Principles of Computer System Design

Final Project

Dongyu Xu 64083137, Ying Wang 13917195

I. Background

In practice, it is not common that things will break and designed functions will fail, it is just a matter of time. Most people who use computers regularly have encountered a failure, either in the form of a software crash, disk failure, power loss, or bus error. In some instances these failures are no more than annoyances; in others they result in significant losses. [1] Thus, this leads us to design an ideal system that would be reliable and never fail. Though this seems impossible in real life, we could implement a system that rather than completely failing, can still continue to operate to make sure the system is safe. That is, we could design a system that can tolerate failure.[2]

There are many reasons for systems to fail. It is possible that the system may have been specified erroneously, causing an incorrect design. In addition, the system may contain a fault that shows up only under certain conditions that haven't been tested yet. Environment problem is also a reason. Last but not the least, aging components may stop working suddenly. Visualizing and understanding random failures caused by aging hardware is easy. However, it is much more difficult to understand how failures happen due to design flaws, poor testing, operator errors and incorrect specifications. And in practice a combination of these conditions lead to failure.

The design of fault-tolerant system describes a computer system or component that, when a component fails, immediately a backup component or procedure can take its place with no loss of service. Software and hardware can provide fault tolerance respectively, or sometimes a combination of both of them.

We could use redundancy to achieve fault tolerance design. In terms of hardware level, fault tolerance is achieved by duplexing hardware component. Using multiple, identical replicas of hardware modules and a voter mechanism, modular redundancy could be realized. The function of voter is to compare all the outputs from the replicas and decides the correct output by using majority vote, for example. The machine continues to function as normal when faulty component is determined and then taken out. It is a general technique to tolerate most hardware faults in a minority of the hardware modules using modular redundancy.

In this project, initially we have a single simpleht server. If the server has a power failure for example, the whole system is crashed and this will lead to whole system loss. Thus we think about designing a fault tolerant system, put data of file system into several identical data servers and use a mediator to compare and return the right output value to client, dealing with crashed server and corrupted data.

II. Implementation

In this project, we use a mediator, a meta server and several data servers to implement the fault tolerant system. The system structure is shown in figure 1 below:

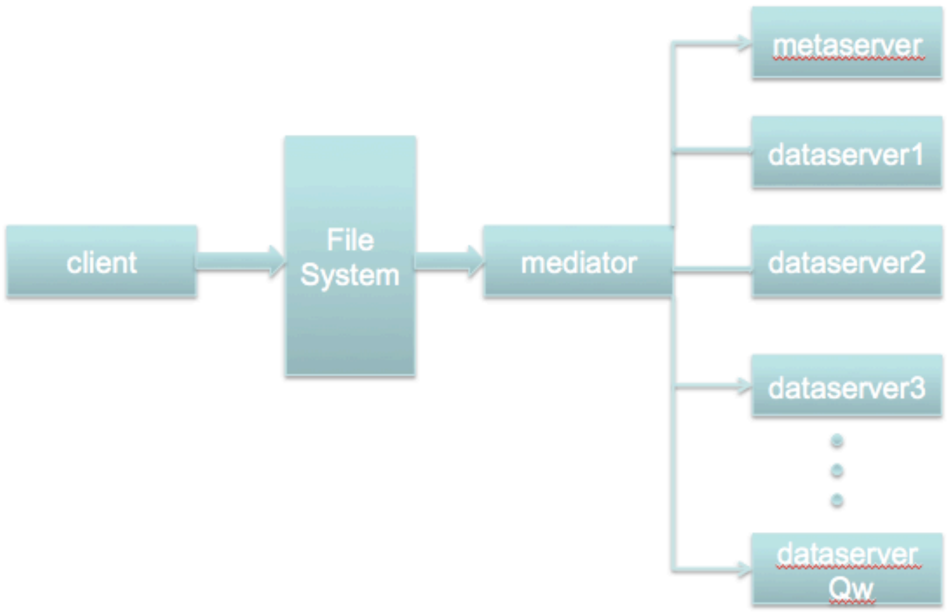
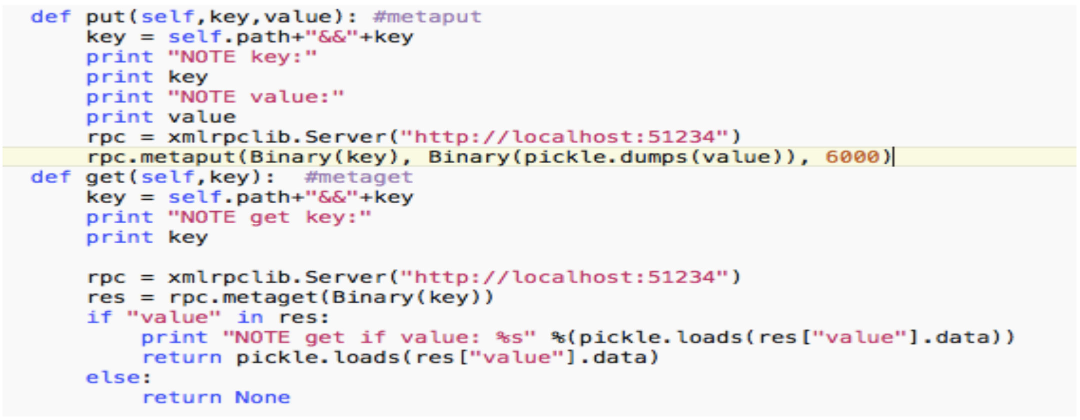


Figure 1 Project Structure

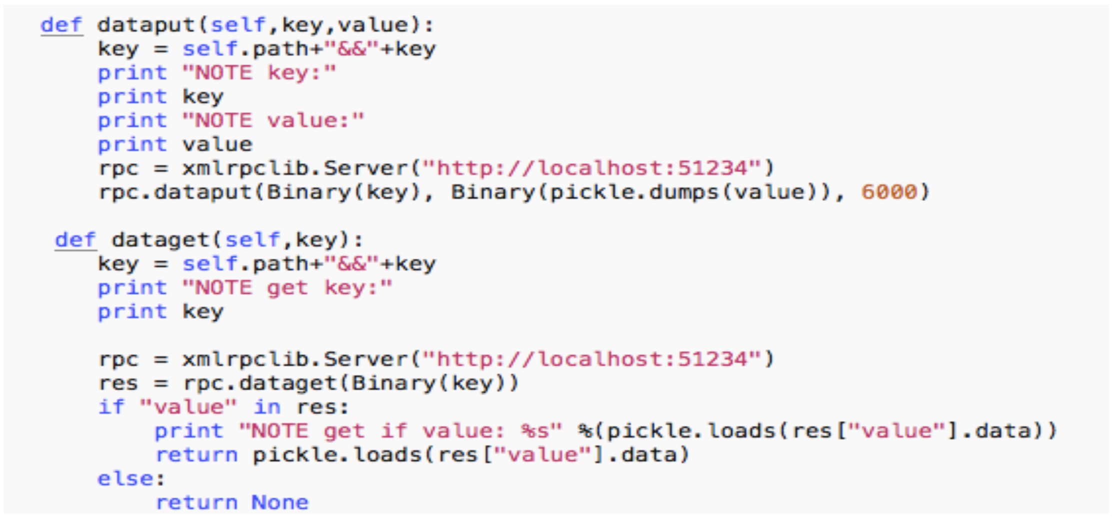
A. File System

We use remote-tree.py as the initial frame of our file system. In remote_tree.py, all data ("meta", "list_nodes" and "data") share the same "put" and "get" functions. In our project, we are required to have a metaserver to store "meta" and "list_nodes" information and dataservers to store only "data" information. So we modified "get" and "put" functions and write four functions instead to store information separately. We have "dataget" and "dataput" functions to put and retrieve data information, and "put" and "get" function to put and retrieve meta information. As shown in figure 2 and figure 3 below, we create our filesystem.py.



```
def put(self,key,value): #metaput
    key = self.path+"&&"+key
    print "NOTE key:"
    print key
    print "NOTE value:"
    print value
    rpc = xmlrpclib.Server("http://localhost:51234")
    rpc.metaput(Binary(key), Binary(pickle.dumps(value)), 6000)
def get(self,key):  #metaget
    key = self.path+"&&"+key
    print "NOTE get key:"
    print key

    rpc = xmlrpclib.Server("http://localhost:51234")
    res = rpc.metaget(Binary(key))
    if "value" in res:
        print "NOTE get if value: %s" %(pickle.loads(res["value"].data))
        return pickle.loads(res["value"].data)
    else:
        return None
```

Figure 2 Meta Put method in file system



```
def dataput(self,key,value):
    key = self.path+"&&"+key
    print "NOTE key:"
    print key
    print "NOTE value:"
    print value
    rpc = xmlrpclib.Server("http://localhost:51234")
    rpc.dataput(Binary(key), Binary(pickle.dumps(value)), 6000)

 def dataget(self,key):
    key = self.path+"&&"+key
    print "NOTE get key:"
    print key

    rpc = xmlrpclib.Server("http://localhost:51234")
    res = rpc.dataget(Binary(key))
    if "value" in res:
        print "NOTE get if value: %s" %(pickle.loads(res["value"].data))
        return pickle.loads(res["value"].data)
    else:
        return None
```

Figure 3 Dataput method in file system

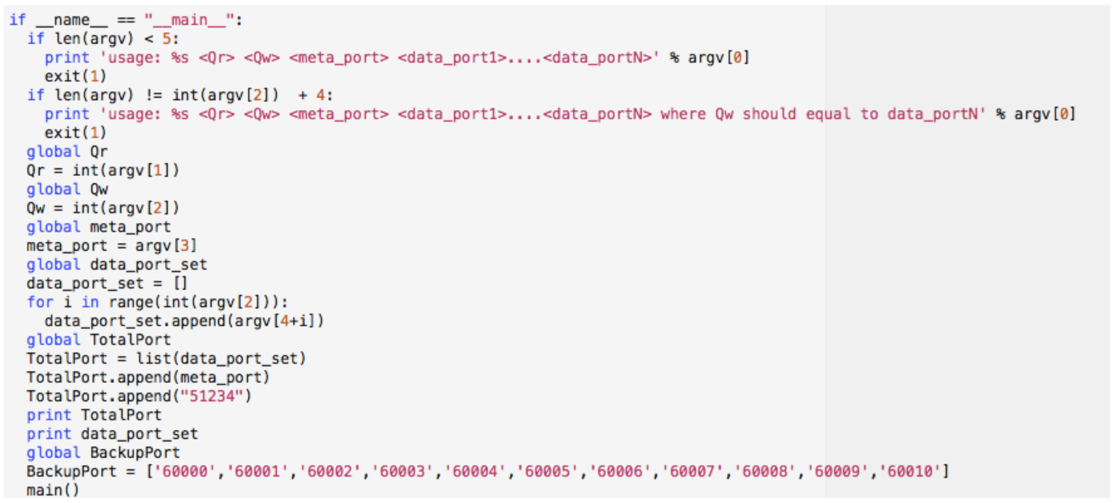
To be consistent with the functions in mediator.py (since we use “metaput” and “metaget” functions), we use “metaput” and “metaget” functions inside “put” and “get” functions as shown in figure 2 above.

B. Mediator

The mediator functions for forwarding data information to several data servers as well as compare and retrieve the correct value to client. Mediator.py is modified based on simpleht.py while we add several modules to talk about.

