Final Project Writeup – Jacob Crain (Graduate), Daniel Hamilton (Undergraduate)

A. Detailed Design Description

Server Side:

- Checksum Implementation

To implement the checksum, we decided to keep it on the server side to reduce the number of requests over the network. The checksum data is appended to the end of the Data block which is size BLOCK_SIZE (defined in config) + 16. This allows a 16B checksum to be stored along side the data while nothing else needs to be changed. When the server gets a request to store/update data it calculates the checksum, appends to the end of the data, and stores it into the specified block. This is shown in the code snippet below:

```
#WRITES TO THE DATA BLOCK
def update_data_block(self, block_number, block_data):
        print("block_number" + str(block_number))
        b = sblock.ADDR_DATA_BLOCKS[block_number - sblock.DATA_BLOCKS_OFFSET].block
        for i in range(0, len(block_data)): b[i] = block_data[i]
        #inplement checksum for the block_data, write to b[i+sblock.BLOCK_SIZE]
        #calculate the checksum
        Checksum_ascii = hashlib.md5((str)(b[0:sblock.BLOCK_SIZE-1])).digest()
        #Now append the checksum data to the additional 16bytes after BLOCK_SIZE bytes in the block
        for i in range(len(Checksum_ascii)):
            b[i+sblock.BLOCK_SIZE] = Checksum_ascii[i]
```

To Check the data for errors on a read the server grabs the data, calculates the checksum again and compares it to the checksum stored. If these checksums match, then the server returns a “decay” variable as bool False. If the checksums do not match, then the server returns the data with variable “decay” as bool True. This allows the client to see whether the data is valid or decayed. Below is the code snippet from Memory.py of this implementation:

```
elif block_number >= sblock.DATA_BLOCKS_OFFSET and block_number < sblock.TOTAL_NO_OF_BLOCKS:
        #load the below into a variable. cutt off the last 16 bytes and perform checksum. Check to see if
        #checksum matches. Then return the data if it does. If it does not match then perform parity to recreate it and send that data to the
        #client and write it back to the block.
        checksumOriginal = sblock.ADDR_DATA_BLOCKS[block_number - sblock.DATA_BLOCKS_OFFSET].block[sblock.BLOCK_SIZE:sblock.BLOCK_SIZE+16]
        #print("MADE IT1")
        datastr = (str)(sblock.ADDR_DATA_BLOCKS[block_number - sblock.DATA_BLOCKS_OFFSET].block[0:sblock.BLOCK_SIZE-1])
        data = sblock.ADDR_DATA_BLOCKS[block_number - sblock.DATA_BLOCKS_OFFSET].block[0:sblock.BLOCK_SIZE-1]
        #print("MADE IT2")
        ChecksumNew = hashlib.md5(datastr).digest()
        #print("MADE IT3")
        decay = False

        #Check if checksum is all null, if they are then the block has no data in it
        #and was just claimed. So return False for decay
        nullCheckSum = True
        for j in range(len(checksumOriginal)):
                if(checksumOriginal[j] != '\x00'): nullCheckSum = False
        if(nullCheckSum == True):
                return (data,decay)
        print("Comparing Checksums in Block: " + str(block_number))
        #compare new checksum and saved checksum.
        #If match then return False, if not match then return True
        count = 0
        for i in range(len(checksumOriginal)):
                #print(checksumOriginal[i],ChecksumNew[i])
                if(checksumOriginal[i] != ChecksumNew[i]):
                        #print("Memory: Block data decayed!")
                        decay = True
                        count += 1
        if((decay == True) and (count == 1)): print("Memory: Block data decayed!")
        return (data,decay)
```

Below is the coed snippet from Server.py of the passing of decay variable to client:

```
def get_data_block(block_number):
        print('+-----'*10)
        print('get_data_block, State: ' + str(state))
        print('+-----'*10)
        passVal = pickle.loads(block_number)
        (retVal,decay)  = filesystem.get_data_block(passVal)
        retVal  = pickle.dumps((retVal,state,decay))
        return retVal
```

- Corruption of Data Block

A new function had to be created to corrupt a bit in the data on a singular block. The lowest function is in Memory.py called corrupt_data_block(). Its input variable is the block number to be corrupted, and it has no return. This function grabs the data in the block and xor's the last byte in the checksum with 0x01. This means that the last bit in the checksum will always be flipped and will always be corrupted after this function is called. Below is the function in Memory.py:

```
#CORRUPTS THE DATA BLOCK
def corrupt_data_block(self, block_number):
        #print("block_number" + str(block_number))
        b = sblock.ADDR_DATA_BLOCKS[block_number - sblock.DATA_BLOCKS_OFFSET].block
        print("b[sblock.BLOCK_SIZE+15]: " + str(b[sblock.BLOCK_SIZE+15]))
        b[sblock.BLOCK_SIZE+15] = chr(ord(b[sblock.BLOCK_SIZE+15])^ord('\x01'))
        #flips bit in the last part of checksum. Throws checksum failure back to client when accessing this block
        print("b[sblock.BLOCK_SIZE+15]: " + str(b[sblock.BLOCK_SIZE+15]))
        #print("here2")
```

