EEL 5737 Project Design
Group Member

Fangyu Zhou UFID: 75638108
Yujia Zhang UFID: 66366383

Problem Solved:

In a distributed system, there are several problems we need to solve.

1) How to store path in dataserver. Normally store path string will raise error when same path stored in same server. Therefore, we use a hash of both path and block information to store the path in the server.

2) How to store data in shelve file in data server. In the beginning, I try to initially open a shelve file for read and write. But this doesn’t work because if the server was shut down by interruption, the shelve file will close without save. Therefore, I design to open and close shelve file in operations to ensure the synchronization.

3) How to add retry during connection loss. My first idea is to add retry in read and write method but there are too much to modify. Therefore, I try to add the retry method in put() and get() RPC call and it works fine.

Meta Server:

Meta Server in this project is not changed from the last home work.

Data Server:

Data Server is changed to support the recovery for the data. In this part I use ‘shelve’ library to store the file in FUSE to the local disk for persistence storage. To achieve this, for any operation, the data server will open the shelve file before the operation and do the operations in the shelve file, then close the shelve file. Therefore, every operation will synchronize the shelve file.

Besides, for each server I store a checksum shelve file for correctness check.

When initializing, I create the new shelve files if not exist.

```
def __init__(self, port):
    self.filename = "dataserver" + str(int(port))
    self.checksumfile = "checksumfile" + str(int(port))
    # self.data = shelve.open(filename, writeback = True)
```

Data Server (Cont):

for put() operation, I not only store original file into the server but also the md5 code for the original file into the checksum file.

```
def put(self, key, value):
    # print 'putting', key, value
    # Remove expired entries
    data = shelve.open(self.filename)
    checksum = shelve.open(self.checksumfile)
    data[key.data] = value.data
    checksum[key.data] = hashlib.md5(value.data).hexdigest()
    data.close()
    checksum.close()
    return True
```

Data Server (Cont):

for get() operation, I compare the original file with its checksum, if they match, return the value, else, return an empty string for recovery operation.

```
def get(self, key):
    # print 'getting', key
    # Default return value
    data = shelve.open(self.filename)
    checksum = shelve.open(self.checksumfile)

    rv = {}
    # If the key is in the data structure, return properly formated results
    key = key.data
    if key in data and key in checksum:
        if checksum[key] == hashlib.md5(data[key]).hexdigest():
            print ("**********************************" + checksum[key])
            print ("**********************************" + hashlib.md5(data[key]).hexdigest())
            rv = Binary(data[key])
        else:
            rv = Binary('')
    else:
        rv = Binary('')

    data.close()
    checksum.close()
    return rv
```

Client:

To support distributed file system, I use sum of ord of path string to hash the path, adding block info to represent a path so that as long as the number of the replica is less than the number of dataserver, there will not be the same path name for the same file in one dataserver.

```
def hashpath(self, path):
    return sum(ord(i) for i in path)
```

to get data from server, first we need to keep trying to read block data for each data server which may store the block data in case any server is down. So that, even if there is only one replica available, it will be retrieved.

In normal case, the client will retrieve the data from a random data server that may contain the block. If there is a data crash, the server will return an empty string. In this case, the client will try to read block from adjacent server and recover the data to the crashed one.

to put or purge data, for each block it will send 3 put RPC call to store or delete 3 replica in 3 adjacent server.

Client (Cont):

Get data method when data is unavailable on current server:

```
for i in range(len(self.dataserv)):
    if self.dataserv[(phash + blk + k + i) % len(self.dataserv)].get(
            Binary(str(blk) + path)).data != '':
        blkdata = self.dataserv[(phash + blk + k + i) % len(self.dataserv)].get(
            Binary(str(blk) + path)).data
        self.dataserv[(phash + blk + k) % len(self.dataserv)].put(Binary(str(blk) + path),
                                                                  Binary(blkdata))
```

Keep reading data while connection loss:

```
k = random.randint(0, replicaNum - 1)
while True:
    try:
        blkdata = self.dataserv[(phash + blk + k) % len(self.dataserv)].get(Binary(str(blk) + path)).data
    except:
        k = (k + 1) % len(self.dataserv)
        continue
```

Corrupt Function:

In corrupt function, I simply simulate the block 0 crashed in a path file for every non-adjacent server.

The command line should be:

python corrupt.py <path> <dataserverport1> <dataserverport2>…

```
def main():
    filename = argv[1]
    dataserv = argv[2:]
    print dataserv
    for i in range(0, len(dataserv), 2):
        s = shelve.open('dataserver' + dataserv[i])
        print '~~~before deleting at dataserver' + dataserv[i] + ': '
        print s
        if str(0) + filename in s:
            del s[str(0) + filename]
        print '~~~after deleting at dataserver' + dataserv[i] + ': '
        print s
        s.close()


if __name__ == "__main__":
  if len(argv) < 3:
        print('usage: %s <path> <dataport1> <dataport2> ..' % argv[0])
        exit(1)
  main()
```

Test:

```
fangyu@Viscount:~/fusepy/proj/fusemount$ echo "qwertyuiop[]" -> 1.txt
fangyu@Viscount:~/fusepy/proj/fusemount$ echo "asdfghjkl;'" -> 2.txt
bash: echo: write error: Bad address
fangyu@Viscount:~/fusepy/proj/fusemount$ echo "asdfghjkl;'" -> 2.txt
fangyu@Viscount:~/fusepy/proj/fusemount$ echo "zxcvbnm,./" -> 3.txt
fangyu@Viscount:~/fusepy/proj/fusemount$ cd ..
fangyu@Viscount:~/fusepy/proj$ ls
checksumfile2222  dataserver4444        distr_retry_write.py  memory.py
checksumfile3333  dataserver5555        examples              metaserver
checksumfile4444  dataserver.py         fusell.py             metaserver.py
checksumfile5555  distr (copy).py       fusemount             README
corrupt.py        distr.py              fuse.py               README.rst
dataserver2222    distr_read (copy).py  fuse.pyc              setup.py
dataserver3333    distr_read.py         MANIFEST.in           test
fangyu@Viscount:~/fusepy/proj$ rm dataserver2222
fangyu@Viscount:~/fusepy/proj$ cd fusemount
fangyu@Viscount:~/fusepy/proj/fusemount$ cat 1.txt
qwertyuiop[] -
fangyu@Viscount:~/fusepy/proj/fusemount$ cat 2.txt
asdfghjkl;' -
fangyu@Viscount:~/fusepy/proj/fusemount$ cat 3.txt
zxcvbnm,./ -
fangyu@Viscount:~/fusepy/proj/fusemount$
```

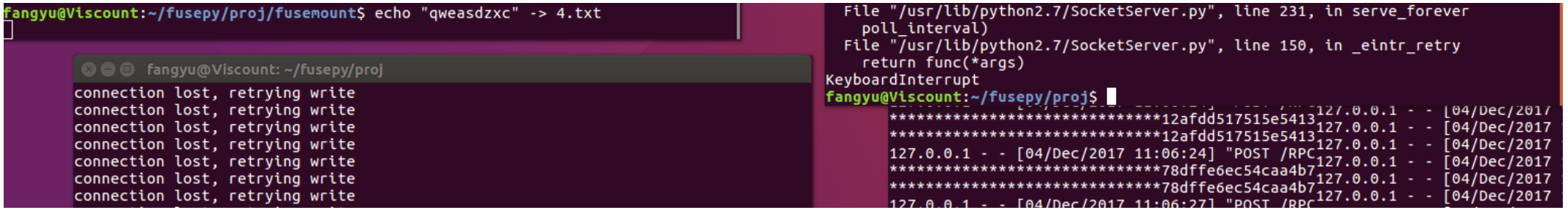
This is to simulate the server crashed with file deleted and restart. The files are also can be retrieved.