B.1 Global parameters definition

First we declare several global variables so we can use them in other functions. Qr and Qw are parameters that determine the minimum number of servers that need to reply to a read request and write request, “metaport” is used to set meta server’s port number and “dataport_set” is used to store a list of data server’s port numbers. The “TotalPort” variable is a list of all servers’ port numbers including the mediator, and we also declare a “BackupPort” list to store a backup servers’ port in case there is a server crashed. In our implementation, we allows for several sever crashes at the same time, therefore backup port numbers might be run out if there are more than 10 server crashes. Available backup ports are updated with the starting of a new server. The above variables are shown as below in figure 4.

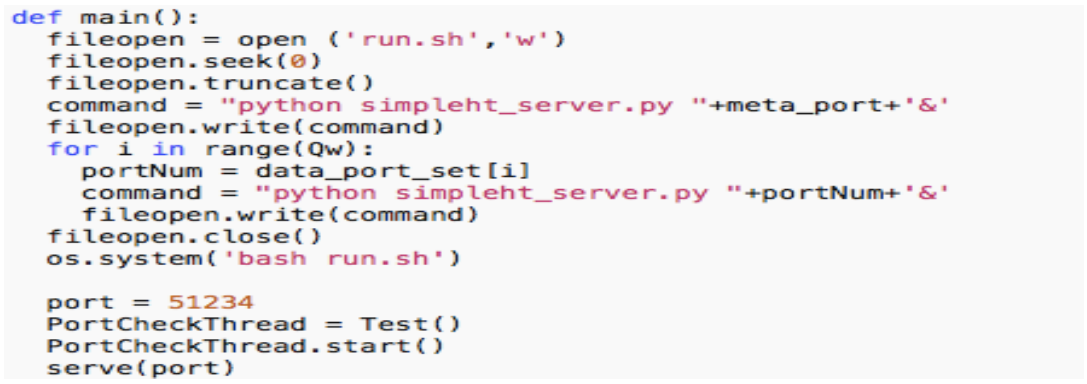


```
if __name__ == "__main__":
  if len(argv) < 5:
    print 'usage: %s <Qr> <Qw> <meta_port> <data_port1>....<data_portN>' % argv[0]
    exit(1)
  if len(argv) != int(argv[2])  + 4:
    print 'usage: %s <Qr> <Qw> <meta_port> <data_port1>....<data_portN> where Qw should equal to data_portN' % argv[0]
    exit(1)
  global Qr
  Qr = int(argv[1])
  global Qw
  Qw = int(argv[2])
  global meta_port
  meta_port = argv[3]
  global data_port_set
  data_port_set = []
  for i in range(int(argv[2])):
    data_port_set.append(argv[4+i])
  global TotalPort
  TotalPort = list(data_port_set)
  TotalPort.append(meta_port)
  TotalPort.append("51234")
  print TotalPort
  print data_port_set
  global BackupPort
  BackupPort = ['60000','60001','60002','60003','60004','60005','60006','60007','60008','60009','60010']
  main()
```

Figure 4 Global parameter definitions

B.2 Main function

Now we dive into main function. Since we need to start meta server and data servers automatically as required, we write commands to a bash file to start metaserver.py and simpleht_server.py, which is modified so that the port number is not fixed, instead it's a configurable parameter ready to accept the port number that we define in "data_port_set" list. So when the main() function is executed, the python files of servers will start automatically, which is the same as we open terminals and write commands to start them. The following figure 5 describes how this works.
There are other ways to start a sever other than using bash command. We will introduce those ways later.



```
def main():
  fileopen = open ('run.sh','w')
  fileopen.seek(0)
  fileopen.truncate()
  command = "python simpleht_server.py "+meta_port+'&'
  fileopen.write(command)
  for i in range(Qw):
    portNum = data_port_set[i]
    command = "python simpleht_server.py "+portNum+'&'
    fileopen.write(command)
  fileopen.close()
  os.system('bash run.sh')

  port = 51234
  PortCheckThread = Test()
  PortCheckThread.start()
  serve(port)
```

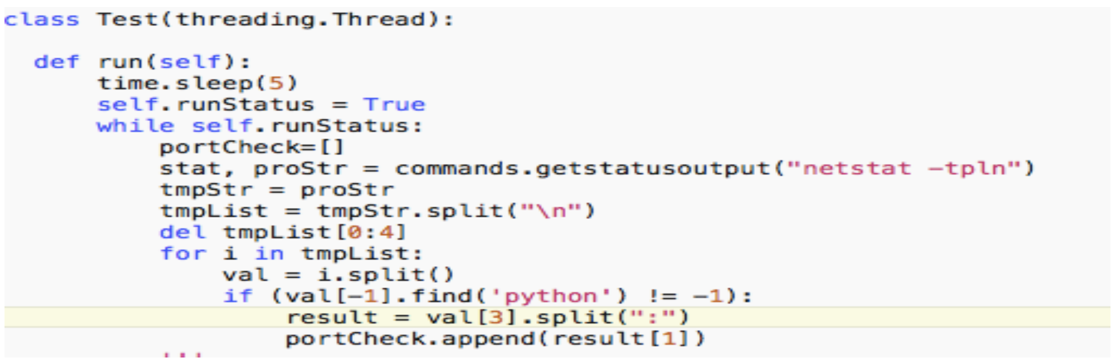
Figure 5 Main function

As the figure shows above, after we start meta servers and several data servers, we call Test() and start the thread, then we call serve(port) function to execute server registered functions as the simpleht.py did before.

B.3 Test port status and port check algorithms

In our implementation, port number is considered as the representative of a live process, which means each server's port number is binding to its process.

We define a class named “Test” that uses thread to monitor servers’ ports periodically, which implements port check. When a server crashes, its process is closed with the corresponding port number. In order to monitor whether a server works properly, we just need to check whether its port number exists or not. Typically, we use “netstat -tpln” command in terminal to check the process ID and port number, thus we are thinking about using commands module in python to execute “netstat -tpln” to extract the servers’ port number. We first define a “portCheck” list to store the servers’ port number, next in a while loop when “runstatus” is true, we use “commands. getoutputstat()” function to execute the netstat command, then we find the corresponding item ending in “python” , finally we extract port number and add them into the “portCheck” list. The following figure 6 shows how this works.
A noteworthy point in running our project is that since we need to find port number of server corresponding to “python” file before we run the mediator.py, we have to clean all python files with port number in the system.

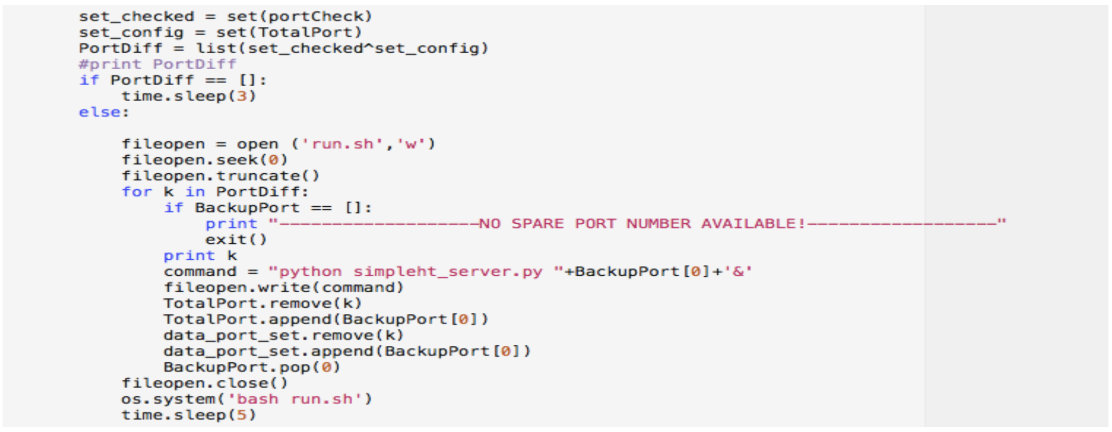


```
class Test(threading.Thread):

  def run(self):
      time.sleep(5)
      self.runStatus = True
      while self.runStatus:
          portCheck=[]
          stat, proStr = commands.getstatusoutput("netstat -tpln")
          tmpStr = proStr
          tmpList = tmpStr.split("\n")
          del tmpList[0:4]
          for i in tmpList:
              val = i.split()
              if (val[-1].find('python') != -1):
                  result = val[3].split(":")
                  portCheck.append(result[1])
          ...
```

Figure 6 Test thread method

If a server crashes, the corresponding port number is gone, and we need a mechanism to restart a server. To check this, we compare the “portCheck” and “TotalPort” lists and find the difference set. If the difference set is none, it means no server crashes and port check will continue after 3 seconds, if any changes are made to port, we have to find the dead port number and remove it both from “portCheck” and “TotalPort” list. In the meanwhile, we write commands to bash file to restart a simpleht_server.py and allocate a new port number from “BackupPort” list to the new server, as well as remove the port number from “BackupPort” list. After 5 seconds, we continue to check port numbers again. Since there is time elapse for a new python file to start, we have to set 5 seconds sleep before next port check. Otherwise, port check thread will find the same difference before the new sever starts.

In this project, only one server will crash according to requirement, but we also deal with crash of multiple servers. Every time a server crashes, a new server with new port number will restart, until backup port numbers are all used, we will print, “No space port number is available”. The code shows below illustrate it.

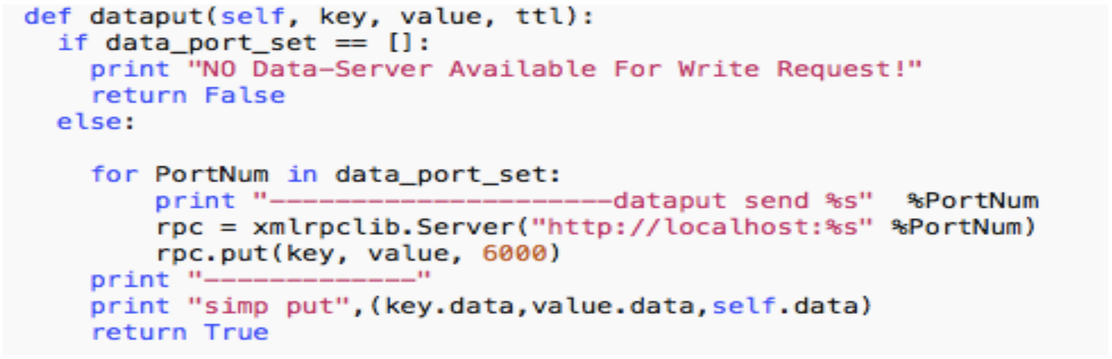


```
set_checked = set(portCheck)
set_config = set(TotalPort)
PortDiff = list(set_checked^set_config)
#print PortDiff
if PortDiff == []:
    time.sleep(3)
else:

    fileopen = open ('run.sh','w')
    fileopen.seek(0)
    fileopen.truncate()
    for k in PortDiff:
        if BackupPort == []:
            print "------------------NO SPARE PORT NUMBER AVAILABLE!------------------"
            exit()
        print k
        command = "python simpleht_server.py "+BackupPort[0]+'&'
        fileopen.write(command)
        TotalPort.remove(k)
        TotalPort.append(BackupPort[0])
        data_port_set.remove(k)
        data_port_set.append(BackupPort[0])
        BackupPort.pop(0)
    fileopen.close()
    os.system('bash run.sh')
    time.sleep(5)
```

Figure 7 Port Check Algorithms

B.4 Read request selection algorithm

Since meta server is assumed to be never crashed as required, there is no modification in "metaput" and "metaget" functions, so here we just talk about how data is put and retrieve from servers in "dataput" and "dataget" functions.
There is not much modification in "dataput" function, we first check whether the "data_port_set" is empty or not. If empty, we will print "No data server available for write request". Otherwise, we use rpc.put to write data in several dataservers. As shown in figure 8.



```
def dataput(self, key, value, ttl):
  if data_port_set == []:
    print "NO Data-Server Available For Write Request!"
    return False
  else:

    for PortNum in data_port_set:
        print "---------------------dataput send %s"  %PortNum
        rpc = xmlrpclib.Server("http://localhost:%s" %PortNum)
        rpc.put(key, value, 6000)
    print "-------------"
    print "simp put",(key.data,value.data,self.data)
    return True
```

Figure 8 Dataput Method

In "dataget" function, we randomly select "Qr" data servers to read data. Data are retrieved from all data servers and put into a list named "getBinaryResult", which means that what we get from "rpc.get(key)" is binary value. Binary value can not be compared, so we use another list named "getValResult" to store the actual value if there is value in "rpc.get(key)".