A new function was added in Server.py called corruptDataBlock() that could be accessed from the backchannel.py in order for the user to choose what server and block number to corrupt. All this function does is act as a handle from backChannel.py to Memory.py. Below is the function in Server.py:

```
def corruptDataBlock(dataBlock):
        passVal = pickle.loads(dataBlock)
        print('+-----'*10)
        print('corruptDatablock'+str(passVal))
        print('+-----'*10)
        retVal = filesystem.corrupt_data_block(passVal)
        print("Data Corrupted on server ")
        retVal = 'Data Corrupted in server ' + str(portNumber)
        retVal = pickle.dumps((retVal,state))
        return retVal
```

The backChannel.py also received an update from the original adding the functionality for the user to choose what block on the server to corrupt aside from the original server to corrupt. Both can be called when testing the server, but not at the same time as specified in the requirements of this project. On the following page is the implementation of this functionality.

```
for i in range(num_servers) :
    print('running server #' + str(portNum+i))
    proxy.append(xmlrpclib.ServerProxy("http://localhost:" + str(portNum + i) + "/"))
    os.system('gnome-terminal -e \"python server.py ' + str(portNum + i) + '\"')
    #time.sleep(1)

while True:
    choice = int(raw_input("Enter 1 to corrupt server, enter 2 to corrupt block on server..."))
    serverNum = int(raw_input("Select Server to Corrupt..."))
    try :
            if(choice == 1):
                    retVal =  proxy[serverNum].corruptData()
                    print(serverNum)
                    print(pickle.loads(retVal))
            if(choice == 2):
                    blockNum = int(raw_input("Select Block to Corrupt..."))
                    serialMessage = pickle.dumps(blockNum)
                    retVal =  proxy[serverNum].corruptDataBlock(serialMessage)


    except Exception as err :
        print('Connection error.. closing all servers.')
        for i in range(num_servers):
            print("Killing server on port " + str(portNum + i))
            try:
                proxy[i].kill()
            except:
                pass
        break
```

- Failure of Server

To fail the server a global variable called state is in Server.py and is returned with every request of the server. State is initialized as bool True, but when the original corruptData() function is called the state of that server changes to bool False. This way the client can easily see if the server is corrupt/down and can not use or reconstruct data appropriately. Below is the function in Server.py:

```
def corruptData():
        print('+-----'*10)
        print('corruptData'+str(portNumber))
        print('+-----'*10)
        global state
        state = False
        retVal = 'Data Corrupted in server ' + str(portNumber)
        retVal = pickle.dumps((retVal,state))
        return retVal
```

Client Side:

- Virtual to Physical Block Implementation

The inodes of the server are stored on every server in the system for ease of use. Since there are many different servers in the system, we need to translate from virtual blocks of the entire system to the physical blocks on each server. To do this we implemented a quick formula that converted from virtual block number to data block number that cen be seen below:

```
SUMMARY: __translate_virtual_to_physical_block
Translates a virtual block number to a physical block number and the port
offset for the target server.
'''
def __translate_virtual_to_physical_block(self, virtual_block_num):
    serverNum = (int)(math.floor(virtual_block_num/self.virtual_block_size))
    localBlockNum = virtual_block_num % self.virtual_block_size
    return (serverNum, localBlockNum)
```

Virtual_block_size is calculated at the beginning of the program to be the number of datablocks on a server times number of servers. self.virtual _block_size is the amount of data blocks on each server. This allows the client to easily convert a virtual to physical data block and the server its located on.

- Parity Implementation

When the client initializes the server system it also claims all the parity blocks which are located on each server. It will claim total data blocks on server system / number of servers in the system. So, for a server system that has 4096 blocks across 4 servers, it will claim 1024 parity blocks so no data transfer can ever request them. Below is the snippet of Initialize() in client_stub.py (num_parity_blocks is defigned in config as stated above):

```
# initialize the servers
def Initialize(self):
    try:
        # initialize connection to the servers
        for i in range(N):
            self.proxy[i].Initialize()

    # initialize the offset table (i'm very sorry this is so ugly)
        count = 0
        for r in range(N-1):
            for s in range(N):
                self.offset_table[r][s] = int(math.floor(count/(N-1)))
                count += 1

        # claim the parity blocks and switch the direction of block claiming
        for i in range(self.num_parity_blocks):
            self.parity_blocks[i] = self.get_valid_data_block()

        # switch block claim direction
        self.block_claim_dir = NEXT
```

