) and (storage2 != -1) and (storage3 != -1):
            # verifying Checksum
```

```
            if (storage1[len(storage1) - 32:] == hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()) and (storage2[len(storage2) - 32:] == hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()) and (storage3[len(storage3) - 32:] == hashlib.md5(storage2[:len(storage3) -
32]).hexdigest()):
                output += storage1[:len(storage1) - 32]

            elif (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()) and (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()) and (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
                output = 'ERROR - All the copies are corrupted.'
                for i in range(0,3):
                   sleep(2)
                break
            elif (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()) and (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()):
                print('First copy and Second copy is corrupted. Recovering....')
                # Recovering First copy and Second copy
                try:
                        self.dataserveradd[blocks[i]].put(path + str(i),storage3)
                except socket.error:
                   print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                    try:
                        self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage3)
                except socket.error:
                   print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                output += storage3[:len(storage3) - 32]
                    print("output" + str(output))

                elif (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()) and (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
                print('First copy and Third copy is corrupted. Recovering....')
                # Recovering First copy and Third copy
                try:
```

```
                    self.dataserveradd[blocks[i]].put(path + str(i),storage2)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                    try:
                        self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage2)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage2[:len(storage2) - 32]
                print("output" + str(output))

            elif (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()) and (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
            print('Second copy and Third copy is corrupted. Recovering....')
            # Recovering Second copy and Third copy
            try:
                    self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage1)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                    try:
                        self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage2)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage1[:len(storage1) - 32]
                print("output" + str(output))

            elif (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()):
            print('First copy is corrupted. Recovering....')
            # Recovering First copy
            try:
                    self.dataserveradd[blocks[i]].put(path + str(i),storage3)
```

```
                except socket.error:
                    print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                output += storage3[:len(storage3) - 32]

                    elif (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()):
                print('Second copy is corrupted. Recovering....')
                # Recovering Second copy
                try:
                            self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage1)
                except socket.error:
                    print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                output += storage1[:len(storage1) - 32]

                    elif (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
                print('Third copy is corrupted. Recovering....')
                # Recovering Third copy
                    try:
                            self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage2)
                except socket.error:
                    print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
                output += storage1[:len(storage1) - 32]


            else:
                output = 'ERROR - Unhandled Exception while verifyinging checksum'
                break
            # First copy is lost
            elif (storage2 != -1) and (storage3 != -1):
            # verifying Checksum
            if (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()) and (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
```

```
            output = 'ERROR - First copy is absent and Second and Third copy is corrupted.'
            for i in range(0,3):
               print(output)
               sleep(2)
            break
               # Recovering First and Second from Third copy
               elif (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()):
                  print('First copy is absent and Second copy is corrupted. Recovering....')
                  try:
               self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage3)
            except socket.error:
               print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                  try:
               self.dataserveradd[blocks[i]].put(path + str(i),storage3)
            except socket.error:
               print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                  output += storage3[:len(storage3) - 32]
                  print("output" + str(output))
               # Recovering First and Third from Second copy
               elif (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
                  print('First copy is absent and Third copy is corrupted. Recovering....')
                  try:
               self.dataserveradd[blocks[i]].put(path + str(i),storage2)
            except socket.error:
               print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                  try:
               self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage2)
            except socket.error:
               print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage2[:len(storage2) - 32]
                  print("output" + str(output))
```

```
                else:
                    try:
                self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage2)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage2[:len(storage2) - 32]
                print("output" + str(output))

        # Second copy lost
        elif (storage1 != -1) and (storage3 != -1):
            # verifying Checksum
            if (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()) and (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
            output = 'ERROR - Second copy is absent and First and Third copy is corrupted.'
            for i in range(0,3):
                print(output)
                sleep(2)
            break
                # Recovering First and Second from Third copy
                elif (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()):
                    print('Second copy is absent and First copy is corrupted. Recovering....')
                    try:
                self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage3)
            except socket.error:
                print('WARNING- Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                    try:
                self.dataserveradd[blocks[i]].put(path + str(i),storage3)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                    output += storage3[:len(storage3) - 32]
                    print("output" + str(output))
                # Recovering Second and Third from First copy
```