Test:

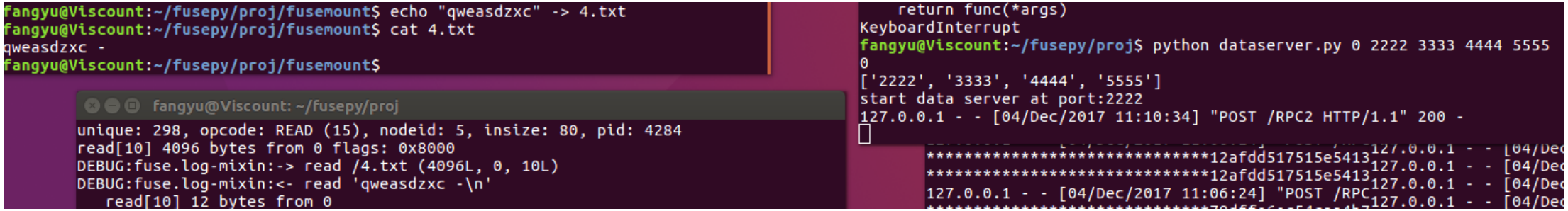
This is to implement corrupt function and test the recovery of the server

```
fangyu@Viscount:~/fusepy/proj/fusemount$ cd ..
fangyu@Viscount:~/fusepy/proj$ python corrupt.py /1.txt 2222 3333 4444 5555
['2222', '3333', '4444', '5555']
~~~before deleting at dataserver2222:
{'0/1.txt': 'qwertyui', '0/3.txt': 'zxcvbnm,', '1/1.txt': 'op[] -\n', '1/4.txt':
 'c -\n', '1/2.txt': "l;' -\n", '0/2.txt': 'asdfghjk'}
~~~after deleting at dataserver2222:
{'1/2.txt': "l;' -\n", '0/3.txt': 'zxcvbnm,', '1/1.txt': 'op[] -\n', '1/4.txt':
'c -\n', '0/2.txt': 'asdfghjk'}
~~~before deleting at dataserver4444:
{'0/1.txt': 'qwertyui', '0/3.txt': 'zxcvbnm,', '1/4.txt': 'c -\n', '1/3.txt': '.
/ -\n', '0/4.txt': 'qweasdzx', '1/2.txt': "l;' -\n"}
~~~after deleting at dataserver4444:
{'1/2.txt': "l;' -\n", '0/4.txt': 'qweasdzx', '0/3.txt': 'zxcvbnm,', '1/4.txt':
'c -\n', '1/3.txt': './ -\n'}
fangyu@Viscount:~/fusepy/proj$ python corrupt.py /2.txt 2222 3333 4444 5555
['2222', '3333', '4444', '5555']
~~~before deleting at dataserver2222:
{'1/2.txt': "l;' -\n", '0/3.txt': 'zxcvbnm,', '1/1.txt': 'op[] -\n', '1/4.txt':
'c -\n', '0/2.txt': 'asdfghjk'}
~~~after deleting at dataserver2222:
{'1/2.txt': "l;' -\n", '0/3.txt': 'zxcvbnm,', '1/1.txt': 'op[] -\n', '1/4.txt':
'c -\n'}
~~~before deleting at dataserver4444:
{'1/2.txt': "l;' -\n", '0/4.txt': 'qweasdzx', '0/3.txt': 'zxcvbnm,', '1/4.txt':
'c -\n', '1/3.txt': './ -\n'}
~~~after deleting at dataserver4444:
{'1/2.txt': "l;' -\n", '0/4.txt': 'qweasdzx', '0/3.txt': 'zxcvbnm,', '1/4.txt':
'c -\n', '1/3.txt': './ -\n'}
fangyu@Viscount:~/fusepy/proj$ cd fusemount
fangyu@Viscount:~/fusepy/proj/fusemount$ cat 1.txt
qwertyuiop[] -
fangyu@Viscount:~/fusepy/proj/fusemount$ cat 2.txt
asdfghjkl;' -
fangyu@Viscount:~/fusepy/proj/fusemount$
```

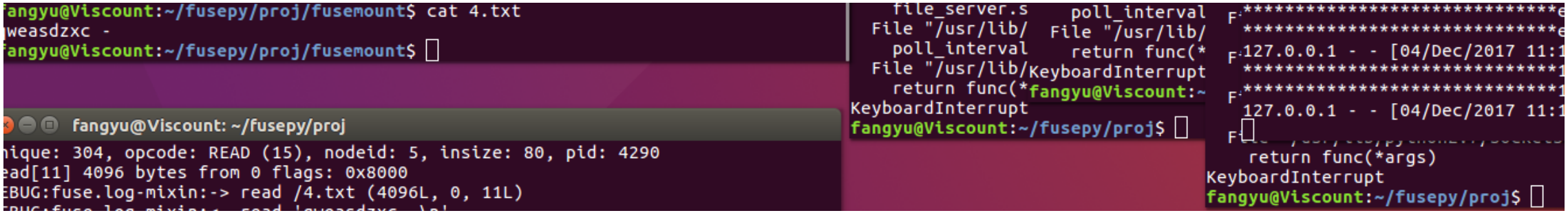
Test:



This is to simulate when one data server is down the write operation will keep trying write until the server is restarted.



Test:



This is to simulate when some servers are down, as long as there is a replica available, the read will work.

Problem:

This project has a problem, when the number of data server is big, when we do some operations, the request may be blocked. This may because when we are about to do some request, that thread may be occupied and the access is locked.

Work Assignment:

We do the project by discussing and code together.

EEL 5737 Project Design for Extra Credits
Group Member

Fangyu Zhou UFID: 75638108
Yujia Zhang UFID: 66366383

What’s extra:

In this part, we move the server to the AWS EC2 instance to support remote distributed file system.