To create the parity blocks we only updated/created them in client_stub.py. The two functions where this happened was free_data_blocks() and update_data_blocks() where they had the same functionality. First the current parity of that data block would be grabbed, same as the current data in the block. The new data which is either Null or actual updating data is then xor'd with the current data. The output of the first xor is then xor'd with current parity data. To find the correct parity block, we translate from physical block data to virtual parity block with the following function:

```
def __pblock_number_to_vparity_number(self, physical_block_num, server_num):
    # parity block list index for specified data block
    r = (physical_block_num - self.first_data_block_num)%(N-1)
    s = server_num

    offset = self.offset_table[r][s]

    row = int(N * math.floor( (physical_block_num - self.first_data_block_num) / (N-1) ))

    iparity = row + offset

    # virtual parity block number
    return self.parity_blocks[iparity]
```

The offset_table is defigned in the beginning of the program and is the repeating array of what data blocks pertain to what parity data block location. For a server system that is N=4 the array looks like [0, 0, 0, 1; 1, 1, 2, 2; 2, 3, 3, 3]. This makes it easier for the program to decide which parity block in the row is located. It will return the virtual block of what data block to write to. Below is the writing of

the parity block in updata_data_block():

```
def update_data_block(self, virtual_block_number, block_data):
    try:
        # read back the current data block contents (1.FIRST READ)
        (serverNumData, pBlockData) = self.__translate_virtual_to_physical_block(virtual_block_number)
        proxyData = self.proxy[serverNumData]                                  # find server
        currData = self.get_data_block(virtual_block_number)

        # read back the current parity block contents (2. SECOND READ)
        vParityNum = self.__pblock_number_to_vparity_number(pBlockData, serverNumData)      # virtual par
        (serverNumParity, pParityNum) = self.__translate_virtual_to_physical_block(vParityNum) # find the
        proxyParity = self.proxy[serverNumParity]
        currParity = self.get_parity_block(serverNumParity, pParityNum)

        # calculate the new parity block contents
        newData = list(block_data)

        # first XOR on the new data and the current data
        midData = self.__xor(newData, currData)

        # second XOR on middle data and current parity to find new parity block
        newParity = self.__xor(midData, currParity)

        # update the parity block contents (4. FIRST WRITE)
        print("Updating parity block " + str(pParityNum) + " on server " + str(serverNumParity))
        serialBlockNum = pickle.dumps(pParityNum)
        serialBlockData = pickle.dumps(newParity)
        rx = proxyParity.update_data_block(serialBlockNum, serialBlockData)
```

Below is the writing of the parity block in free_data_block():

```
def free_data_block(self, virtual_block_number):
    try:
        # read back the current data block contents (1.FIRST READ)
        (serverNumData, pBlockData) = self.__translate_virtual_to_physical_block(virtual_block_number)
        print("Freeing data block " + str(pBlockData) + " from server " + str(serverNumData))
        proxyData = self.proxy[serverNumData]                                  # find server
        currData = self.get_data_block(virtual_block_number)

        # read back the current parity block contents (2. SECOND READ)
        vParityNum = self.__pblock_number_to_vparity_number(pBlockData, serverNumData)      # virtual p
        (serverNumParity, pParityNum) = self.__translate_virtual_to_physical_block(vParityNum) # find t
        proxyParity = self.proxy[serverNumParity]
        currParity = self.get_parity_block(serverNumParity, pParityNum)

        # XOR to update the parity block with data being deleted
        newParity = self.__xor(currData, currParity)

        # update the parity block
        serialNewParity = pickle.dumps(newParity)
        serialBlockNumParity = pickle.dumps(pParityNum)
        proxyParity.update_data_block(serialBlockNumParity, serialNewParity)

        # free the selected data block (finally :P)
        serialPBlock = pickle.dumps(pBlockData)
        rx = proxyData.free_data_block(serialPBlock)
        deserialized = pickle.loads(rx)
        return deserialized[0]
    except Exception:
        print "ERROR (free_data_block): Server failure.."
        return -1
```

- Raid 1 Implementation

RAID 1 configuration is a lot easier to implement than the RAID 5 configuration. Parity is not needed and all that needs to happen is: on every pair of servers, get and write to the same block. If a server failed, then just grab data from the other clone server. Below is the Write and read from the RAID 1 implementation. We also implemented the checksum into RAID 1 as well. :

Read data Block:

```
# +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
#                            BLOCK FUNCTIONS
# +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

def get_data_block(self, virtual_block_number):
    try:
        (serverNum,physicalBlock) = self.__translate_virtual_to_physical_block(virtual_block_number)
        serialMessage = pickle.dumps(physicalBlock)
        for i in range(2):
            p = self.proxy[serverNum*2 + i]
            rx = p.get_data_block(serialMessage)
            deserialized = pickle.loads(rx)
            if(deserialized[2] == True):
                #data has decayed
                #fail the read
                print "ERROR (get_data_block): Server data decay failure, reconstructing data.."
                #read correct data from other server
                p1 = self.proxy[serverNum*2 + 1-i]
                rx = p1.get_data_block(serialMessage)
                message = pickle.loads(rx)
                #write data back to corrupted block
                serialIn1 = pickle.dumps(physicalBlock)
                serialIn2 = pickle.dumps(message[0])
                p.update_data_block(serialIn1,serialIn2)
                return message[0]
            #if state is true, then return message. If False server failed and read from other server
            if(deserialized[1] == True): break
        return deserialized[0]
    except Exception:
        print "ERROR (get_data_block): Server failure.."
        return -1
```