The mediator then finds the majority value retrieved from all data servers and sends it to client. We use "Counter(getValResult)" to compute the frequency of each value and assign it to variable "countResult", then we find the majority value using "countResult.mostcommon()", we get this value and find its index in the list "getValResult", then we use this index to get the corresponding value in "getBinaryResult" and returned. The code is shown in following figure 9.

```
def dataget(self, key):
  RandDataServer = random.sample(data_port_set, Qr)
  getBinaryResult = []
  getValResult = []
  print "---------------random select read dataserver-----------"
  print RandDataServer
  for e in RandDataServer:
    rpc = xmlrpclib.Server("http://localhost:%s" %e)
    tmp = rpc.get(key)
    getBinaryResult.append(tmp)
    if "value" in tmp:
      getValResult.append(pickle.loads(tmp["value"].data))
    else:
      getValResult.append('None')


  print  "---------------read value set-----------"
  print getBinaryResult
  print '----------get bi | get val--------'
  print getValResult
  #Find resluts that appeas most-commonly, choose this value as voter selected one
  countResult = Counter(getValResult)
  print countResult
  SelectValue = countResult.most_common()
  selectVal = SelectValue[0][0]
  indexNum = getValResult.index(selectVal)
  res = getBinaryResult[indexNum]
  print res
  return res
```

Figure 9 Read request selection algorithm

C. Data Server

In this project, we need to deal with data corruption and server crash. In our design of data server, we implement the required 3 functions as follows.

C.1 List content function

The function "list_contents()" returns a list of keys corresponding to values stored on the data server. We print the key-value pair in the server terminal and return it to the client end, shown in figure 10 below.

```
def list_contents(self):
  print "-----------------Key -> Data -------------------"
  #two solutions for search with different time cost, the other is
  for key in self.data.keys():
    print "%s             --->              %s" %(key,self.data[key])
  return self.data
```

Figure 10 List content funciton

C.2 Terminate function

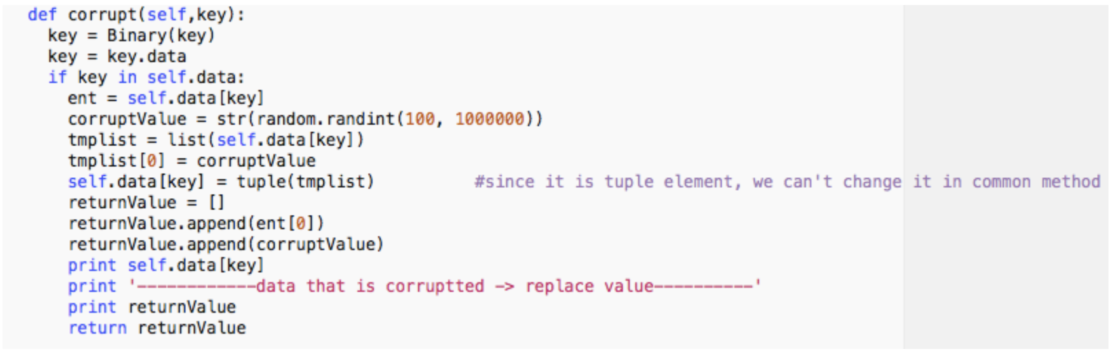
The method "terminate()" is used to stop the server. In our implementation, we use "os.system" to execute system commands, to kill the server process. The code is shown in Figure 11 below.

```
def terminate(self):
  killport = str(port)
  os.system('fuser -k  %s/tcp' %killport)
  return True
```

Figure 11 Terminate funciton

C.3 Corrupt function

The method "corrupt(key)" returns the corrupted value. This method is used to corrupt the value corresponding to the key in the data-server. In our implementation, we first convert the key into binary value, and then we produce a random number between 100 and 1000000 as the corrupted value. Since "self.data[key]" is a tuple element and it is immutable, if we want to add the corrupted value into it, we first have to put the "self.data[key]" into a list and add the corrupted value into the list, then covert the list to tuple element again. In our implementation, we return both the correct value and corrupted value in order to compare explicitly. The following figure11 shows the code.



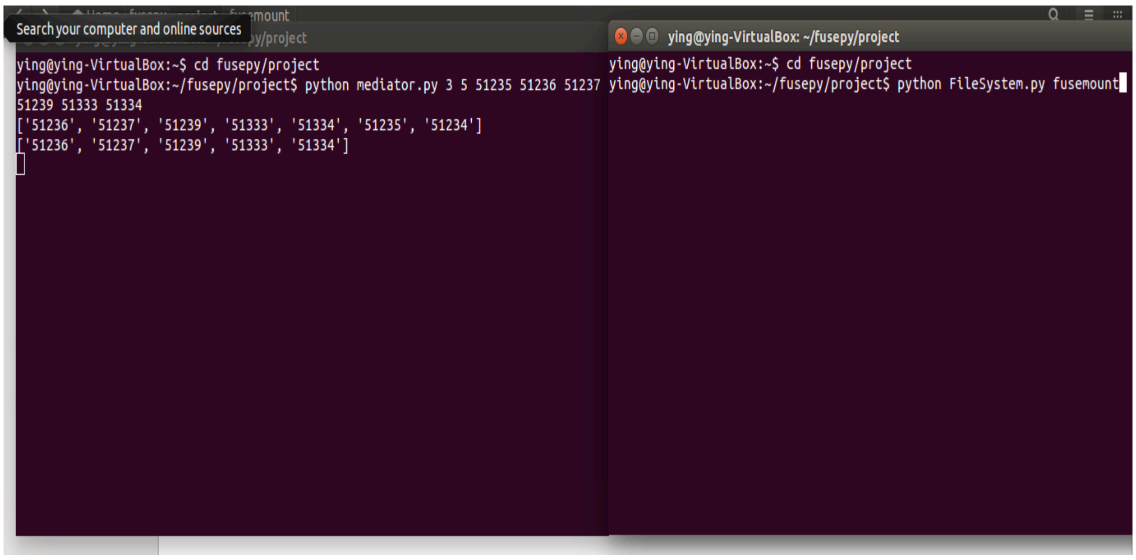
```
def corrupt(self,key):
  key = Binary(key)
  key = key.data
  if key in self.data:
    ent = self.data[key]
    corruptValue = str(random.randint(100, 1000000))
    tmplist = list(self.data[key])
    tmplist[0] = corruptValue
    self.data[key] = tuple(tmplist)         #since it is tuple element, we can't change it in common method
    returnValue = []
    returnValue.append(ent[0])
    returnValue.append(corruptValue)
    print self.data[key]
    print '------------data that is corruptted -> replace value----------'
    print returnValue
    return returnValue
```

Figure 11 Corrupt function

III. Testing Mechanism

A. Start Servers

We first start mediator.py, here we choose Qr = 3, Qw = 5, and we type meta server's port number and five data servers' port numbers, the two lists shown below are total existing port list and assigned data server port list , as shown in figure12.

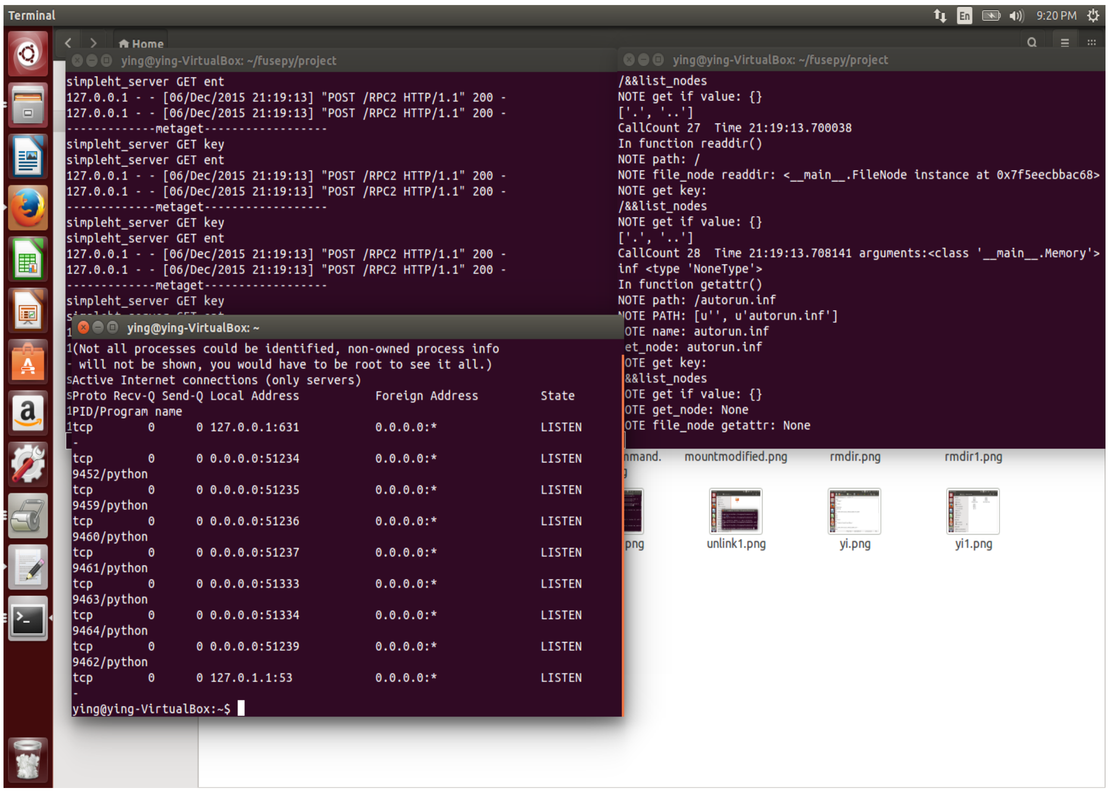

Search your computer and online sources
ying@ying-VirtualBox: ~/fusepy/project

```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ python mediator.py 3 5 51235 51236 51237
51239 51333 51334
['51236', '51237', '51239', '51333', '51334', '51235', '51234']
['51236', '51237', '51239', '51333', '51334']
```

```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ python FileSystem.py fusemount
```

Figure 12 Start Server

Then we start FileSystem.py, and use “netstat -tpln” commands to see that all servers start and is waiting for connections, as shown in figure 13 below.



```
ying@ying-VirtualBox: ~/fusepy/project
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:19:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:19:13] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:19:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:19:13] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:19:13] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:19:13] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
```