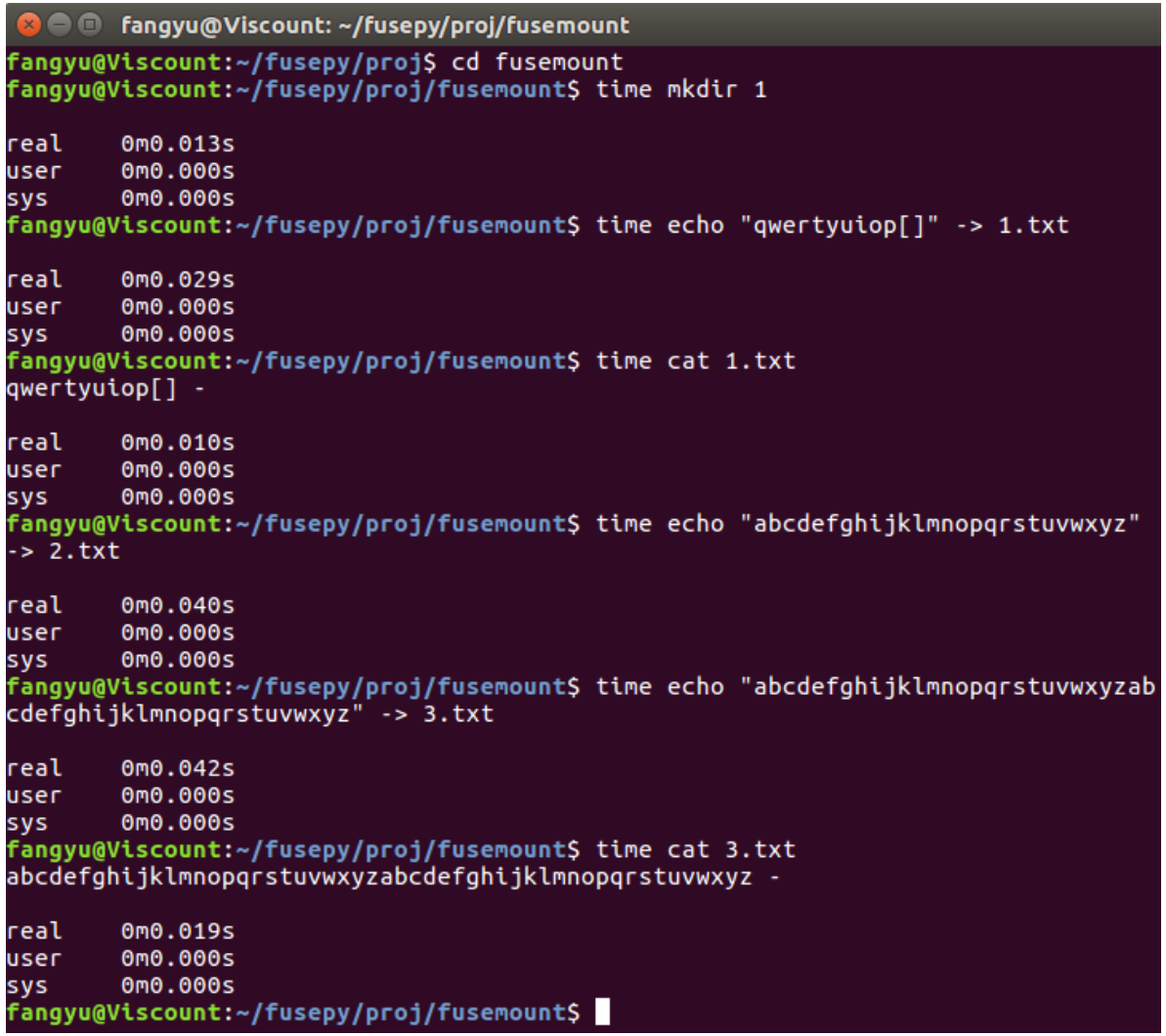
How to measure latency:

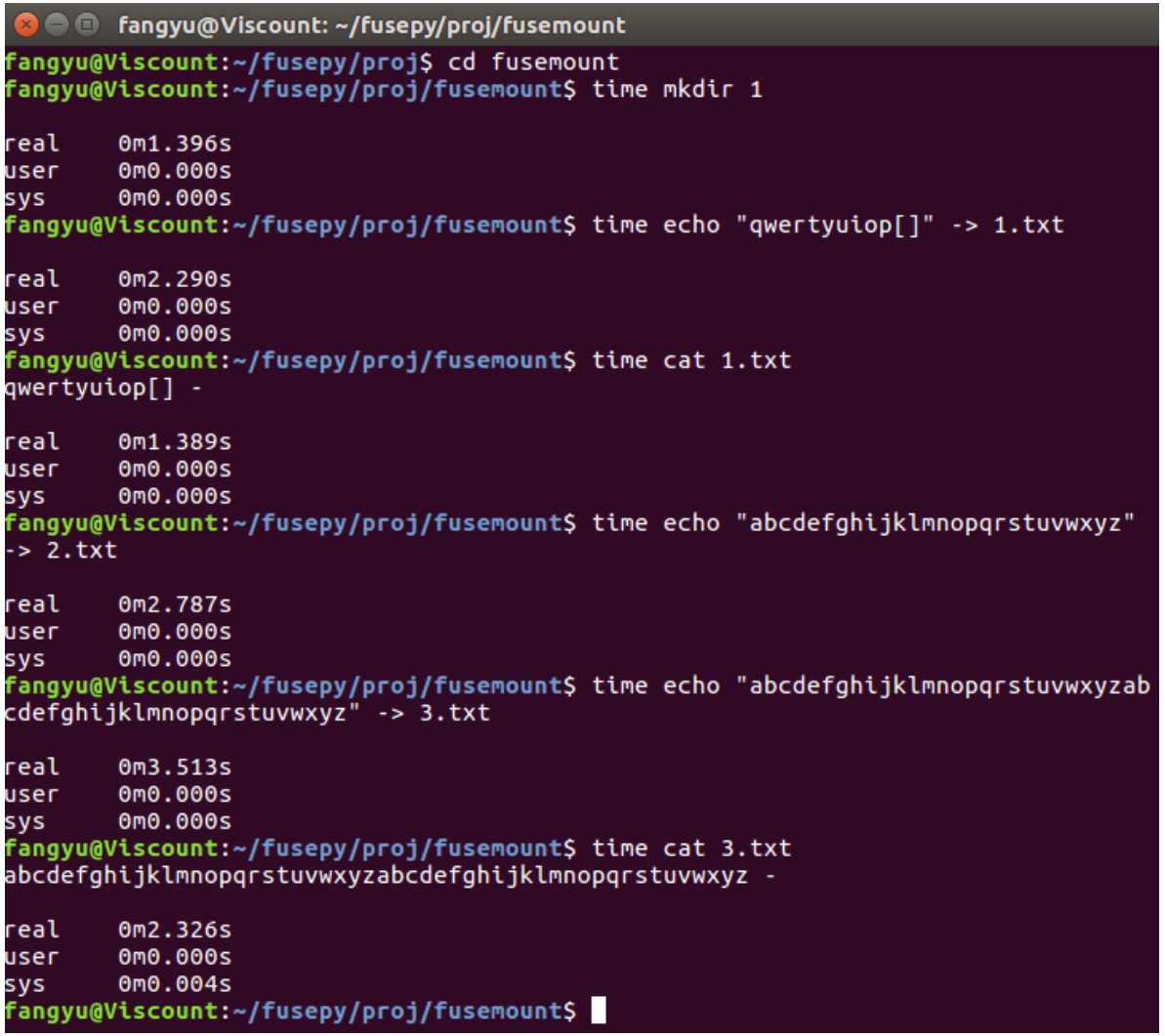
To measure latency in RPC, I use time system call to collect the running time of each operation because its measure the same period time for the same command in either local RPC or remote RPC. For operations, I choose mkdir, echo, cat to run because these operations is typical and their conditions are easy to control.