Update Data Block:

```
def update_data_block(self, virtual_block_number, block_data):
    try:
        (serverNum,physicalBlock) = self.__translate_virtual_to_physical_block(virtual_block_number)
        serialIn1 = pickle.dumps(physicalBlock)
        serialIn2 = pickle.dumps(block_data)
        #Update two data blocks at the same time on the pair of servers
        p1 = self.proxy[serverNum*2]
        p2 = self.proxy[serverNum*2 + 1]
        rx1 = p1.update_data_block(serialIn1, serialIn2)
        rx2 = p2.update_data_block(serialIn1, serialIn2)
        deserialized = pickle.loads(rx1)
        return deserialized[0]
    except Exception:
        print "ERROR (update_data_block): Server failure.."
        return -1
```

- Reconstruction of Data and Handling of Failed Server Implementation

To reconstruct the data in RAID 5 was tedious. First, we had to differentiate between a parity block read and a data block read since the ways to reconstruct the data in these two reads is different. Because of this we created a separate get_parity_block() from the get_data_block() to handle the parity block reads. Starting off with get_data_block() we needed to see if the server or checksum failed on the read. To do this we just read the state and decay of the read. If both passed then the data that was returned would be passed on, but if either failed then the data reconstruction would occur. To do this we read the parity block of this data block and stored that. Then we calculated the other two data blocks that needed to be read that the parity block contained. Once they are calculated we read in the data of these data blocks and xor'd them

together with the parity block data. Doing this gave us the correct reconstructed data from the corrupted block or server. If the checksum failed, we update the block again to write over the corrupted data and then pass the reconstructed data on to the caller of the function. Below is the snippet of this functionality:

```
if ((state == True) and (decay == False)):
    # data is good..
    return data
else:
    # data is bad.. reconstruct the block using all other blocks
    if(decay == True): print("Data block decay failure.. reconstructing data")
    else: print("Server  " + str(serverNum) + " failure detected.. reconstructing data")

    serverNumList = [None for i in range(N-2)]
    pBlockNumList = [None for i in range(N-2)]

    #read parity block and remaining data blocks
    # read back the current parity block contents (2. SECOND READ)
    vParityNum = self.__pblock_number_to_vparity_number(physicalBlock, serverNum)   # virtual parity block number
    (serverNumParity, physical_parity_block) = self.__translate_virtual_to_physical_block(vParityNum) # find the physical block number and se
    proxyParity = self.proxy[serverNumParity]                                                          # find server to read/write parity data f
    xorData = self.get_parity_block(serverNumParity, physical_parity_block)

    parityNum = N-1-serverNumParity #0-3 for repetition

    #find the rows which data falls on and server associated with it
    listCount = 0
    for r in range(N-1):
            for s in range(N):
                    if((self.offset_table[r][s] == parityNum)and(s != serverNum)):
                            pBlockNumList[listCount] = int(r + 3*(math.floor((physical_parity_block-12)/N)) + self.first_data_block_num)
                            serverNumList[listCount] = s
                            listCount += 1

    #xor data from server and block numbers, since we use parity for the first xor we only need to read N-2 data blocks
    for j in range(N-2):
            print("Fetching Block " + str(pBlockNumList[j]) + " from server " + str(serverNumList[j]))
            #read from each block and server
            tempSerialMessage = pickle.dumps(pBlockNumList[j])
            tempP = self.proxy[serverNumList[j]]
            temprx = tempP.get_data_block(tempSerialMessage)
            (tempData, state, tempdecay) = pickle.loads(temprx)
            xorData = self.__xor(tempData,xorData)
    passfail = True
    if(decay == True):
            print("Sending data back to server " + str(serverNum) + " in block " + str(physicalBlock))
            # update the data block with new data (3. SECOND WRITE)
            serialBlockNum = pickle.dumps(physicalBlock)
            serialBlockData = pickle.dumps(xorData)
            rx = p.update_data_block(serialBlockNum, serialBlockData)

    return xorData
```