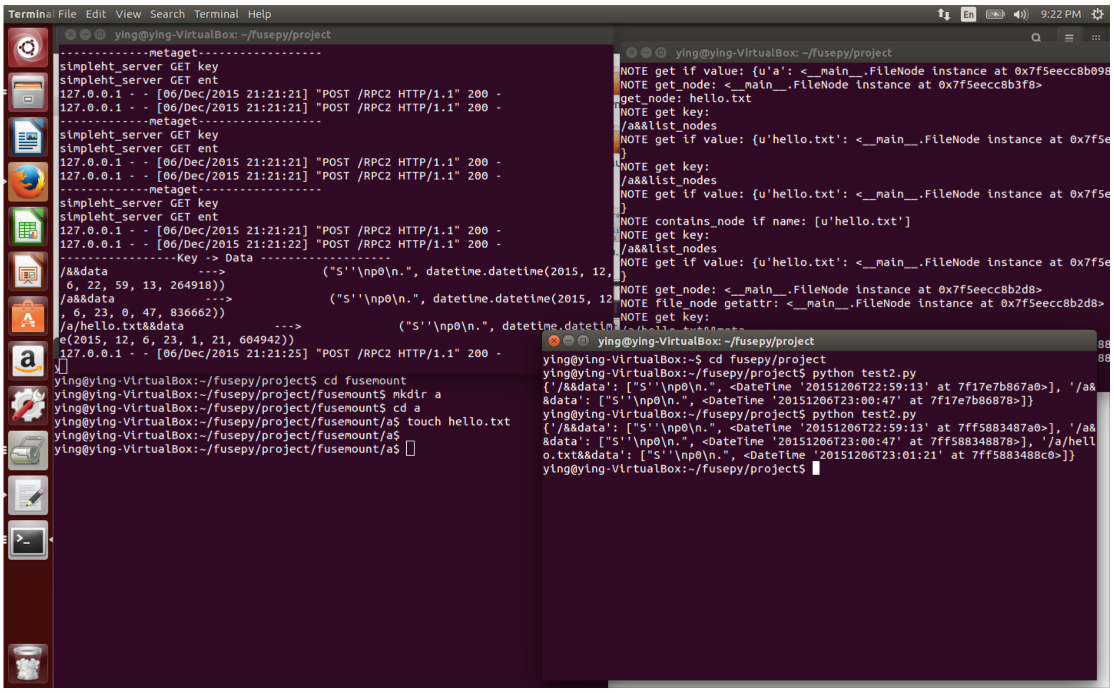
```
ying@ying-VirtualBox: ~/fusepy/project
/&&list_nodes
NOTE get if value: {}
['.', '..']
CallCount 27  Time 21:19:13.700038
In function readdir()
NOTE path: /
NOTE file_node readdir: <__main__.FileNode instance at 0x7f5eecbbac68>
NOTE get key:
/&&list_nodes
NOTE get if value: {}
['.', '..']
CallCount 28  Time 21:19:13.708141 arguments:<class '__main__.Memory'>
inf <type 'NoneType'>
In function getattr()
NOTE path: /autorun.inf
NOTE PATH: [u'', u'autorun.inf']
OTE name: autorun.inf
et_node: autorun.inf
OTE get key:
&&list_nodes
OTE get if value: {}
OTE get_node: None
OTE file_node getattr: None
```

```
ying@ying-VirtualBox: ~
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:51234           0.0.0.0:*               LISTEN
9452/python
tcp        0      0 0.0.0.0:51235           0.0.0.0:*               LISTEN
9459/python
tcp        0      0 0.0.0.0:51236           0.0.0.0:*               LISTEN
9460/python
tcp        0      0 0.0.0.0:51237           0.0.0.0:*               LISTEN
9461/python
tcp        0      0 0.0.0.0:51333           0.0.0.0:*               LISTEN
9463/python
tcp        0      0 0.0.0.0:51334           0.0.0.0:*               LISTEN
9464/python
tcp        0      0 0.0.0.0:51239           0.0.0.0:*               LISTEN
9462/python
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
-
ying@ying-VirtualBox:~$
```

Figure 13 Listening on the port

B. Test list_contents function

We write a test2.py file to start list_contents method in server. In the terminal of bottom left corner, we get into fusemount directory and use “mkdir” command to build a folder named “a”, then we start test2.py in bottom right corner, we can see contents corresponding to “a”. Then we build a file named “hello.txt” in directory a, run test2.py again, we can see contents of the txt file both from terminals of test2.py and mediator.py, as shown in figure 14.



```
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:21:21] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:21:21] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:21:21] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:21:21] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:21:21] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:21:22] "POST /RPC2 HTTP/1.1" 200 -
----------------Key -> Data ------------------
/&&data              --->              ("S''\np0\n.", datetime.datetime(2015, 12,
 6, 22, 59, 13, 264918))
/a&&data              --->              ("S''\np0\n.", datetime.datetime(2015, 12
, 6, 23, 0, 47, 836662))
/a/hello.txt&&data              --->              ("S''\np0\n.", datetime.datetim
e(2015, 12, 6, 23, 1, 21, 604942))
127.0.0.1 - - [06/Dec/2015 21:21:25] "POST /RPC2 HTTP/1.1" 200 -
```

```
ying@ying-VirtualBox:~/fusepy/project$ cd fusemount
ying@ying-VirtualBox:~/fusepy/project/fusemount$ mkdir a
ying@ying-VirtualBox:~/fusepy/project/fusemount$ cd a
ying@ying-VirtualBox:~/fusepy/project/fusemount/a$ touch hello.txt
ying@ying-VirtualBox:~/fusepy/project/fusemount/a$
ying@ying-VirtualBox:~/fusepy/project/fusemount/a$
```

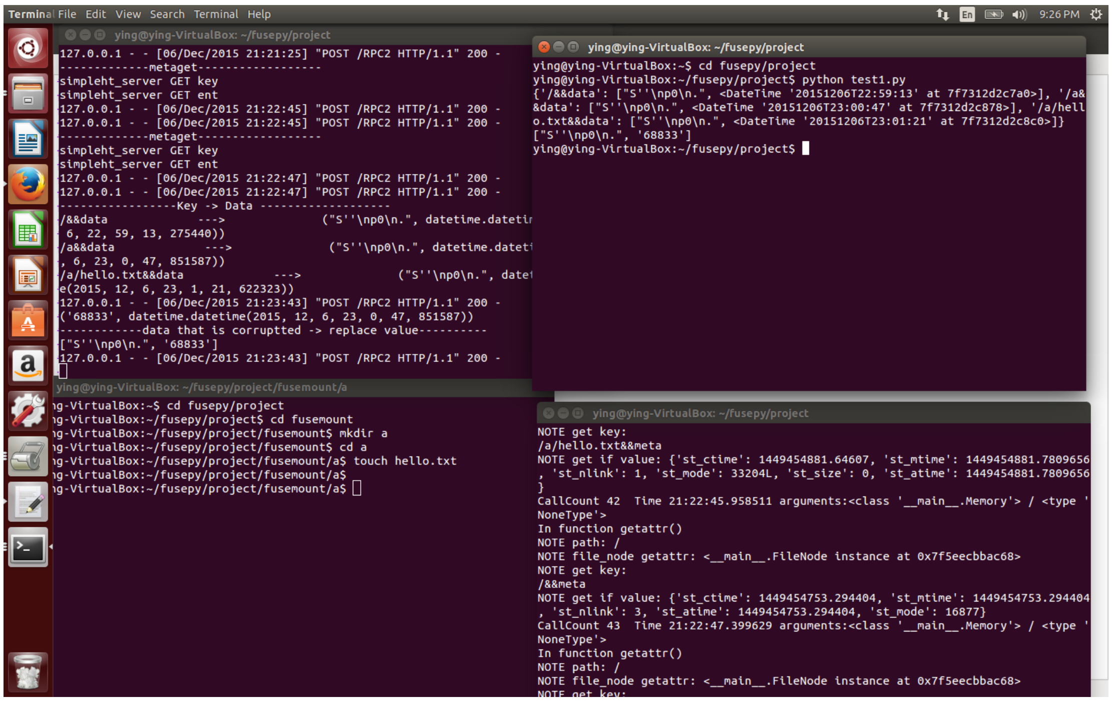
```
NOTE get if value: {u'a': <__main__.FileNode instance at 0x7f5eecc8b098
NOTE get_node: <__main__.FileNode instance at 0x7f5eecc8b3f8>
get_node: hello.txt
NOTE get key:
/a&&list_nodes
NOTE get if value: {u'hello.txt': <__main__.FileNode instance at 0x7f5e
}
NOTE get key:
/a&&list_nodes
NOTE get if value: {u'hello.txt': <__main__.FileNode instance at 0x7f5e
}
NOTE contains_node if name: [u'hello.txt']
NOTE get key:
/a&&list_nodes
NOTE get if value: {u'hello.txt': <__main__.FileNode instance at 0x7f5e
}
NOTE get_node: <__main__.FileNode instance at 0x7f5eecc8b2d8>
NOTE file_node getattr: <__main__.FileNode instance at 0x7f5eecc8b2d8>
NOTE get key:
```

```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ python test2.py
{'/&&data': ["S''\np0\n.", <DateTime '20151206T22:59:13' at 7f17e7b867a0>], '/a&
&data': ["S''\np0\n.", <DateTime '20151206T23:00:47' at 7f17e7b86878>]}
ying@ying-VirtualBox:~/fusepy/project$ python test2.py
{'/&&data': ["S''\np0\n.", <DateTime '20151206T22:59:13' at 7ff5883487a0>], '/a&
&data': ["S''\np0\n.", <DateTime '20151206T23:00:47' at 7ff588348878>], '/a/hell
o.txt&&data': ["S''\np0\n.", <DateTime '20151206T23:01:21' at 7ff5883488c0>]}
ying@ying-VirtualBox:~/fusepy/project$
```

Figure 14 Test project functionality and list content function

C. Test corrupt (key) function

We write a test1.py file which connects to port number 51333. To start corrupt (key) method in data server, we start a new terminal to run test1.py, since we print both correct value and corrupted value in order to compare explicitly, we can see that in the terminal of test1.py in top right corner, a random number "68833" is retrieved as a corrupted value. In the terminal of mediator.py in left top corner, correct value of "/a/hello.txt" is retrieved, to see clearly, we print data value that is corrupted "68833" as well, as shown in figure 15.



```
127.0.0.1 - - [06/Dec/2015 21:21:25] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:22:45] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:22:45] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:22:47] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:22:47] "POST /RPC2 HTTP/1.1" 200 -
----------------Key -> Data ------------------
/&&data              --->              ("S''\np0\n.", datetime.datetin
 6, 22, 59, 13, 275440))
/a&&data              --->              ("S''\np0\n.", datetime.dateti
, 6, 23, 0, 47, 851587))
/a/hello.txt&&data              --->              ("S''\np0\n.", datet
e(2015, 12, 6, 23, 1, 21, 622323))
127.0.0.1 - - [06/Dec/2015 21:23:43] "POST /RPC2 HTTP/1.1" 200 -
('68833', datetime.datetime(2015, 12, 6, 23, 0, 47, 851587))
-----------data that is corruptted -> replace value----------
["S''\np0\n.", '68833']
127.0.0.1 - - [06/Dec/2015 21:23:43] "POST /RPC2 HTTP/1.1" 200 -
```

```
ng-VirtualBox:~$ cd fusepy/project
ng-VirtualBox:~/fusepy/project$ cd fusemount
ng-VirtualBox:~/fusepy/project/fusemount$ mkdir a
ng-VirtualBox:~/fusepy/project/fusemount$ cd a
ng-VirtualBox:~/fusepy/project/fusemount/a$ touch hello.txt
ng-VirtualBox:~/fusepy/project/fusemount/a$
ng-VirtualBox:~/fusepy/project/fusemount/a$
```