Some findings:

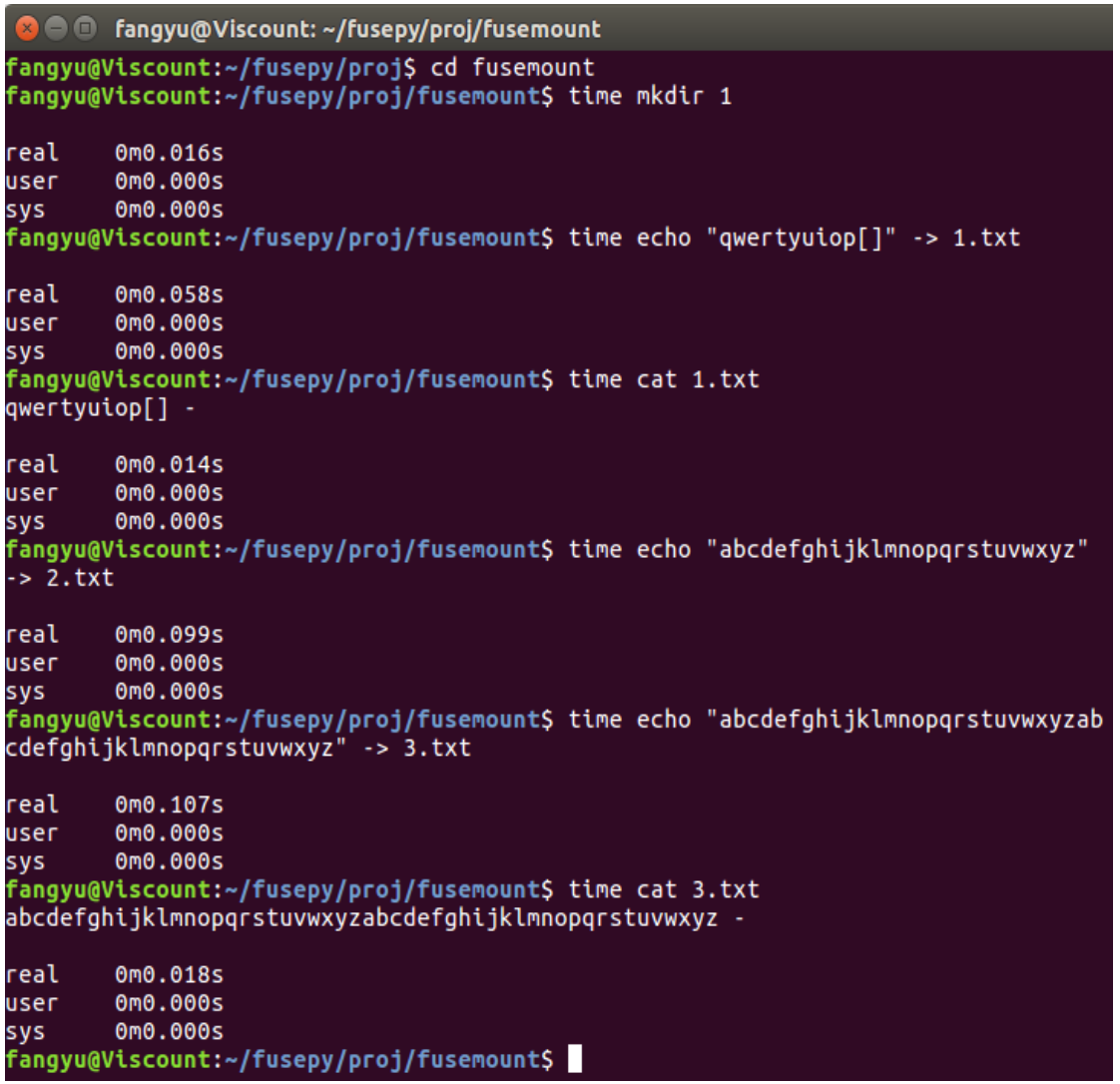
In some network conditions, the connection may be congested if the number of server is big. For example, in some bad network conditions, I start 2 dataservers and it works well. When I start 3 or more dataservers, in the cache clear part, the connection get congested and the file system will be down and return time out failure.

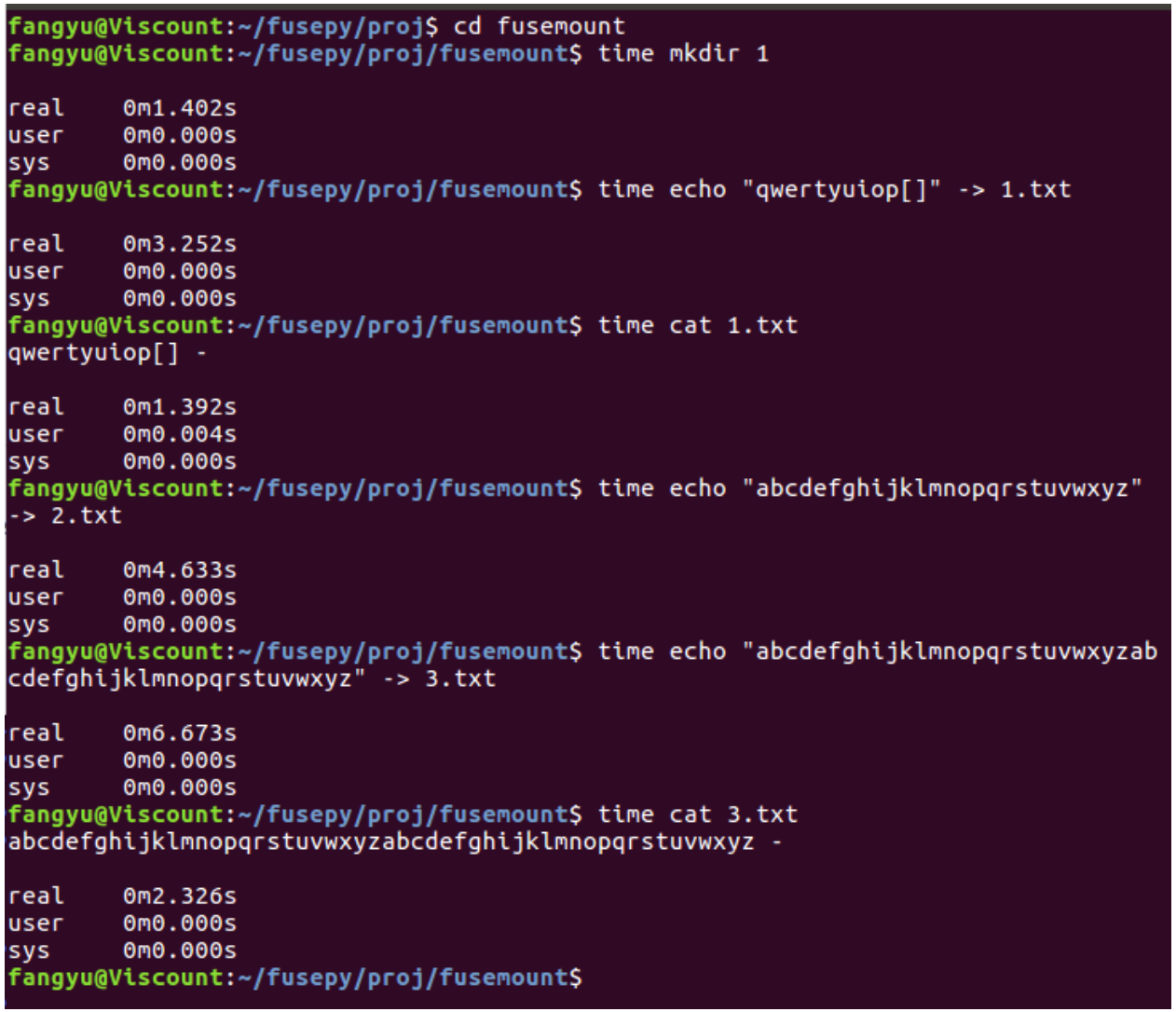
Test: I did exactly the same command for both localhost RPC and Remote RPC for 1 dataserver and 1 replica for each block. The result is in below.