```
                elif (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
                    print('Second copy is absent and Third copy is corrupted. Recovering....')
                    try:
                self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage1)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                    try:
                self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage1)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage1[:len(storage1) - 32]
                print("output" + str(output))
            else:
                try:
                self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage1)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage1[:len(storage1) - 32]
                print("output" + str(output))
        # Third copy lost
        elif (storage1 != -1) and (storage2 != -1):
        # verifying Checksum
        if (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()) and (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()):
            output = 'ERROR - Third copy is absent and First and Second copy is corrupted.'
            for i in range(0,3):
                print(output)
                sleep(2)
            break
                # Recovering First and Third from Second copy
```

```
                elif (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()):
                    print('Third copy is absent and First copy is corrupted. Recovering....')
                    try:
                self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage2)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
                    try:
                self.dataserveradd[blocks[i]].put(path + str(i),storage2)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                    output += storage2[:len(storage2) - 32]
                    print("output" + str(output))
                # Recovering Second and Third from First copy
                elif (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()):
                    print('Third copy is absent and Second copy is corrupted. Recovering....')
                    try:
                self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage1)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                    try:
                self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage1)
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage1[:len(storage1) - 32]
                    print("output" + str(output))
                else:
                    try:
                self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage1)
            except socket.error:
```

```
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')
            output += storage1[:len(storage1) - 32]
                print("output" + str(output))
        # First and Second copies are lost
        elif (storage3 != -1):
        # Third copy is corrupted
            if (storage3[len(storage3) - 32:] != hashlib.md5(storage3[:len(storage3) -
32]).hexdigest()):
                output = 'ERROR - First and Second copy is absent and Third copy is
corrupted.'
            for i in range(0,3):
                print(output)
                sleep(3)
            break
        # Recovering First and Second Copy
        try:
            self.dataserveradd[blocks[i]].put(path + str(i),storage3)
        except socket.error:
            print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                try:
            self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage3)
        except socket.error:
            print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')

        output += storage3[:len(storage3) - 32]
            print("output" + str(output))
        # First and Third copies are lost
        elif (storage2 != -1):
        # Second copy is corrupted
            if (storage2[len(storage2) - 32:] != hashlib.md5(storage2[:len(storage2) -
32]).hexdigest()):
                output = 'ERROR - First and Third copy is absent and Second copy is
corrupted.'
            for i in range(0,3):
                print(output)
```

```
                sleep(3)
            break
        # Recovering First and Third Copy
        try:
            self.dataserveradd[blocks[i]].put(path + str(i),storage2)
        except socket.error:
            print('WARNING - Data server at port ' + str(self.dataserverport[blocks[i]]) + ' is
absent while data correction.')
                try:
            self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage2)
        except socket.error:
            print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')

        output += storage2[:len(storage2) - 32]
            print("output" + str(output))

         # Second and Third copies are lost
        elif (storage1 != -1):
        # First copy is corrupted
            if (storage1[len(storage1) - 32:] != hashlib.md5(storage1[:len(storage1) -
32]).hexdigest()):
                output = 'ERROR - Second and Third copy is absent and First copy is
corrupted.'
            for i in range(0,3):
                print(output)
                sleep(3)
            break
        # Recovering Second and Third Copy
        try:
                self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),storage1)
        except socket.error:
            print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 1) %
len(self.dataserverport)]) + ' is absent while data correction.')
                try:
            self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),storage1)
```

```
            except socket.error:
                print('WARNING - Data server at port ' + str(self.dataserverport[(blocks[i] + 2) %
len(self.dataserverport)]) + ' is absent while data correction.')