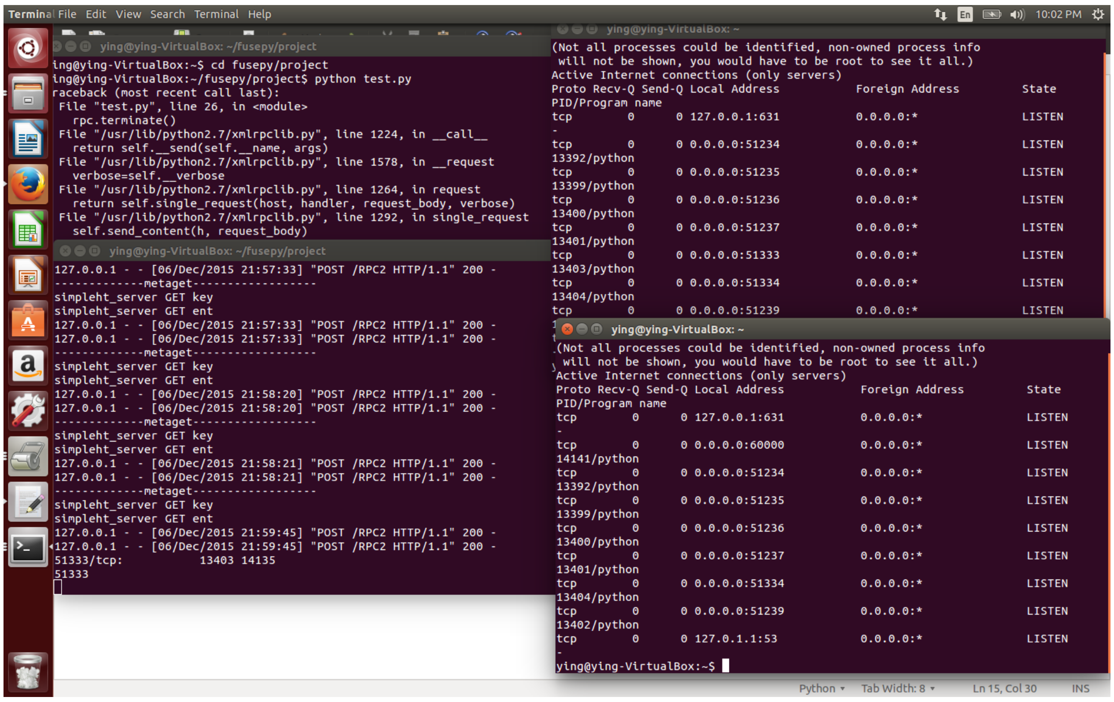
```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ python test1.py
{'/&&data': ["S''\np0\n.", <DateTime '20151206T22:59:13' at 7f7312d2c7a0>], '/a&
&data': ["S''\np0\n.", <DateTime '20151206T23:00:47' at 7f7312d2c878>], '/a/hell
o.txt&&data': ["S''\np0\n.", <DateTime '20151206T23:01:21' at 7f7312d2c8c0>]}
["S''\np0\n.", '68833']
ying@ying-VirtualBox:~/fusepy/project$
```

```
NOTE get key:
/a/hello.txt&&meta
NOTE get if value: {'st_ctime': 1449454881.64607, 'st_mtime': 1449454881.7809656
, 'st_nlink': 1, 'st_mode': 33204L, 'st_size': 0, 'st_atime': 1449454881.7809656
}
CallCount 42  Time 21:22:45.958511 arguments:<class '__main__.Memory'> / <type '
NoneType'>
In function getattr()
NOTE path: /
NOTE file_node getattr: <__main__.FileNode instance at 0x7f5eecbbac68>
NOTE get key:
/&&meta
NOTE get if value: {'st_ctime': 1449454753.294404, 'st_mtime': 1449454753.294404
, 'st_nlink': 3, 'st_atime': 1449454753.294404, 'st_mode': 16877}
CallCount 43  Time 21:22:47.399629 arguments:<class '__main__.Memory'> / <type '
NoneType'>
In function getattr()
NOTE path: /
NOTE file_node getattr: <__main__.FileNode instance at 0x7f5eecbbac68>
NOTE get key:
```

Figure 15 Corrupt function

D. Test terminate () function

We write a test.py file corresponding to port number 51333. In order to compare, we use "netstat" command to show the port numbers of server before and after starting terminate method. To start terminate() method in data server, we start test.py in a new terminal, after that, we run "netstat" again to see the change of port numbers. In the figure16 below, we can see that port number "51333" is replaced by the first backup port number "60000". That means server with port number "51333" is terminated and a new server with port "60000" restarts.



```
ing@ying-VirtualBox:~$ cd fusepy/project
ing@ying-VirtualBox:~/fusepy/project$ python test.py
raceback (most recent call last):
 File "test.py", line 26, in <module>
   rpc.terminate()
 File "/usr/lib/python2.7/xmlrpclib.py", line 1224, in __call__
   return self.__send(self.__name, args)
 File "/usr/lib/python2.7/xmlrpclib.py", line 1578, in __request
   verbose=self.__verbose
 File "/usr/lib/python2.7/xmlrpclib.py", line 1264, in request
   return self.single_request(host, handler, request_body, verbose)
 File "/usr/lib/python2.7/xmlrpclib.py", line 1292, in single_request
   self.send_content(h, request_body)
```

```
127.0.0.1 - - [06/Dec/2015 21:57:33] "POST /RPC2 HTTP/1.1" 200 -
-------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:57:33] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:57:33] "POST /RPC2 HTTP/1.1" 200 -
-------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:58:20] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:58:20] "POST /RPC2 HTTP/1.1" 200 -
-------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:58:21] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:58:21] "POST /RPC2 HTTP/1.1" 200 -
-------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:59:45] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:59:45] "POST /RPC2 HTTP/1.1" 200 -
51333/tcp:           13403 14135
51333
```

```
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:51234           0.0.0.0:*               LISTEN
13392/python
tcp        0      0 0.0.0.0:51235           0.0.0.0:*               LISTEN
13399/python
tcp        0      0 0.0.0.0:51236           0.0.0.0:*               LISTEN
13400/python
tcp        0      0 0.0.0.0:51237           0.0.0.0:*               LISTEN
13401/python
tcp        0      0 0.0.0.0:51333           0.0.0.0:*               LISTEN
13403/python
tcp        0      0 0.0.0.0:51334           0.0.0.0:*               LISTEN
13404/python
tcp        0      0 0.0.0.0:51239           0.0.0.0:*               LISTEN
```

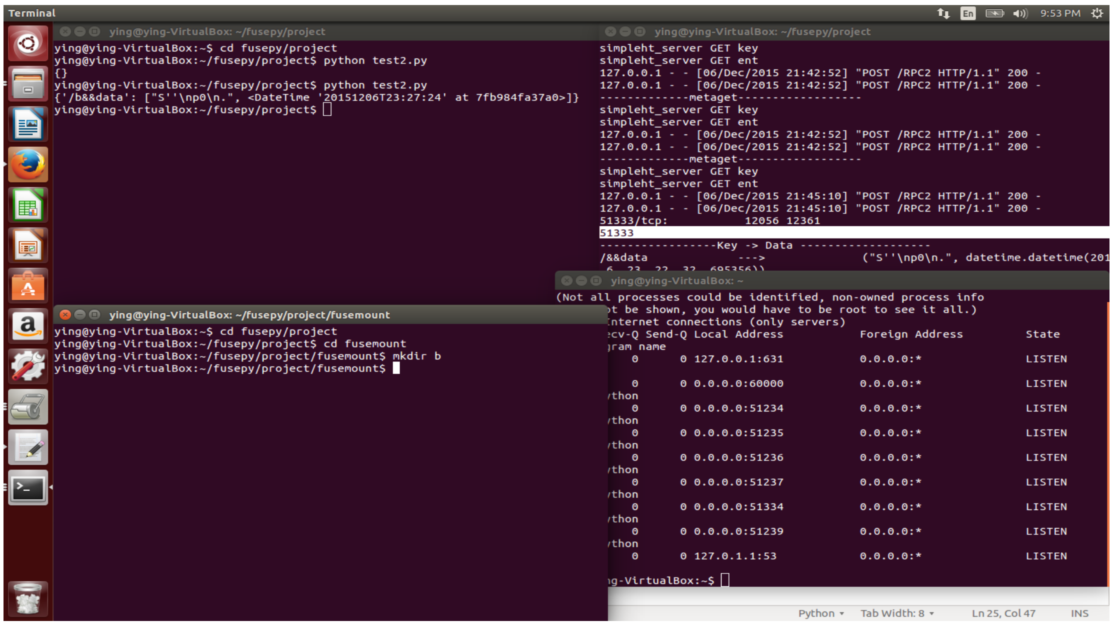
```
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:60000           0.0.0.0:*               LISTEN
14141/python
tcp        0      0 0.0.0.0:51234           0.0.0.0:*               LISTEN
13392/python
tcp        0      0 0.0.0.0:51235           0.0.0.0:*               LISTEN
13399/python
tcp        0      0 0.0.0.0:51236           0.0.0.0:*               LISTEN
13400/python
tcp        0      0 0.0.0.0:51237           0.0.0.0:*               LISTEN
13401/python
tcp        0      0 0.0.0.0:51334           0.0.0.0:*               LISTEN
13404/python
tcp        0      0 0.0.0.0:51239           0.0.0.0:*               LISTEN
13402/python
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
-
ying@ying-VirtualBox:~$
```

Figure 16 Terminate function

E. Test status of replacement server

When server with port number 51333 is terminated and a new server with port 6000 restarts, (as shown in figure 17 below in the top right corner and bottom right corner.) we need to check whether the new server starts from a blank state as required, that is, without any file data.

In the terminal in top left corner, we start test2.py to see the contents of the newly start server with port 60000. We can see that only a null list is retrieved. That means server with port 60000 starts from a blank state. We then get into fusemount directory and use "mkdir" command to build a directory named "b", run test2.py again, and this time we can see there is content corresponding to "b" is retrieved. That means the latest information will be put into a restart server.



```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ python test2.py
{}
ying@ying-VirtualBox:~/fusepy/project$ python test2.py
{'/b&&data': ["S''\np0\n.", <DateTime '20151206T23:27:24' at 7fb984fa37a0>]}
ying@ying-VirtualBox:~/fusepy/project$
```

```
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:42:52] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:42:52] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:42:52] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:42:52] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 21:45:10] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 21:45:10] "POST /RPC2 HTTP/1.1" 200 -
51333/tcp:            12056 12361
51333
----------------Key -> Data ------------------
/&&data               --->              ("S''\np0\n.", datetime.datetime(201
```