Test: I did exactly the same command for both localhost RPC and Remote RPC for 4 dataservers and 3 replicas for each block. The result is in below.





Analysis:

For the single-server system, the running time of each operation of Remote RPC is much more than the localhost RPC. The more data block I was writing, the more time it takes.

The best operation to compute latency is read because it contains less RPC than others.

The total time of read 3.txt is 2.326s on Remote RPC and 0.018s on localhost RPC, 3.txt contains 7 blocks, so in the read operation, there are 14 RPCs (one getmeta() and one getdata() per block). There fore, the latency is about $\frac{2.326 - 0.018}{14} \approx 0.164s = 164ms$ (regardless of some other system calls).

For the multi-server system, the write operation takes more time because it has to write duplicas in adjacent data servers. But the read time is the same as single server since read only need one replica on one server.

Summary:

In this remote file system, there are lots of things that can be improved. Like higher internet access speed, more concise code for both server and client.

Corrupt:

```
    import shelve
from sys import argv


def main():
    filename = argv[1]
    dataserv = argv[2:]
    print dataserv
    for i in range(0, len(dataserv), 2):
        s = shelve.open('dataserver' + dataserv[i])
        print '~~~before deleting at dataserver' + dataserv[i] + ': '
        print s
        if str(0) + filename in s:
            del s[str(0) + filename]
        print '~~~after deleting at dataserver' + dataserv[i] + ': '
        print s
        s.close()


if __name__ == "__main__":
    if len(argv) < 3:
        print('usage: %s <path> <dataport1> <dataport2> ..' % argv[0])
        exit(1)
    main()
```

Metaserver:

```
    #!/usr/bin/env python
"""
Author: David Wolinsky
Version: 0.02

Description:
The XmlRpc API for this library is:
  get(base64 key)
    Returns the value and ttl associated with the given key using a dictionary
      or an empty dictionary if there is no matching key
    Example usage:
      rv = rpc.get(Binary("key"))
      print rv => {"value": Binary, "ttl": 1000}
      print rv["value"].data => "value"
  put(base64 key, base64 value, int ttl)
    Inserts the key / value pair into the hashtable, using the same key will
      over-write existing values
    Example usage:  rpc.put(Binary("key"), Binary("value"), 1000)
  print_content()
    Print the contents of the HT
  read_file(string filename)
    Store the contents of the Hahelperable into a file
  write_file(string filename)
    Load the contents of the file into the Hahelperable
"""

import sys, SimpleXMLRPCServer, getopt, pickle, time, threading, xmlrpclib,
unittest
import shelve
from datetime import datetime, timedelta
from xmlrpclib import Binary


# Presents a HT interface
class SimpleHT:
    def __init__(self, port):
        self.filename = "metaserver"
        # self.data = shelve.open(filename, writeback = True)
        self.data = {}

    def count(self):
```

```
        s = shelve.open(self.filename)
        data = s
        s.close()
        return len(data)

    # Retrieve something from the HT
    def get(self, key):
        # print 'getting', key
        # Default return value
        data = shelve.open(self.filename)
        rv = {}
        # If the key is in the data structure, return properly formated results
        key = key.data
        if key in data:
            rv = Binary(data[key])
        data.close()
        return rv

    # Insert something into the HT
    def put(self, key, value):
        # print 'putting', key, value
        # Remove expired entries
        data = shelve.open(self.filename)
        data[key.data] = value.data
        data.close()
        return True

    # Load contents from a file
    def read_file(self, filename):
        f = open(filename.data, "rb")
        self.data = pickle.load(f)
        f.close()
        return True

    # Clear all contents
    def clear(self):
        data = shelve.open(self.filename)
        data.clear()
        data.close()
        return True

    # Delete a file
    def remove(self, key):
```

```
        # print 'removing', key
        data = shelve.open(self.filename)
        if key.data in data:
            del data[key.data]
            data.close()
            return True
        else:
            data.close()
            return False

    # Write contents to a file
    def write_file(self, filename):
        f = open(filename.data, "wb")
        pickle.dump(self.data, f)
        f.close()
        return True

    # Print the contents of the hashtable
    def print_content(self):
        print self.data
        return True


def main():
    #  optlist, args = getopt.getopt(sys.argv[1:], "", ["port=", "test"])
    #  ol={}
    #  for k,v in optlist:
    #    ol[k] = v

    #  port = 51234
    #  if "--port" in ol:
    #    port = int(ol["--port"])
    #  if "--test" in ol:
    #    sys.argv.remove("--test")
    #   unittest.main()
    #    return
    print "Start meta server at port:" + sys.argv[1]
    serve(int(sys.argv[1]))


# Start the xmlrpc server
def serve(port):
    file_server = SimpleXMLRPCServer.SimpleXMLRPCServer(('', port))
```

```
    file_server.register_introspection_functions()
    sht = SimpleHT(port)
    file_server.register_function(sht.clear)
    file_server.register_function(sht.get)
    file_server.register_function(sht.put)
    file_server.register_function(sht.print_content)
    file_server.register_function(sht.read_file)
    file_server.register_function(sht.write_file)
    file_server.register_function(sht.remove)
    file_server.serve_forever()


# Execute the xmlrpc in a thread ... needed for testing
class serve_thread:
    def __call__(self, port):
        serve(port)


# Wrapper functions so the tests don't need to be concerned about Binary blobs
class Helper:
    def __init__(self, caller):
        self.caller = caller

    def put(self, key, val, ttl):
        return self.caller.put(Binary(key), Binary(val), ttl)

    def get(self, key):
        return self.caller.get(Binary(key))

    def write_file(self, filename):
        return self.caller.write_file(Binary(filename))

    def read_file(self, filename):
        return self.caller.read_file(Binary(filename))


class SimpleHTTest(unittest.TestCase):
    def test_direct(self):
        helper = Helper(SimpleHT(port))
        self.assertEqual(helper.get("test"), {}, "DHT isn't empty")
        self.assertTrue(helper.put("test", "test", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test", "Failed to perform
single get")
```

```
        self.assertTrue(helper.put("test", "test0", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test0", "Failed to perform
overwrite")
        self.assertTrue(helper.put("test", "test1", 2), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test1", "Failed to perform
overwrite")
        time.sleep(2)
        self.assertEqual(helper.get("test"), {}, "Failed expire")
        self.assertTrue(helper.put("test", "test2", 20000))
        self.assertEqual(helper.get("test")["value"], "test2", "Store new value")

        helper.write_file("test")
        helper = Helper(SimpleHT(port))

        self.assertEqual(helper.get("test"), {}, "DHT isn't empty")
        helper.read_file("test")
        self.assertEqual(helper.get("test")["value"], "test2", "Load
unsuccessful!")
        self.assertTrue(helper.put("some_other_key", "some_value", 10000))
        self.assertEqual(helper.get("some_other_key")["value"], "some_value",
"Different keys")
        self.assertEqual(helper.get("test")["value"], "test2", "Verify contents")

    # Test via RPC
    def test_xmlrpc(self):
        output_thread = threading.Thread(target=serve_thread(), args=(51234,))
        output_thread.setDaemon(True)
        output_thread.start()

        time.sleep(1)
        helper = Helper(xmlrpclib.Server("http://127.0.0.1:51234"))
        self.assertEqual(helper.get("test"), {}, "DHT isn't empty")
        self.assertTrue(helper.put("test", "test", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test", "Failed to perform
single get")
        self.assertTrue(helper.put("test", "test0", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test0", "Failed to perform
overwrite")
        self.assertTrue(helper.put("test", "test1", 2), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test1", "Failed to perform
overwrite")
        time.sleep(2)
        self.assertEqual(helper.get("test"), {}, "Failed expire")
```

```
        self.assertTrue(helper.put("test", "test2", 20000))
        self.assertEqual(helper.get("test")["value"], "test2", "Store new value")


if __name__ == "__main__":
    main()
```