            output += storage1[:len(storage1) - 32]
                print("output" + str(output))

            else:
            output = 'ERROR - Unhandled Exception'
            break
        return output




    def readdir(self, path, fh):
            metadata=pickle.loads(self.metaserveradd.get(path))
            # load metadata from the metaserver

            print (metadata)
        return ['.', '..'] + [x for x in metadata['files']]

    def readlink(self, path):
            p=pickle.loads(self.metaserveradd.get(path))
            data=self.readdata(path,p['blocks'])
            return data[offset:offset+size]
        return data

    def removexattr(self, path, name):
            if self.metaserveradd.get(path) == -1:
            return "   # Should return ENOATTR
        metadata = pickle.loads(self.metaserveradd.get(path))
        attrs = metadata.get('attrs', {})

        try:
            del attrs[name]
            metadata.set('attrs', attrs)
            self.metaserveradd.put(path,pickle.dumps(metadata))
        except KeyError:
```

```
        pass        # Should return ENOATTR

    def rename(self, old, new):
        metadataold=pickle.loads(self.metaserveradd.get(old))
        op,oc=self.dividepath(old)
        np,nc=self.dividepath(new)
        if metadataold['st_mode'] & 0770000 == S_IFDIR:
                #self.mkdir(new,S_IFDIR)
                self.metaserveradd.put(new,pickle.dumps(dict(st_mode=(S_IFDIR | 0o777),
st_ctime=time(),st_mtime=time(), st_atime=time(), st_nlink=metadataold['st_nlink'], st_size=0,
files = [])))
                metadata = pickle.loads(self.metaserveradd.get(np))
                metadata['files'].append(nc)
        metadata['st_nlink'] += 1
        self.metaserveradd.put(np,pickle.dumps(metadata))
                for files in metadataold['files']:
                        self.rename(old + '/' + files, new + '/' + files)
                self.rmdir(old)
      else:
                #self.create(new,S_IFREG)

                self.metaserveradd.put(new,pickle.dumps(dict(st_mode=(S_IFREG | 0o777),
st_nlink=1, st_size=0, st_ctime=time(), st_mtime=time(), st_atime=time(), files = [], blocks =
[])))
        metadata = pickle.loads(self.metaserveradd.get(np))
        metadata['files'].append(nc)
        self.metaserveradd.put(np,pickle.dumps(metadata))
        self.fd += 1
                metadatanew = metadataold
        Data = self.readdata(old,metadataold['blocks'])
        blockdata = []
        blocks = []
        pointer = hash(new)
        count = 1
        for i in range(0,len(Data),self.size_block):
                        blockdata.append(Data[i : i + self.size_block])
                        blocks.append((pointer + count - 1) % len(self.dataserverport))
                        count += 1;
```

```
            self.writedata(new,blockdata,blocks)
            metadatanew['st_size'] = len(Data)
            metadatanew['blocks'] = blocks
                    self.metaserveradd.put(new,pickle.dumps(metadatanew))
                    metadata_new = pickle.loads(self.metaserveradd.get(old))
            if (metadata_new['st_mode'] & S_IFREG) == S_IFREG:
                            blocks = metadata_new['blocks']
                    metadata_new = pickle.loads(self.metaserveradd.get(op))
                    metadata_new['files'].remove(oc)
                    self.metaserveradd.put(op,pickle.dumps(metadata_new))
                    self.metaserveradd.pop_entry(old)

    def removedata(self,path,blocks):
        for i in range(0,len(blocks)):
            link = False
            while link is False:
                try:
                    link = self.dataserveradd[blocks[i]].pop_entry(path + str(i))
                except socket.error:
                    print('WARNING - Data Removal: Data server at port ' +
str(self.dataserverport[blocks[i]]) + ' is absent.')
                    print('Trying to re-connect.....')
                    sleep(5)
            link = False
            while link is False:
                try:
                    link = self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].pop_entry(path
+ str(i))
                except socket.error:
                    print('WARNING - Data Removal: Data server at port ' +
str(self.dataserverport[(blocks[i] + 1) % len(self.dataserverport)]) + ' is absent.')
                    print('Trying to re-connect.....')
                    sleep(5)
            link = False
            while link is False:
                try:
                    link = self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].pop_entry(path
+ str(i))
                except socket.error:
```