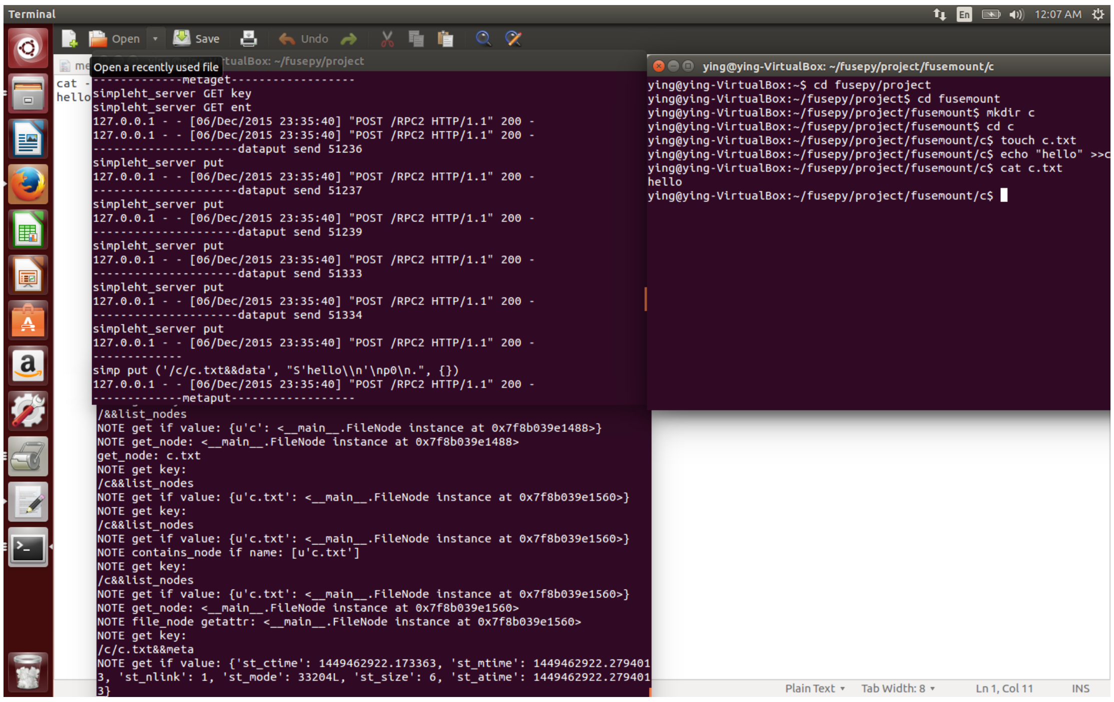
```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ cd fusemount
ying@ying-VirtualBox:~/fusepy/project/fusemount$ mkdir b
ying@ying-VirtualBox:~/fusepy/project/fusemount$
```

```
(Not all processes could be identified, non-owned process info
ot be shown, you would have to be root to see it all.)
nternet connections (only servers)
ecv-Q Send-Q Local Address           Foreign Address         State
gram name
    0      0 127.0.0.1:631           0.0.0.0:*               LISTEN

    0      0 0.0.0.0:60000           0.0.0.0:*               LISTEN
ython
    0      0 0.0.0.0:51234           0.0.0.0:*               LISTEN
ython
    0      0 0.0.0.0:51235           0.0.0.0:*               LISTEN
ython
    0      0 0.0.0.0:51236           0.0.0.0:*               LISTEN
ython
    0      0 0.0.0.0:51237           0.0.0.0:*               LISTEN
ython
    0      0 0.0.0.0:51334           0.0.0.0:*               LISTEN
ython
    0      0 0.0.0.0:51239           0.0.0.0:*               LISTEN
ython
    0      0 127.0.1.1:53            0.0.0.0:*               LISTEN

g-VirtualBox:~$
```

Figure 17 Test status of replacement server

E. Test Write Request

The following figure 18 shows how data is written to Qw data servers(In this case, Qw equals five). We can see from figure in top left corner that, the data information is put into five data servers respectively, denoted by "dataput send" corresponding to certain port.



```
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
--------------------dataput send 51236
simpleht_server put
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
--------------------dataput send 51237
simpleht_server put
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
--------------------dataput send 51239
simpleht_server put
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
--------------------dataput send 51333
simpleht_server put
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
--------------------dataput send 51334
simpleht_server put
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
------------
simp put ('/c/c.txt&&data', "S'hello\\n'\np0\n.", {})
127.0.0.1 - - [06/Dec/2015 23:35:40] "POST /RPC2 HTTP/1.1" 200 -
------------metaput-----------------
```

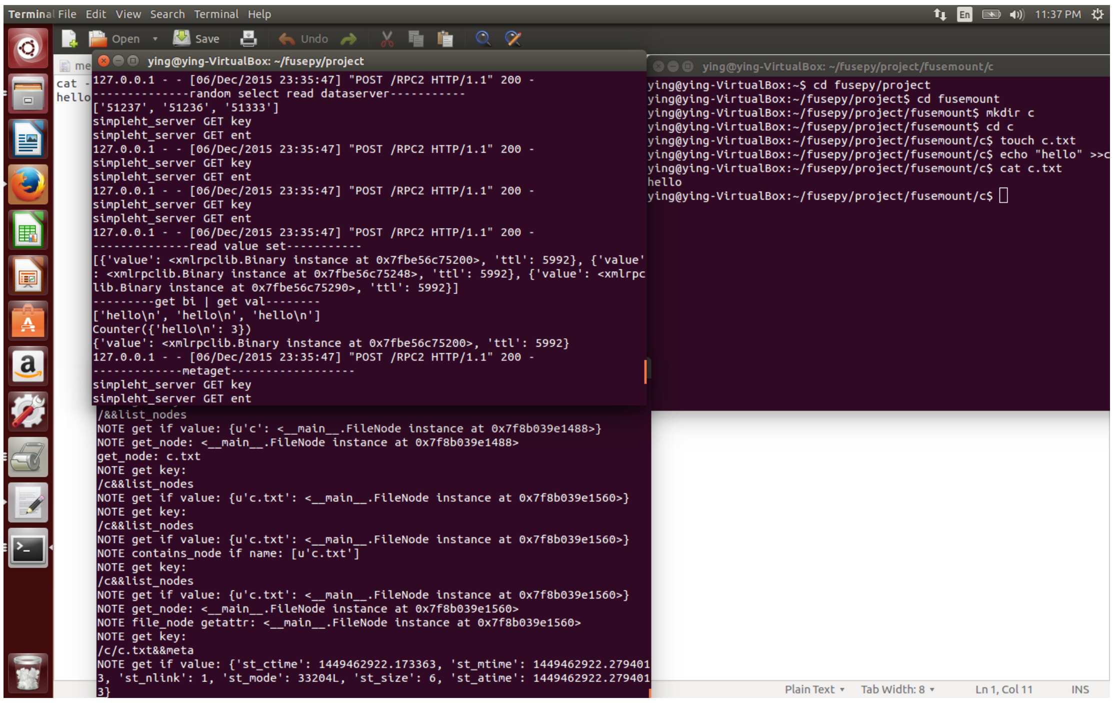
```
/&&list_nodes
NOTE get if value: {u'c': <__main__.FileNode instance at 0x7f8b039e1488>}
NOTE get_node: <__main__.FileNode instance at 0x7f8b039e1488>
get_node: c.txt
NOTE get key:
/c&&list_nodes
NOTE get if value: {u'c.txt': <__main__.FileNode instance at 0x7f8b039e1560>}
NOTE get key:
/c&&list_nodes
NOTE get if value: {u'c.txt': <__main__.FileNode instance at 0x7f8b039e1560>}
NOTE contains_node if name: [u'c.txt']
NOTE get key:
/c&&list_nodes
NOTE get if value: {u'c.txt': <__main__.FileNode instance at 0x7f8b039e1560>}
NOTE get_node: <__main__.FileNode instance at 0x7f8b039e1560>
NOTE file_node getattr: <__main__.FileNode instance at 0x7f8b039e1560>
NOTE get key:
/c/c.txt&&meta
NOTE get if value: {'st_ctime': 1449462922.173363, 'st_mtime': 1449462922.279401
3, 'st_nlink': 1, 'st_mode': 33204L, 'st_size': 6, 'st_atime': 1449462922.279401
3}
```

```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ cd fusemount
ying@ying-VirtualBox:~/fusepy/project/fusemount$ mkdir c
ying@ying-VirtualBox:~/fusepy/project/fusemount$ cd c
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$ touch c.txt
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$ echo "hello" >>c
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$ cat c.txt
hello
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$
```

Figure 18 Test Write Request

F. Test Read Request

In order to test read request, we use commands "touch" to build a file named c.txt, write some text in it and read from it, to see how the read process works. As is shown in figure in the top left figure, we first randomly select Qr read data servers (In this case, Qr equals 3), we can see they are servers with port number 51237, 51236, 51333. Then we get values from all the three servers, compute the frequency of each value and retrieve the most common value. In this case, all the values from data servers are the same, so we just return the first value of data server.



```
ying@ying-VirtualBox: ~/fusepy/project
127.0.0.1 - - [06/Dec/2015 23:35:47] "POST /RPC2 HTTP/1.1" 200 -
-------------random select read dataserver----------
['51237', '51236', '51333']
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 23:35:47] "POST /RPC2 HTTP/1.1" 200 -
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 23:35:47] "POST /RPC2 HTTP/1.1" 200 -
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [06/Dec/2015 23:35:47] "POST /RPC2 HTTP/1.1" 200 -
-------------read value set----------
[{'value': <xmlrpclib.Binary instance at 0x7fbe56c75200>, 'ttl': 5992}, {'value'
: <xmlrpclib.Binary instance at 0x7fbe56c75248>, 'ttl': 5992}, {'value': <xmlrpc
lib.Binary instance at 0x7fbe56c75290>, 'ttl': 5992}]
---------get bi | get val--------
['hello\n', 'hello\n', 'hello\n']
Counter({'hello\n': 3})
{'value': <xmlrpclib.Binary instance at 0x7fbe56c75200>, 'ttl': 5992}
127.0.0.1 - - [06/Dec/2015 23:35:47] "POST /RPC2 HTTP/1.1" 200 -
------------metaget-----------------
simpleht_server GET key
simpleht_server GET ent
```

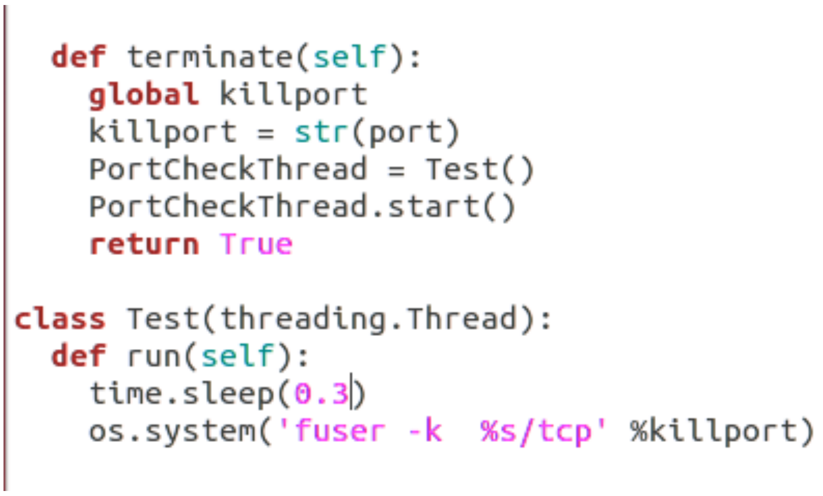
```
/&&list_nodes
NOTE get if value: {u'c': <__main__.FileNode instance at 0x7f8b039e1488>}
NOTE get_node: <__main__.FileNode instance at 0x7f8b039e1488>
get_node: c.txt
NOTE get key:
/c&&list_nodes
NOTE get if value: {u'c.txt': <__main__.FileNode instance at 0x7f8b039e1560>}
NOTE get key:
/c&&list_nodes
NOTE get if value: {u'c.txt': <__main__.FileNode instance at 0x7f8b039e1560>}
NOTE contains_node if name: [u'c.txt']
NOTE get key:
/c&&list_nodes
NOTE get if value: {u'c.txt': <__main__.FileNode instance at 0x7f8b039e1560>}
NOTE get_node: <__main__.FileNode instance at 0x7f8b039e1560>
NOTE file_node getattr: <__main__.FileNode instance at 0x7f8b039e1560>
NOTE get key:
/c/c.txt&&meta
NOTE get if value: {'st_ctime': 1449462922.173363, 'st_mtime': 1449462922.279401
3, 'st_nlink': 1, 'st_mode': 33204L, 'st_size': 6, 'st_atime': 1449462922.279401
3}
```

```
ying@ying-VirtualBox: ~/fusepy/project/fusemount/c
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ cd fusemount
ying@ying-VirtualBox:~/fusepy/project/fusemount$ mkdir c
ying@ying-VirtualBox:~/fusepy/project/fusemount$ cd c
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$ touch c.txt
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$ echo "hello" >>c
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$ cat c.txt
hello
ying@ying-VirtualBox:~/fusepy/project/fusemount/c$
```