Dataserver:

```
    #!/usr/bin/env python
"""
Author: David Wolinsky
Version: 0.02

Description:
The XmlRpc API for this library is:
  get(base64 key)
    Returns the value and ttl associated with the given key using a dictionary
      or an empty dictionary if there is no matching key
    Example usage:
      rv = rpc.get(Binary("key"))
      print rv => {"value": Binary, "ttl": 1000}
      print rv["value"].data => "value"
  put(base64 key, base64 value, int ttl)
    Inserts the key / value pair into the hashtable, using the same key will
      over-write existing values
    Example usage:  rpc.put(Binary("key"), Binary("value"), 1000)
  print_content()
    Print the contents of the HT
  read_file(string filename)
    Store the contents of the Hahelperable into a file
  write_file(string filename)
    Load the contents of the file into the Hahelperable
"""


import sys, SimpleXMLRPCServer, getopt, pickle, time, threading, xmlrpclib,
unittest, shelve, hashlib
from datetime import datetime, timedelta
from xmlrpclib import Binary


# Presents a HT interface
class SimpleHT:
    def __init__(self, port):
        self.filename = "dataserver" + str(int(port))
        self.checksumfile = "checksumfile" + str(int(port))
        # self.data = shelve.open(filename, writeback = True)
        self.data = {}

    def count(self):
```

```
        s = shelve.open(self.filename)
        data = s
        s.close()
        return len(data)

    # Retrieve something from the HT
    def get(self, key):
        # print 'getting', key
        # Default return value
        data = shelve.open(self.filename)
        checksum = shelve.open(self.checksumfile)

        rv = {}
        # If the key is in the data structure, return properly formated results
        key = key.data
        if key in data and key in checksum:
            if checksum[key] == hashlib.md5(data[key]).hexdigest():
                print ("******************************" + checksum[key])
                print ("******************************" +
hashlib.md5(data[key]).hexdigest())
                rv = Binary(data[key])
            else:
                rv = Binary('')
        else:
            rv = Binary('')

        data.close()
        checksum.close()
        return rv

    # Insert something into the HT
    def put(self, key, value):
        # print 'putting', key, value
        # Remove expired entries
        data = shelve.open(self.filename)
        checksum = shelve.open(self.checksumfile)
        data[key.data] = value.data
        checksum[key.data] = hashlib.md5(value.data).hexdigest()
        data.close()
        checksum.close()
        return True

    # Load contents from a file
```

```
    def read_file(self, filename):
        f = open(filename.data, "rb")
        self.data = pickle.load(f)
        f.close()
        return True

    # Clear all contents
    def clear(self):
        data = shelve.open(self.filename)
        checksum = shelve.open(self.checksumfile)
        checksum.clear()
        data.clear()
        checksum.close()
        data.close()
        return True

    # Delete a file
    def remove(self, key):
        # print 'removing', key
        data = shelve.open(self.filename)
        checksum = shelve.open(self.checksumfile)
        if key.data in data:
            del data[key.data]
            del checksum[key.data]
            data.close()
            checksum.close()
            return True
        else:
            data.close()
            checksum.close()
            return False

    # Write contents to a file
    def write_file(self, filename):
        f = open(filename.data, "wb")
        pickle.dump(self.data, f)
        f.close()
        return True

    # Print the contents of the hashtable
    def print_content(self):
        print self.data
        return True
```

```
def main():
    optlist, args = getopt.getopt(sys.argv[1:], "", ["port=", "test"])
    index = sys.argv[1]
    para = sys.argv[2:]

    print index
    print para
    port = int(para[int(index)])
    print "start data server at port:" + str(int(port))
    #  ol={}
    #  for k,v in optlist:
    #    ol[k] = v


    #  port = 51234
    #  if "--port" in ol:
    #    port = int(ol["--port"])
    #  if "--test" in ol:
    #    sys.argv.remove("--test")
    #    unittest.main()
    #    return
    serve(port)


# Start the xmlrpc server
def serve(port):
    file_server = SimpleXMLRPCServer.SimpleXMLRPCServer(('', port))
    file_server.register_introspection_functions()
    sht = SimpleHT(port)
    file_server.register_function(sht.clear)
    file_server.register_function(sht.get)
    file_server.register_function(sht.put)
    file_server.register_function(sht.print_content)
    file_server.register_function(sht.read_file)
    file_server.register_function(sht.write_file)
    file_server.register_function(sht.remove)
    file_server.serve_forever()


# Execute the xmlrpc in a thread ... needed for testing
class serve_thread:
```

```
    def __call__(self, port):
        serve(port)


# Wrapper functions so the tests don't need to be concerned about Binary blobs
class Helper:
    def __init__(self, caller):
        self.caller = caller

    def put(self, key, val, ttl):
        return self.caller.put(Binary(key), Binary(val), ttl)

    def get(self, key):
        return self.caller.get(Binary(key))

    def write_file(self, filename):
        return self.caller.write_file(Binary(filename))

    def read_file(self, filename):
        return self.caller.read_file(Binary(filename))


class SimpleHTTest(unittest.TestCase):
    def test_direct(self):
        helper = Helper(SimpleHT(port))
        self.assertEqual(helper.get("test"), {}, "DHT isn't empty")
        self.assertTrue(helper.put("test", "test", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test", "Failed to perform
single get")
        self.assertTrue(helper.put("test", "test0", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test0", "Failed to perform
overwrite")
        self.assertTrue(helper.put("test", "test1", 2), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test1", "Failed to perform
overwrite")
        time.sleep(2)
        self.assertEqual(helper.get("test"), {}, "Failed expire")
        self.assertTrue(helper.put("test", "test2", 20000))
        self.assertEqual(helper.get("test")["value"], "test2", "Store new value")

        helper.write_file("test")
        helper = Helper(SimpleHT(port))
```

```
        self.assertEqual(helper.get("test"), {}, "DHT isn't empty")
        helper.read_file("test")
        self.assertEqual(helper.get("test")["value"], "test2", "Load
unsuccessful!")
        self.assertTrue(helper.put("some_other_key", "some_value", 10000))
        self.assertEqual(helper.get("some_other_key")["value"], "some_value",
"Different keys")
        self.assertEqual(helper.get("test")["value"], "test2", "Verify contents")

    # Test via RPC
    def test_xmlrpc(self):
        output_thread = threading.Thread(target=serve_thread(), args=(51234,))
        output_thread.setDaemon(True)
        output_thread.start()

        time.sleep(1)
        helper = Helper(xmlrpclib.Server("http://127.0.0.1:51234"))
        self.assertEqual(helper.get("test"), {}, "DHT isn't empty")
        self.assertTrue(helper.put("test", "test", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test", "Failed to perform
single get")
        self.assertTrue(helper.put("test", "test0", 10000), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test0", "Failed to perform
overwrite")
        self.assertTrue(helper.put("test", "test1", 2), "Failed to put")
        self.assertEqual(helper.get("test")["value"], "test1", "Failed to perform
overwrite")
        time.sleep(2)
        self.assertEqual(helper.get("test"), {}, "Failed expire")
        self.assertTrue(helper.put("test", "test2", 20000))
        self.assertEqual(helper.get("test")["value"], "test2", "Store new value")


if __name__ == "__main__":
    main()
```