```
                print('WARNING - Data Removal: Data server at port ' +
str(self.dataserverport[(blocks[i] + 2) % len(self.dataserverport)]) + ' is absent.')
                print('Trying to re-connect.....')
                sleep(5)

    def rmdir(self, path):
            parent,child=self.dividepath(path)
            meta_data_path=pickle.loads(self.metaserveradd.get(path))
            metadata=pickle.loads(self.metaserveradd.get(parent))
            if meta_data_path['st_mode'] & 0770000 == S_IFDIR:
                    if not meta_data_path['files'] == []:
                            raise FuseOSError(ENOTEMPTY)
# raise a FUSE error.
            metadata['files'].remove(child)
            metadata['st_nlink'] -=1                                                    #
decrement the st_nlink
            self.metaserveradd.put(parent,pickle.dumps(metadata))
            self.metaserveradd.pop_entry(path)


    def setxattr(self, path, name, value, options, position=0):
        # Ignore options
            if self.metaserveradd.get(path) == -1:
            return ''   # Should return ENOATTR
        metadata = pickle.loads(self.metaserveradd.get(path))
        attrs = metadata.setdefault('attrs', {})
        attrs[name] = value
        metadata.set('attrs', attrs)
        self.metaserveradd.put(path,pickle.dumps(metadata))


    def statfs(self, path):
        return dict(f_bsize=self.size_block, f_blocks=4096, f_bavail=2048)

    def symlink(self, target, source):
        self.metaserveradd.put(target,pickle.dumps(dict(st_mode=(S_IFLNK | 0o777), st_nlink=1,
st_size=len(source), blocks = [], files=[])))
            metaData = pickle.loads(self.metaserveradd.get(target))
            x = hash(target)
```

```
    blocks_newdata = []
    blocks = []
    j = 1
    for i in range(0,len(source),self.size_block):
        blocks_newdata.append(source[i : i + self.size_block])
        blocks.append((x + j - 1) % len(self.dataserverport))
        j += 1;
    self.writedata(target,blocks_newdata,blocks)
        metaData['blocks'] = blocks
        self.metaserveradd.put(target,pickle.dumps(metaData))
        parentpath,childpath = self.splitpath(target)
        meta_data = pickle.loads(self.metaserveradd.get(parentpath))
    meta_data['files'].append(childpath)
    self.metaserveradd.put(parentpath,pickle.dumps(meta_data))


def truncate(self, path, length, fh=None):
        h = hash(path)
        blocks_newdata = []
    blocks = []
        metadata = pickle.loads(self.metaserveradd.get(path))
    data1 = self.readdata(path,metadata['blocks'])
    data2 = data1[:length]
    n = 1
    for i in range(0,len(data2),self.size_block):
        blocks_newdata.append(data2[i : i + self.size_block])
        blocks.append((h + n - 1) % len(self.dataserverport))
        n += 1;
    self.writedata(path,blocks_newdata,blocks)
    #for i in range (0,len(secdata)):
        #         self.dataserveradd[blocks[i]].put(path+str(i),secdata[i])
        metadata['st_size'] = length
        metadata['blocks'] = blocks
        self.metaserveradd.put(path,pickle.dumps(metadata))

def unlink(self, path):
        parent,child=self.dividepath(path)
        metadata=pickle.loads(self.metaserveradd.get(path))
```