This is very similar to the functionality of reconstruction in get_parity_block(). The only difference when recreating the data in a parity block is we need to grab all the data from the data blocks associated with the parity block and xor them. That is precisely what we do to recreate the parity data and write back to the block if the checksum failed. This data is then passed on to the caller of the function. On the next page is the snippet of the code of get_parity_block() located in client_stub.py and is only called by other functions in this file.

```
def get_parity_block(self, serverNumParity, physical_parity_block):
    try:
            #read all data blocks
            print("Fetching Parity Block " + str(physical_parity_block) + " from server " + str(serverNumParity))
            serialMessage = pickle.dumps(physical_parity_block)
            p = self.proxy[serverNumParity]
            rx = p.get_data_block(serialMessage)
            (data, state, decay) = pickle.loads(rx)
            if((state == False) or (decay == True)):
                    # data is bad.. reconstruct the block using all other blocks
                    if(decay): print("Parity block data decay failure.. reconstructing parity data")
                    else: print("Server " + str(serverNumParity) + " failure detected.. reconstructing parity data")

                    serverNumList = [None for i in range(N-1)]
                    pBlockNumList = [None for i in range(N-1)]
                    parityNum = N-1-serverNumParity #0-3 for repetition

                    #find the rows which data falls on and server associated with it
                    listCount = 0
                    for r in range(N-1):
                            for s in range(N):
                                    if(self.offset_table[r][s] == parityNum):
                                            pBlockNumList[listCount] = int(r + 3*(math.floor((physical_parity_block-12)/N)) + self.first_data_block_num)
                                            serverNumList[listCount] = s
                                            listCount += 1

                    #xor data from server and block numbers
                    parityData = ['\x00' for i in range(config.BLOCK_SIZE)]
                    for j in range(N-1):
                            print("Fetching Block " + str(pBlockNumList[j]) + " from server " + str(serverNumList[j]))
                            #read from each block and server
                            tempSerialMessage = pickle.dumps(pBlockNumList[j])
                            tempP = self.proxy[serverNumList[j]]
                            temprx = tempP.get_data_block(tempSerialMessage)
                            (tempData, state, tempdecay) = pickle.loads(temprx)
                            parityData = self.__xor(tempData,parityData)
                    passfail = True

                    if(decay):
                            print("Sending parity data back to server " + str(serverNumParity) + " in block " + str(physical_parity_block))
                            # update the data block with new data (3. SECOND WRITE)
                            serialBlockNum = pickle.dumps(physical_parity_block)
                            serialBlockData = pickle.dumps(parityData)
                            rx = p.update_data_block(serialBlockNum, serialBlockData)
                    return parityData

            return data
```

- Distributed Load Implementation

The distributed load was done in the get_valid_data_block() function in client_stub.py. The function keeps a counter and every time it is called it increments from 0 to N-1. This way, every time the function is called it grabs a new data block from a different server than before. Below is the snippet of this functionality. The self.block_claim_dir is inititally set to backwards for initially getting all parity blocks and is then set to next and resets the count to 0 for server 0.

```
def get_valid_data_block(self):
    try:
        if(self.block_claim_dir != self.block_claim_dir_old): self.data_blk_ptr = 0
        # Retrieve the physical block
        p = self.proxy[self.data_blk_ptr]
        rx = p.get_valid_data_block()
        (blockNum,state) = pickle.loads(rx)
        print("Retrieved free data block " + str(blockNum) + " from server " + str(self.data_blk_ptr))
        # map physical block number to virtual block number before returning
        # to the client.
        serverNum = self.data_blk_ptr
        pBlockNum = blockNum
        virtual_block_number = self.__translate_physical_to_virtual_block(serverNum, pBlockNum)

        # point to the next server to write data to..
        # block_claim_dir is changed in the init function so the parity
        # blocks and data blocks are claimed in opposite directions.
        if self.block_claim_dir == NEXT:
                # server pattern: 0, 1, 2, 3,.. 0, 1, 2, 3
                self.data_blk_ptr = self.__next(self.data_blk_ptr)
        else:
                # server pattern: 3, 2, 1, 0,.. 3, 2, 1, 0
                self.data_blk_ptr = self.__prev(self.data_blk_ptr)
        self.block_claim_dir_old = self.block_claim_dir
        return virtual_block_number

    except Exception:
        print "ERROR (get_valid_data_block): Server failure.."
        return -1
```

B. Detailed Tests

To test our program, we utilized many of the tests from the previous homework's. The most notable one is one that creates /a, /a/b, /a/b/file.txt, writes to /a/b/file.txt, and then reads from /a/b/file.txt. This works perfectly as it should. Below is a snippet of this test. You can access this test by typing "test" into the command console user interface.

```
elif cmd == TEST:
    # TEST: This is a dummy function used to quickly
    # set up the file system for testing an input.
    dir1        = '/a'
    dir2        = '/b'
    filename    = '/file.txt'
    msg         = 'Hi this is a test'
    offset      = 0
    print("MKDIRS")
    fs.mkdir(dir1)
    fs.mkdir(dir1 + dir2)
    print("Create")
    fs.create(dir1 + dir2 + filename)
    print("WRITE")
    fs.write(dir1 + dir2 + filename, msg, offset)
    print("READ")
    fs.read(dir1 + dir2 + filename, offset, 17)
```