Figure 19 Test Read Request

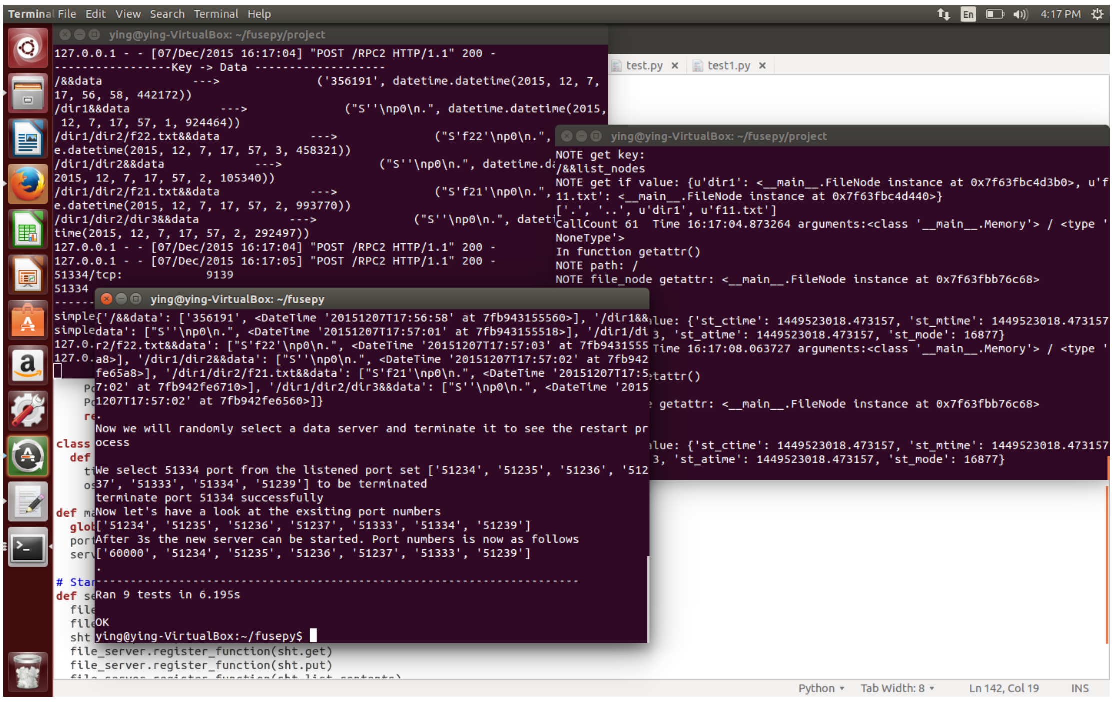
G. General test based on standard test.py

In this test, we used the standard test file provided by our TA, namely unitest.py. We add listing content, corrupting data and terminate 1 server in this file. In order to test for all possible cases, I choose random data server that is register currently to test these three functions. Selecting a random port number can be done in the same way as in B3. The three functions has been introduced before. So all we need is to use RPC to call these functions and show the result. Since we use kill process method to stop a server, we need to implement a thread method to delay the kill process before client RPC can receive a return value. Thus we changed terminate function a little as shown in Figure 20 below. Result is shown in Figure 21.



```python
  def terminate(self):
    global killport
    killport = str(port)
    PortCheckThread = Test()
    PortCheckThread.start()
    return True

class Test(threading.Thread):
  def run(self):
    time.sleep(0.3)
    os.system('fuser -k  %s/tcp' %killport)
```

Figure 20 Terminate Function



```
127.0.0.1 - - [07/Dec/2015 16:17:04] "POST /RPC2 HTTP/1.1" 200 -
-----------------Key -> Data -------------------
/&&data              --->                ('356191', datetime.datetime(2015, 12, 7,
17, 56, 58, 442172))
/dir1&&data              --->                ("S''\np0\n.", datetime.datetime(2015,
 12, 7, 17, 57, 1, 924464))
/dir1/dir2/f22.txt&&data              --->                ("S'f22'\np0\n.",
e.datetime(2015, 12, 7, 17, 57, 3, 458321))
/dir1/dir2&&data              --->                ("S''\np0\n.", datetime.da
2015, 12, 7, 17, 57, 2, 105340))
/dir1/dir2/f21.txt&&data              --->                ("S'f21'\np0\n.",
e.datetime(2015, 12, 7, 17, 57, 2, 993770))
/dir1/dir2/dir3&&data              --->                ("S''\np0\n.", dateti
time(2015, 12, 7, 17, 57, 2, 292497))
127.0.0.1 - - [07/Dec/2015 16:17:04] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 16:17:05] "POST /RPC2 HTTP/1.1" 200 -
51334/tcp:             9139
51334
```

```
NOTE get key:
/&&list_nodes
NOTE get if value: {u'dir1': <__main__.FileNode instance at 0x7f63fbc4d3b0>, u'f
11.txt': <__main__.FileNode instance at 0x7f63fbc4d440>}
['.', '..', u'dir1', u'f11.txt']
CallCount 61  Time 16:17:04.873264 arguments:<class '__main__.Memory'> / <type '
NoneType'>
In function getattr()
NOTE path: /
NOTE file_node getattr: <__main__.FileNode instance at 0x7f63fbb76c68>

lue: {'st_ctime': 1449523018.473157, 'st_mtime': 1449523018.473157
3, 'st_atime': 1449523018.473157, 'st_mode': 16877}
Time 16:17:08.063727 arguments:<class '__main__.Memory'> / <type '

tattr()

 getattr: <__main__.FileNode instance at 0x7f63fbb76c68>

lue: {'st_ctime': 1449523018.473157, 'st_mtime': 1449523018.473157
3, 'st_atime': 1449523018.473157, 'st_mode': 16877}
```

```
{'/&&data': ['356191', <DateTime '20151207T17:56:58' at 7fb943155560>], '/dir1&&
data': ["S''\np0\n.", <DateTime '20151207T17:57:01' at 7fb943155518>], '/dir1/di
r2/f22.txt&&data': ["S'f22'\np0\n.", <DateTime '20151207T17:57:03' at 7fb9431555
a8>], '/dir1/dir2&&data': ["S''\np0\n.", <DateTime '20151207T17:57:02' at 7fb942
fe65a8>], '/dir1/dir2/f21.txt&&data': ["S'f21'\np0\n.", <DateTime '20151207T17:5
7:02' at 7fb942fe6710>], '/dir1/dir2/dir3&&data': ["S''\np0\n.", <DateTime '2015
1207T17:57:02' at 7fb942fe6560>]}
.
Now we will randomly select a data server and terminate it to see the restart pr
ocess

We select 51334 port from the listened port set ['51234', '51235', '51236', '512
37', '51333', '51334', '51239'] to be terminated
terminate port 51334 successfully
Now let's have a look at the exsiting port numbers
['51234', '51235', '51236', '51237', '51333', '51334', '51239']
After 3s the new server can be started. Port numbers is now as follows
['60000', '51234', '51235', '51236', '51237', '51333', '51239']
.
----------------------------------------------------------------------
Ran 9 tests in 6.195s

OK
ying@ying-VirtualBox:~/fusepy$
```

```
  file_server.register_function(sht.get)
  file_server.register_function(sht.put)
```

Terminal File Edit View Search Terminal Help
4:17 PM
ying@ying-VirtualBox: ~/fusepy/project
ying@ying-VirtualBox: ~/fusepy
test.py
test1.py
Python
Tab Width: 8
Ln 142, Col 19
INS

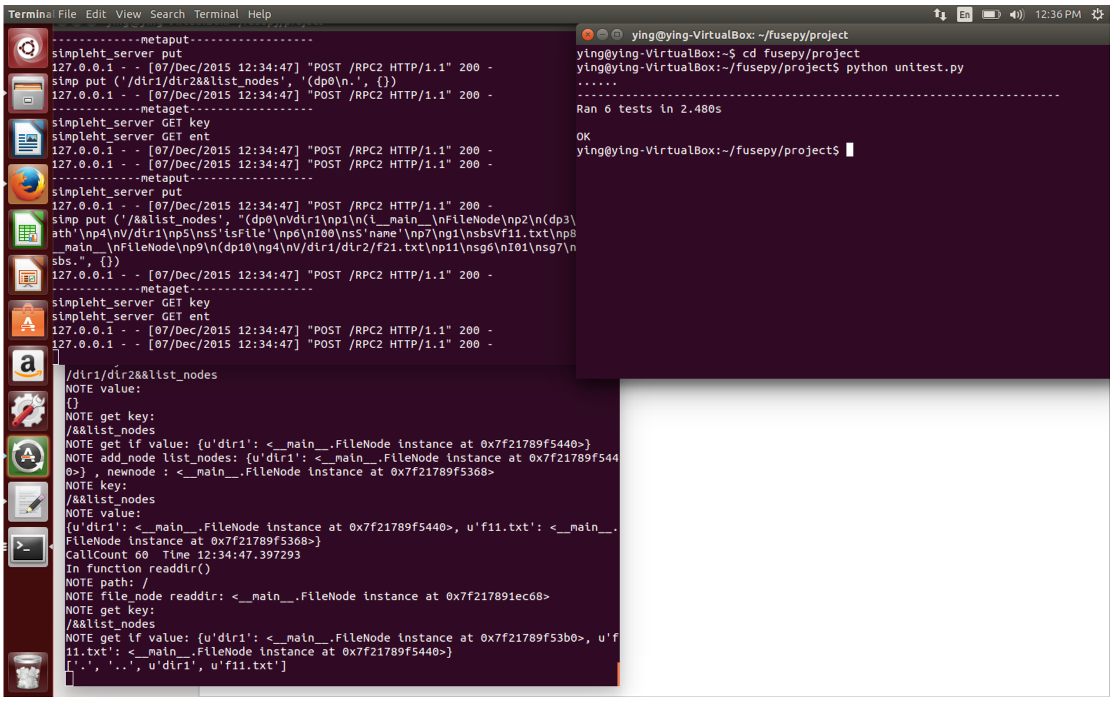
Figure 21 Standard Test

IV. Evaluation Mechanism

We evaluate our system in two aspects, both in functionality and performance.
As for functionality aspect, we use the test file in homework3 to test whether our system can achieve the 6 tests listed in the file and the time spent to pass all the tests. We also compare the run time with the system with one single client and one single server as we did before.
In terms of performance aspect, we would compare performance of different approaches. In our system, for example, using bash file to start servers compared to using threads.

V. Evaluation

To test the functionality of our system using unitest.py, we first start mediator.py, then run the file system, and start unitest.py next to see how our system works. As shown in figure 22 below, we can see our system pass all the six tests of file system(such as read file, write file, remove file, etc.) And the total execution time is 2.480s.