Client:

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

import logging, xmlrpclib, pickle, random

from xmlrpclib import Binary
from collections import defaultdict
from errno import ENOENT, 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

bsize = 8
replicaNum = 3


class Memory(LoggingMixIn, Operations):
    """Implements a hierarchical file system by using FUSE virtual filesystem.
       The file structure and data are stored in local memory in variable.
       Data is lost when the filesystem is unmounted"""

    def __init__(self, mport, dports):
        self.fd = 0
        self.metaserv = xmlrpclib.ServerProxy("http://localhost:" +
str(int(mport)))
        self.dataserv = [xmlrpclib.ServerProxy("http://localhost:" + str(int(i)))
for i in dports]
        self.metaserv.clear()
        for i in self.dataserv:
            i.clear()
        #        now = time()
        self.putmeta('/', dict(st_mode=(S_IFDIR | 0o755), st_ctime=time(),
                               st_mtime=time(), st_atime=time(), st_nlink=2,
files=[]))
        # The key 'files' holds a dict of filenames(and their attributes
        #  and 'files' if it is a directory) under each level
```

```
    def hashpath(self, path):
        return sum(ord(i) for i in path)

    def getmeta(self, path):
        metadata = pickle.loads(self.metaserv.get(Binary(path)).data)
        sleep(0.005)
        return metadata

    def putmeta(self, path, meta):
        return self.metaserv.put(Binary(path), Binary(pickle.dumps(meta)))

    def purgemeta(self, path):
        return self.metaserv.remove(Binary(path))

    def getdata(self, path, blks):
        phash = self.hashpath(path)
        stringlist = []
        for blk in blks:
            k = random.randint(0, replicaNum - 1)
            while True:
                try:
                    blkdata = self.dataserv[(phash + blk + k) %
len(self.dataserv)].get(Binary(str(blk) + path)).data
                except:
                    k = (k + 1) % len(self.dataserv)
                    continue
                #     blkdata = self.dataserv[(phash + blk + k) %
len(self.dataserv)].get(Binary(str(blk) + path)).data
                break

            if blkdata == '':
                while True:
                    try:
                        for i in range(len(self.dataserv)):
                            if self.dataserv[(phash + blk + k + i) %
len(self.dataserv)].get(
                                    Binary(str(blk) + path)).data != '':
                                blkdata = self.dataserv[(phash + blk + k + i) %
len(self.dataserv)].get(
                                    Binary(str(blk) + path)).data
                                self.dataserv[(phash + blk + k) %
len(self.dataserv)].put(Binary(str(blk) + path),
```

```
Binary(blkdata))
                                print "*****************************copy data
from another duplica"
                                break
                            else:
                                print "*************************this blk doesnt
contain this data"
                    except:
                        continue
                    break
            stringlist.append(blkdata)
        return stringlist

    #    def getdata(self, path, blks):
    #        phash = self.hashpath(path)
    #        return [self.dataserv[(phash + blk + random.randint(0,replicaNum -
1)) % len(self.dataserv)].get(Binary(str(blk) + path)).data for blk in blks]

    def putdata(self, path, blks, datablks):
        phash = self.hashpath(path)
        print "+++++++++++++++put"
        print blks
        for i in range(len(blks)):
            for k in range(replicaNum):
                while True:
                    try:
                        self.dataserv[(phash + blks[i] + k) %
len(self.dataserv)].put(Binary(str(blks[i]) + path),

Binary(datablks[i]))
                    except:
                        print "connection lost, retrying write"
                        continue
                    break

    def purgedata(self, path, blks):
        phash = self.hashpath(path)
        #  purged = []
        #  while True:
        #      try:
        #          purged = [self.dataserv[(phash + blk) %
len(self.dataserv)].remove(Binary(str(blk) + path)) for blk in blks]
        #      except:
```

```
        #           print "connection lost"
        #       break
        #         return purged
        print "&&&&&&&&&&&&&&&&&&&&&&&purge"
        print blks
        for i in range(len(blks)):
            for k in range(replicaNum):
                while True:
                    try:
                        self.dataserv[(phash + blks[i] + k) %
len(self.dataserv)].remove(Binary(str(blks[i]) + path))
                    except:
                        print "connection lost, retrying purge"
                        continue
                    break

    def splitpath(self, path):
        childpath = path[path.rfind('/') + 1:]
        parentpath = path[:path.rfind('/')]
        if parentpath == '':
            parentpath = '/'
        return parentpath, childpath

    def chmod(self, path, mode):
        p = self.getmeta(path)
        p['st_mode'] &= 0o770000
        p['st_mode'] |= mode
        self.putmeta(path, p)
        return 0

    def chown(self, path, uid, gid):
        p = self.getmeta(path)
        p['st_uid'] = uid
        p['st_gid'] = gid
        self.putmeta(path)

    def create(self, path, mode):
        ppath, cname = self.splitpath(path)
        p = self.getmeta(ppath)
        p['files'].append(cname)
        self.putmeta(ppath, p)
        self.putmeta(path, dict(st_mode=(S_IFREG | mode), st_nlink=1,
                                st_size=0, st_ctime=time(), st_mtime=time(),
```