We then tested the mv and rm functionality through the use of the user interface

```
$ test
MKDIRS
Create
WRITE
Retrieved free data block 278 from server 0
Fetching Block 278 from server 0
Fetching Parity Block 22 from server 3
Updating parity block 22 on server 3
Updating data block 278 on server 0
READ
Fetching Block 278 from server 0
/a/b/file.txt : Hi this is a test
$ mkdir /b
$ mv /a/b/file.txt /b/file.txt
$ read /a/b/file.txt 0 20
Error FileNameLayer: Lookup Failure-notinparent!
$ read /b/file.txt 0 20
Fetching Block 278 from server 0
/b/file.txt : Hi this is a test
$ rm /b/file.txt
Freeing data block 278 from server 0
Fetching Block 278 from server 0
Fetching Parity Block 22 from server 3
$ read /b/file.txt 0 20
Error FileNameLayer: Lookup Failure-notinparent!
$
```

Then we moved on to testing the parity and checksum. To do this we compared the read data to the data we reconstructed on a server failure and they matched ensuring us that our reconstruction methods were correct. For the checksum we just compare the new and stored checksum together so we knew we were creating and storing it correctly. The next page shows what the tests for when a server failed.

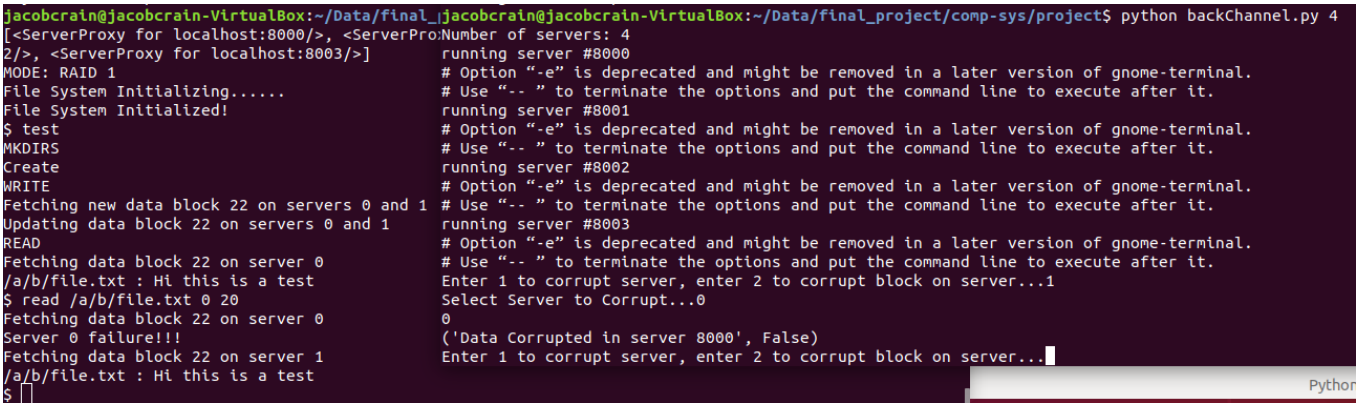
```
$ test                                          KeyboardInterrupt
MKDIRS                                          jacobcrain@jacobcrain-VirtualBox:~/Data/final_project/comp-sys/project$ python backChann
Create                                          Number of servers: 4
WRITE                                           running server #8000
Retrieved free data block 278 from server 0     # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
Fetching Block 278 from server 0                # Use "-- " to terminate the options and put the command line to execute after it.
Fetching Parity Block 22 from server 3          running server #8001
Updating parity block 22 on server 3            # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
Updating data block 278 on server 0             # Use "-- " to terminate the options and put the command line to execute after it.
READ                                            running server #8002
Fetching Block 278 from server 0                # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
/a/b/file.txt : Hi this is a test               # Use "-- " to terminate the options and put the command line to execute after it.
$ read /a/b/file.txt 0 20                       running server #8003
Fetching Block 278 from server 0                # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
Server  0 failure detected.. reconstructing data # Use "-- " to terminate the options and put the command line to execute after it.
Fetching Parity Block 22 from server 3          Enter 1 to corrupt server, enter 2 to corrupt block on server...1
Fetching Block 284 from server 1                Select Server to Corrupt...0
Fetching Block 284 from server 2                0
/a/b/file.txt : Hi this is a test               ('Data Corrupted in server 8000', False)
$                                               Enter 1 to corrupt server, enter 2 to corrupt block on server...
```

In the test above, we started backchannel and then started the FileSystem. Then we ran 'test' in user window. After this we selected option one in the right window to select to corrupt a server and selected server 0 because the data in the first write is in server 0. Remember that every write after this one is the next server, so it goes 0,1,2,3,0,1,2,3. After we corrupted the data on the server, we ran another read on the user interface which can be seen to recognize the server failure and reconstructed the data correctly. The data displayed is the reconstructed data. Below is the testing of the checksum functionality:

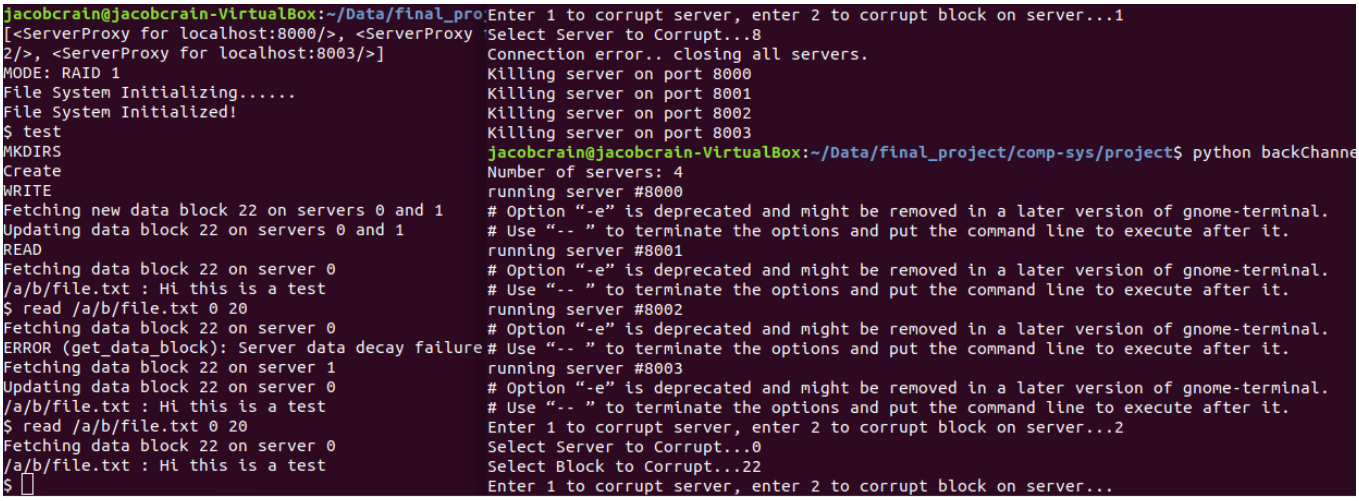
```
$ test                                          jacobcrain@jacobcrain-VirtualBox:~/Data/final_project/comp-sys/project$ python backChannel.py 4
MKDIRS                                          Number of servers: 4
Create                                          running server #8000
WRITE                                           # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
Retrieved free data block 278 from server 0     # Use "-- " to terminate the options and put the command line to execute after it.
Fetching Block 278 from server 0                running server #8001
Fetching Parity Block 22 from server 3          # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
Updating parity block 22 on server 3            # Use "-- " to terminate the options and put the command line to execute after it.
Updating data block 278 on server 0             running server #8002
READ                                            # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
Fetching Block 278 from server 0                # Use "-- " to terminate the options and put the command line to execute after it.
/a/b/file.txt : Hi this is a test               running server #8003
$ read /a/b/file.txt 0 20                       # Option "-e" is deprecated and might be removed in a later version of gnome-terminal.
Fetching Block 278 from server 0                # Use "-- " to terminate the options and put the command line to execute after it.
Data block decay failure.. reconstructing data  Enter 1 to corrupt server, enter 2 to corrupt block on server...2
Fetching Parity Block 22 from server 3          Select Server to Corrupt...0
Fetching Block 284 from server 1                Select Block to Corrupt...278
Fetching Block 284 from server 2                Enter 1 to corrupt server, enter 2 to corrupt block on server...
Sending data back to server 0 in block 278
/a/b/file.txt : Hi this is a test
$ read /a/b/file.txt 0 20
Fetching Block 278 from server 0
/a/b/file.txt : Hi this is a test
$
```

For the checksum we started the backchannel, then the filesystem in Raid 5 configuration. Our 'test' was run and then we entered option 2 for corrupting a block on a server. This will test the checksum on that block. Since the data is on server 0, that server was selected. The first block in the servers for data is 278 as seen under WRITE in the user window where it requests the data block. This is the data block we wished to be corrupted. We than performed a read of the file in the user window which showed that the checksum had failed and there was a decay of data. It reconstructed the data and sent it back to the server to overwrite the decayed data. The data that is displayed in the user window is the recreated data from parity. We than ran another read to ensure that the data was updated, and a new checksum was created for the new data getting rid of the decay error. It succeeded with a successful read.

To test RAID 1 with a server failure and a checksum failure the same methodology was used. We first ran a failure of server 0 where the data is stored. To test the checksum, we corrupted data block ___ on server 0 to see the correction of data. These tests are shown on the following page.



In this test we ran backchannel with 4 servers, then Filesystem in RAID 1. In the client window we ran our 'test', then failed server 0 in the backchannel window. We then performed a read to ensure we grabbed valid data on server 1 which is a clone of server 0. Servers 0,1 are a pair and servers 2,3 are a pair. Next we tested the checksum failure on server 0.