Terminal File Edit View Search Terminal Help
12:36 PM
ying@ying-VirtualBox: ~/fusepy/project

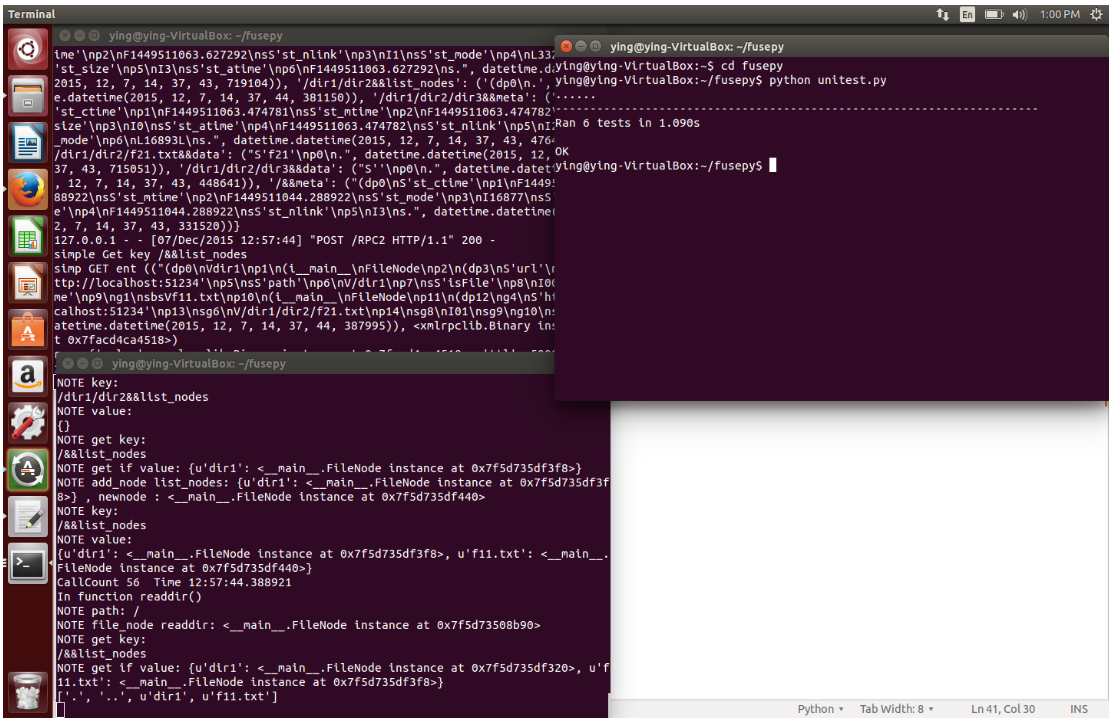
```
-------------metaput------------------
simpleht_server put
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
simp put ('/dir1/dir2&&list_nodes', '(dp0\n.', {})
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
-------------metaget------------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
-------------metaput------------------
simpleht_server put
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
simp put ('/&&list_nodes', "(dp0\nVdir1\np1\n(i__main__\nFileNode\np2\n(dp3\
ath'\np4\nV/dir1\np5\nsS'isFile'\np6\nI00\nsS'name'\np7\ng1\nsbsVf11.txt\np8
__main__\nFileNode\np9\n(dp10\ng4\nV/dir1/dir2/f21.txt\np11\nsg6\nI01\nsg7\n
sbs.", {})
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
-------------metaget------------------
simpleht_server GET key
simpleht_server GET ent
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [07/Dec/2015 12:34:47] "POST /RPC2 HTTP/1.1" 200 -
```

```
ying@ying-VirtualBox:~$ cd fusepy/project
ying@ying-VirtualBox:~/fusepy/project$ python unitest.py
......
----------------------------------------------------------------------
Ran 6 tests in 2.480s

OK
ying@ying-VirtualBox:~/fusepy/project$
```

```
/dir1/dir2&&list_nodes
NOTE value:
{}
NOTE get key:
/&&list_nodes
NOTE get if value: {u'dir1': <__main__.FileNode instance at 0x7f21789f5440>}
NOTE add_node list_nodes: {u'dir1': <__main__.FileNode instance at 0x7f21789f544
0>} , newnode : <__main__.FileNode instance at 0x7f21789f5368>
NOTE key:
/&&list_nodes
NOTE value:
{u'dir1': <__main__.FileNode instance at 0x7f21789f5440>, u'f11.txt': <__main__.
FileNode instance at 0x7f21789f5368>}
CallCount 60  Time 12:34:47.397293
In function readdir()
NOTE path: /
NOTE file_node readdir: <__main__.FileNode instance at 0x7f217891ec68>
NOTE get key:
/&&list_nodes
NOTE get if value: {u'dir1': <__main__.FileNode instance at 0x7f21789f53b0>, u'f
11.txt': <__main__.FileNode instance at 0x7f21789f5440>}
['.', '..', u'dir1', u'f11.txt']
```

Figure 22

Then we run the system with one single server and single client as before, we can see from figure in top right corner that the total execution time is 1.090s, ss shown in figure 23.



```
ime'\np2\nF1449511063.627292\nsS'st_nlink'\np3\nI1\nsS'st_mode'\np4\nL332
'st_size'\np5\nI3\nsS'st_atime'\np6\nF1449511063.627292\ns.", datetime.da
2015, 12, 7, 14, 37, 43, 719104)), '/dir1/dir2&&list_nodes': ('(dp0\n.',
e.datetime(2015, 12, 7, 14, 37, 44, 381150)), '/dir1/dir2/dir3&&meta': ('
'st_ctime'\np1\nF1449511063.474781\nsS'st_mtime'\np2\nF1449511063.474782'
size'\np3\nI0\nsS'st_atime'\np4\nF1449511063.474782\nsS'st_nlink'\np5\nI2
_mode'\np6\nL16893L\ns.", datetime.datetime(2015, 12, 7, 14, 37, 43, 476
/dir1/dir2/f21.txt&&data': ("S'f21'\np0\n.", datetime.datetime(2015, 12,
37, 43, 715051)), '/dir1/dir2/dir3&&data': ("S''\np0\n.", datetime.dateti
, 12, 7, 14, 37, 43, 448641)), '/&&meta': ("(dp0\nS'st_ctime'\np1\nF14495
88922\nsS'st_mtime'\np2\nF1449511044.288922\nsS'st_mode'\np3\nI16877\nsS'
e'\np4\nF1449511044.288922\nsS'st_nlink'\np5\nI3\ns.", datetime.datetime(
2, 7, 14, 37, 43, 331520))}
127.0.0.1 - - [07/Dec/2015 12:57:44] "POST /RPC2 HTTP/1.1" 200 -
simple Get key /&&list_nodes
simp GET ent (("(dp0\nVdir1\np1\n(i__main__\nFileNode\np2\n(dp3\nS'url'\n
ttp://localhost:51234'\np5\nsS'path'\np6\nV/dir1\np7\nsS'isFile'\np8\nI00
me'\np9\ng1\nsbsVf11.txt\np10\n(i__main__\nFileNode\np11\n(dp12\ng4\nS'ht
calhost:51234'\np13\nsg6\nV/dir1/dir2/f21.txt\np14\nsg8\nI01\nsg9\ng10\ns
atetime.datetime(2015, 12, 7, 14, 37, 44, 387995)), <xmlrpclib.Binary ins
t 0x7facd4ca4518>)
```

```
ying@ying-VirtualBox:~$ cd fusepy
ying@ying-VirtualBox:~/fusepy$ python unitest.py
......
----------------------------------------------------------------------
Ran 6 tests in 1.090s

OK
ying@ying-VirtualBox:~/fusepy$
```

```
NOTE key:
/dir1/dir2&&list_nodes
NOTE value:
{}
NOTE get key:
/&&list_nodes
NOTE get if value: {u'dir1': <__main__.FileNode instance at 0x7f5d735df3f8>}
NOTE add_node list_nodes: {u'dir1': <__main__.FileNode instance at 0x7f5d735df3f
8>} , newnode : <__main__.FileNode instance at 0x7f5d735df440>
NOTE key:
/&&list_nodes
NOTE value:
{u'dir1': <__main__.FileNode instance at 0x7f5d735df3f8>, u'f11.txt': <__main__.
FileNode instance at 0x7f5d735df440>}
CallCount 56  Time 12:57:44.388921
In function readdir()
NOTE path: /
NOTE file_node readdir: <__main__.FileNode instance at 0x7f5d73508b90>
NOTE get key:
/&&list_nodes
NOTE get if value: {u'dir1': <__main__.FileNode instance at 0x7f5d735df320>, u'f
11.txt': <__main__.FileNode instance at 0x7f5d735df3f8>}
['.', '..', u'dir1', u'f11.txt']
```

Figure 23

We can see from the two figures above that execution time of a system with one single client and one single server is nearly half of the time of our fault tolerant system. The reason is we have several replicas of data servers instead of one, so putting data in all these servers cost more time. In addition, our fault tolerant system has to deal with server crash and data corruption, so the mediator has to compare all the values the data servers retrieve and return the correct one, meanwhile, the mediator is responsible for restarting a new server in case a server crash. These procedures all cost more time than the simple system we did before.
Thus we can conclude that a fault tolerant system will be more reliable than a single client-server system. But it has to sacrifice execution time as trade-off.

In terms of performance aspect, we compare two different approaches to start servers. We use both bash file and threads to achieve that, and compare the performance of the two approaches. Running threads is more efficient, while using bash is similar to write commands in terminal and will take more time.

VI. Potential Issues

In our design, there are following features:

i) There is one single client, one mediator, one meta server and several data servers. The mediator is responsible for forwarding data to data servers, as well as compare and retrieve the correct value of data servers.

ii) We separate "put" and "get" functions of "meta" and "data" in file system, which simplifies the job of mediator, it receive "meta" and "data" information separately and forwards them to meta server and data servers respectively.

iii) We write bash file in mediator to start servers automatically.

iv) The mediator monitors port numbers of servers periodically, when it finds there is port number gone, which means server crashes, immediately it will restart a new server and put latest the information into it. Our system can also deal with multiple servers crash.

v) The mediator also acts as a comparator of all the values data servers retrieve. We calculate the frequency of each value and finds the most common one as the correct value. Since we assume there is only one data server corruption in this project, the mediator can always retrieve the correct value.

vi) When client sends read request, we randomly pick Qr read data servers, which is more flexible and similar to practical system.

vii) On the data server side, we use commands module to kill a server process, just like open a terminal and write kill command, which is much simpler than most terminate methods online.

However, there are also some issues in this design, the system we build is sort of ideal. There may exist following problems:

i) In this project we assume the meta server never crashes, which is impractical in life. If the meta server stop working, there will also be huge loss of data. Dealing with the crash of meta server is more complicated.

ii) The way we return the correct value of data server to client is to compare and return the most common value. In this project, this is feasible since we assume there is only one data server corruption. However, in real life, there could be several corrupted data servers, so retrieving the most common value would still be wrong result in case that the frequency of correct value and wrong value is the same or the wrong result is the most common one.

iii) In this project, we use bash file to start servers, if we use threads instead, the system will work faster. Since what we use needs to call "os.system" function, which is similar to write commands in terminal, this will lead to system run relatively slower.

VII. Reference

[1] http://www.cs.colostate.edu/~malaiya/530/somani.pdf
[2] http://www2.cs.uidaho.edu/~krings/CS449/Notes.S13/449-13-01.pdf

VIII. Appendix

In our project, we use port number to distinguish data server and meta server, therefore our code files includes Filesystem.py, mediator.py, simpleht_server.py and unitest.py. For any user, you just need to run *python mediator.py <Qr> <Qw> <meta_port> <data_port1>….<data_portN>* first and then run *python FileSystem.py*

<mount-point >. If you want to test, modify the corresponding path in unitest.py and then run it.