```
                                st_atime=time()))
        self.fd += 1
        return self.fd

    def getattr(self, path, fh=None):
        try:
            p = self.getmeta(path)
        except:
            raise FuseOSError(ENOENT)
        return {attr: p[attr] for attr in p.keys() if attr != 'files'}

    def getxattr(self, path, name, position=0):
        p = self.getmeta(path)
        attrs = p.get('attrs', {})
        try:
            return attrs[name]
        except KeyError:
            return ''  # Should return ENOATTR

    def listxattr(self, path):
        p = self.getmeta(path)
        attrs = p.get('attrs', {})
        return attrs.keys()

    def mkdir(self, path, mode):
        ppath, cname = self.splitpath(path)
        p = self.getmeta(ppath)
        p['files'].append(cname)
        p['st_nlink'] += 1
        self.putmeta(ppath, p)
        self.putmeta(path, dict(st_mode=(S_IFDIR | mode), st_nlink=2,
                                st_size=0, st_ctime=time(), st_mtime=time(),
                                st_atime=time(), files=[]))

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

    def read(self, path, size, offset, fh):
        p = self.getmeta(path)
        if offset + size > p['st_size']:
            size = p['st_size'] - offset
        dd = ''.join(self.getdata(path, range(offset // bsize, (offset + size -
```

```
1) // bsize + 1)))
        dd = dd[offset % bsize:offset % bsize + size]
        return dd

    def readdir(self, path, fh):
        p = self.getmeta(path)
        return ['.', '..'] + p['files']

    def readlink(self, path):
        p = self.getmeta(path)
        return ''.join(self.getdata(path, range(p['st_size'] // bsize)))

    def removexattr(self, path, name):
        p = self.getmeta(path)
        attrs = p.get('attrs', {})
        try:
            del attrs[name]
        except KeyError:
            pass  # Should return ENOATTR
        self.putmeta(path, p)

    def rename(self, old, new):
        #         po, po1 = self.traverseparent(old)
        #         pn, pn1 = self.traverseparent(new)
        #         if po['files'][po1]['st_mode'] & 0o770000 == S_IFDIR:
        #             po['st_nlink'] -= 1
        #             pn['st_nlink'] += 1
        #         pn['files'][pn1] = po['files'].pop(po1)
        #         do, do1 = self.traverseparent(old, True)
        #         dn, dn1 = self.traverseparent(new, True)
        #         dn[dn1] = do.pop(do1)
        ppathold, cnameold = self.splitpath(old)
        ppathnew, cnamenew = self.splitpath(new)

        ppold = self.getmeta(ppathold)
        #  pold = self.getmeta(old)
        ppold['files'].remove(cnameold)
        self.putmeta(ppathold, ppold)

        pold = self.getmeta(old)
        size = pold['st_size']
        self.purgemeta(old)
```

```
        ppnew = self.getmeta(ppathnew)
        ppnew['files'].append(cnamenew)
        self.putmeta(ppathnew, ppnew)
        self.putmeta(new, pold)

        olddata = self.getdata(old, range(size // bsize + 1))
        self.purgedata(old, range(size // bsize + 1))
        self.putdata(new, range(size // bsize + 1), olddata)

    def rmdir(self, path):
        p = self.getmeta(path)
        if len(p['files']) > 0:
            raise FuseOSError(ENOTEMPTY)
        self.purgemeta(path)
        ppath, cname = self.splitpath(path)
        p = self.getmeta(ppath)
        p['files'].remove(cname)
        p['st_nlink'] -= 1
        self.putmeta(ppath, p)

    def setxattr(self, path, name, value, options, position=0):
        # Ignore options
        p = self.getmeta(path)
        attrs = p.setdefault('attrs', {})
        attrs[name] = value

    def statfs(self, path):
        return dict(f_bsize=512, f_blocks=4096, f_bavail=2048)

    def symlink(self, target, source):
        ppath, cname = self.splitpath(target)
        p = self.getmeta(ppath)
        p['files'].append(cname)
        self.putmeta(ppath, p)
        self.putmeta(target, dict(st_mode=(S_IFLNK | 0o777), st_nlink=1,
                                  st_size=len(source)))
        datablks = [source[i:i + bsize] for i in range(0, len(source), bsize)]
        self.putdata(target, range(len(datablks)), datablks)

    def truncate(self, path, length, fh=None):
        p = self.getmeta(path)
        currblks = range((p['st_size'] - 1) // bsize + 1)
        newblks = range((length - 1) // bsize + 1)
```

```
        # create new blocks as needed
        blks_to_create = list(set(newblks[:-1]) - set(currblks))
        self.putdata(path, blks_to_create, ['\x00' * bsize] *
len(blks_to_create))
        # purge existing blocks as required
        blks_to_purge = list(set(currblks) - set(newblks))
        self.purgedata(path, blks_to_purge)
        # last block trunc
        if len(newblks) > 0:
            if newblks[-1] in currblks:
                self.putdata(path, [newblks[-1]], [self.getdata(path, [newblks[-
1]])[0][:length % offset]])
            else:
                self.putdata(path, [newblks[-1]], ['\x00' * (length % bsize)])
        p = self.getmeta(path)
        p['st_size'] = length
        self.putmeta(path, p)

    def unlink(self, path):
        ppath, cname = self.splitpath(path)
        p = self.getmeta(ppath)
        p['files'].remove(cname)
        self.putmeta(ppath, p)
        p = self.getmeta(path)
        self.purgemeta(path)
        blks = range((p['st_size'] - 1) // bsize + 1)
        self.purgedata(path, blks)

    def utimens(self, path, times=None):
        now = time()
        atime, mtime = times if times else (now, now)
        p = self.getmeta(path)
        p['st_atime'] = atime
        p['st_mtime'] = mtime
        self.putmeta(path, p)

    def write(self, path, data, offset, fh):
        p = self.getmeta(path)
        currblks = range((p['st_size'] - 1) // bsize + 1)
        if offset > p['st_size']:
            lfill = [(self.getdata(path, [i])[0] if i in currblks else
''').ljust(bsize, '\x00') for i in
                     range(offset // bsize)] \
```

```
                    + [(self.getdata(path, [offset // bsize])[0][
                        :offset % bsize] if offset // bsize in currblks else
'').ljust(offset % bsize, '\x00')]
            self.putdata(path, range(0, offset // bsize), lfill)
        size = len(data)
        sdata = [data[:bsize - (offset % bsize)]] + [data[i:i + bsize] for i in
                                                     range(bsize - (offset %
bsize), size, bsize)]
        blks = range(offset // bsize, (offset + size - 1) // bsize + 1)
        mod = blks[:]
        mod[0] = (self.getdata(path, [blks[0]])[0][:offset % bsize] if blks[0] in
currblks else '').ljust(
            offset % bsize, '\x00') + sdata[0]
        if len(mod[0]) != bsize and blks[0] in currblks:
            mod[0] = mod[0] + self.getdata(path, [blks[0]])[0][len(mod[0]):]
        mod[1:-1] = sdata[1:-1]
        if len(blks) > 1:
            mod[-1] = sdata[-1] + (self.getdata(path, [blks[-1]])[0][len(sdata[-
1]):] if blks[-1] in currblks else '')
        self.putdata(path, blks, mod)
        p['st_size'] = offset + size if offset + size > p['st_size'] else
p['st_size']
        self.putmeta(path, p)
        return size


if __name__ == '__main__':
    if len(argv) < 4:
        print('usage: %s <mountpoint> <metaport> <dataport1> <dataport2> ..' %
argv[0])
        exit(1)
    logging.basicConfig(level=logging.DEBUG)
    fuse = FUSE(Memory(argv[2], argv[3:]), argv[1], foreground=True, debug=True)
```