For this test we ran backchannel with 4 servers, then Filesystem in RAID 1. In the client window we ran our 'test', then failed data block 22 on server 0. Data block 22 is the first data block written to as ween after WRITE in 'test'. A read was then executed of the file which signified a decay failure, so the client automatically read from server 1 and updated server 0 with the correct data. A second read was executed to ensure the checksum failure was fixed which it was returning no errors.

C. Step by step instructions on how to use the programs

backChannel.py

- To run in N server configuration: python backChannel.py N
  Ex. To run in 4 server configuration: python backchannel.py 4

- To corrupt a server, first enter option 1
  Then enter the server number to be corrupted
  Ex. Corrupt server 0:

```
Enter 1 to corrupt server, enter 2 to corrupt block on server...1
Select Server to Corrupt...0
0
('Data Corrupted in server 8000', False)
```

- To corrupt a block on a specific server, first enter option 2
  Then enter the server number to be corrupted
  Then enter the block number to be corrupted
  Ex. Corrupt data block ___ on server __

```
Enter 1 to corrupt server, enter 2 to corrupt block on server...2
Select Server to Corrupt...0
Select Block to Corrupt...22
```

FileSystem.py

- To run in RAID 1 or 5 just enter the RAID number after FileSystem.py
  Ex. To run in RAID 5: python FileSystem.py 5
  Ex. To run in RAID 1: python FileSystem.py 1

- To run the 'test' mentioned in this document, just type test once FileSystem is running

```
jacobcrain@jacobcrain-VirtualBox:~/Data/final_project/comp-sys/project$ py
[<ServerProxy for localhost:8000/>, <ServerProxy for localhost:8001/>, <Se
2/>, <ServerProxy for localhost:8003/>]
MODE: RAID 1
File System Initializing......
File System Initialized!
$ test
```

- mkdir – mkdir "directory"
  ex. mkdir /a/c

- create – create "dir+filename"
  ex. create /a/c/file.txt

- write – write "filename" "message" "offset"
  ex. write /a/c/file.txt Hello world I am a program 0

- read – read "filename" "offset" "length"
  ex. read /a/c/file.txt 0 20

- mv – mv “currentFileLocation” “newFileLocation”
  ex. mv /a/b/file.txt /a/file.txt

- rn – rm “FileLocation”
  ex. rm /a/file.txt

- status – status “server”
  ex. status 0

D. Performance Results

To test the difference in the RAID 5 configuration compared to the homework 4 configuration we counted the number of requests each server had and compared them. We also recorded how the server system would behave when a server failure was recorded and the increase in load each server would receive. Below are the quantitative results of each test.

1. Homework 4: 72 requests

```
TEST BENCH COMPLETE - RESULTS BELOW
+-----+-----+-----+-----+-----
REQUESTS
+-----+-----+-----+-----+-----
S0 Req: 72
```

2. RAID 5 without failure: 74 requests -

```
+-----+-----+-----+-----+-----
SERVER REQUESTS
+-----+-----+-----+-----+-----
S0 Req: 17
S1 Req: 9
S2 Req: 15
S3 Req: 33
+-----+-----+-----+-----+-----
SERVER REQUESTS
+-----+-----+-----+-----+-----
S0 Fail: 0
S1 Fail: 0
S2 Fail: 0
S3 Fail: 0
```

3. RAID 5 with Failure: 95 requests with 7 failures on server 0

```
+-----+-----+-----+-----+-----
SERVER REQUESTS
+-----+-----+-----+-----+-----
S0 Req: 17
S1 Req: 16
S2 Req: 22
S3 Req: 40
+-----+-----+-----+-----+-----
SERVER REQUESTS
+-----+-----+-----+-----+-----
S0 Fail: 7
S1 Fail: 0
S2 Fail: 0
S3 Fail: 0
$
```

4. Test Bench used

```
# PERF: This is a dummy function used to test
# the performance of RAID 5 vs HW4's fileystem.
fs.mkdir('/a')
fs.create('/file.txt')
fs.write('/file.txt', 'Hello world', 0)
fs.read('/file.txt', 0, 20)

fs.write('/file.txt', 'This is an overwrite', 0)
fs.read('/file.txt', 0, 20)

fs.write('/file.txt', 'Even load on all servers', 0)
fs.read('/file.txt', 0, 20)

fs.mv('/file.txt', '/a/file.txt')
fs.write('/a/file.txt', 'New data at new location', 0)
fs.read('/a/file.txt', 0, 20)

fs.write('/a/file.txt', 'More and more and more and more', 0)
fs.read('/a/file.txt', 0, 20)

fs.rm('/a/file.txt')
fs.rm('/a')

print("TEST BENCH COMPLETE - RESULTS BELOW")
fs.rf()
```

As we can see above, the RAID 5 configuration had much less average load across the servers compared to homework 4. The RAID 5 with failures has more requests and failures were recorded which is to be expected from these tests. All in all the RAID 5 configuration is the most effective and practical because not only does it reduce load, but it also allows the reconstruction of potentially lost data.