```
        self.metaserveradd.put(parent,pickle.dumps(metadata))
    if (metadata['st_mode'] & S_IFREG) == S_IFREG:
            self.removedata(path,metadata['blocks'])
            parent,child=self.dividepath(path)
        meta_data = pickle.loads(self.metaserveradd.get(parent))
        meta_data['files'].remove(child)
        self.metaserveradd.put(parent,pickle.dumps(metadata))
        self.metaserveradd.pop_entry(path)


def utimens(self, path, times=None):
    now = time()
    atime, mtime = times if times else (now, now)
        metadata=pickle.loads(self.metaserveradd.get(path))
    metadata['st_atime'] = atime
    metadata['st_mtime'] = mtime
        self.metaserveradd.put(path,pickle.dumps(metadata))


def write(self, path, data, offset, fh):
        h = hash(path)
        blocks_newdata = []
    blocks = []
    metadata = pickle.loads(self.metaserveradd.get(path))
    if len(metadata['blocks']) == 0:
        oldData = ''
    else:
        oldData = self.readdata(path,metadata['blocks'])
    newdata = oldData[:offset].ljust(offset,'\x00') + data + oldData[offset + len(data):]
    n = 1
    for a in range(0,len(newdata), self.size_block):
        blocks_newdata.append(newdata[a : a + self.size_block])
        blocks.append((h + n - 1) % len(self.dataserverport))
        n += 1;
    self.writedata(path,blocks_newdata,blocks)
        metadata['st_size'] = len(newdata)
    metadata['blocks'] = blocks
    self.metaserveradd.put(path,pickle.dumps(metadata))
    return len(data)
```

```
    def writedata(self,path,blocks_newdata,blocks):
            for i in range(0,len(blocks_newdata)):
            # Storing First replica of block
            link = False
            while link is False:
                try:
                    # Storing block
                    link = self.dataserveradd[blocks[i]].put(path + str(i),blocks_newdata[i] +
hashlib.md5(blocks_newdata[i]).hexdigest())
                        print("in write" + " newdata blocks" + str(blocks_newdata[i]) + "first copy" +
str(self.dataserverport[blocks[i]]))
                except socket.error:
                    print('WRITE - link to Data Server at port ' + str(self.dataserverport[blocks[i]]) + ' is
lost.')
                    sleep(5)

            # Store Second replica of block
            link = False
            while link is False:
                try:
                    # Storing block
                    link = self.dataserveradd[(blocks[i] + 1) % len(self.dataserverport)].put(path +
str(i),blocks_newdata[i] + hashlib.md5(blocks_newdata[i]).hexdigest())
                        print("in write" + " newdata blocks" + str(blocks_newdata[i]) + "second copy"
+ str(self.dataserverport[(blocks[i] + 1) % len(self.dataserverport)]))
                except socket.error:
                    print('WRITE - link to Data Server at port ' + str(self.dataserverport[(blocks[i] + 1)
% len(self.dataserverport)]) + ' is lost.')
                    sleep(5)
                # Store Third replica of block
            link = False
            while link is False:
                try:
                    # Storing block
                    link = self.dataserveradd[(blocks[i] + 2) % len(self.dataserverport)].put(path +
str(i),blocks_newdata[i] + hashlib.md5(blocks_newdata[i]).hexdigest())
                        print("in write" + " newdata blocks" + str(blocks_newdata[i]) + "third copy" +
str(self.dataserverport[(blocks[i] + 2) % len(self.dataserverport)]))
```

```
            except socket.error:
                print('WRITE - link to Data Server at port ' + str(self.dataserverport[(blocks[i] + 2)
% len(self.dataserverport)]) + ' is lost.')
                sleep(5)
            link = False
        while link is False:
            try:

                link = self.dataserveradd[(blocks[i] + 3) % len(self.dataserverport)].get(path + str(i))

            except socket.error:
                print('WRITE - link to Data Server at port ' + str(self.dataserverport[(blocks[i] + 3)
% len(self.dataserverport)]) + ' is lost.')
                sleep(5)

if __name__ == '__main__':
    if len(argv) < 6:
        print('usage: %s <mountpoint> <metaserver port> <dataserver port>' % argv[0])
        exit(1)
    metaserverport=int(argv[2])
    dataserverport=[]
    for i in range(3,len(argv)):
            dataserverport.append(int(argv[i]))

    logging.basicConfig(level=logging.DEBUG)
    fuse = FUSE(Memory(metaserverport,dataserverport), argv[1], foreground=True, debug=
True)
```