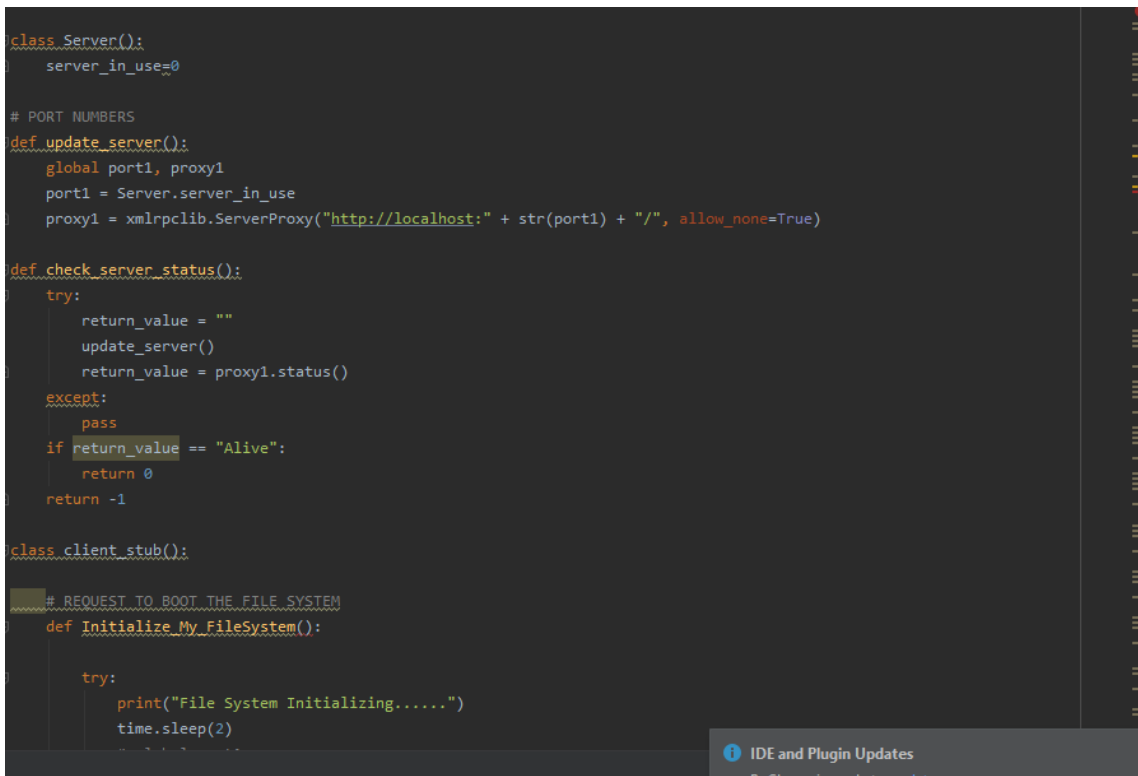


# Project Design Document

## RPC Connection-

- RPC connection is made between memory Interface layer(client side) and Memory layer(server side) of the file system in HW4 and it is scaled to work for at least 4 servers simultaneously.
- To make connection between client and server, we have written functions in client\_stub, memory interface and server stub-
  - **Client stub functions-** update\_server()- Gives the port number of the server which is in use, check\_server\_status()- Provides the status of each server to the client if the servers are alive/active to establish a connection or not.
  - **Memory Interface functions-** get\_active\_server()- Checks the status and returns active servers and active ports list, store\_wait\_time()- provide wait time to read/write data, config\_server()- Takes active port number as input and returns active server linked to it, print\_acknowledgement()- gives message if connection is established.
  - **Server functions-** We have written function check\_status() in server.py to provide the status to the client

## Client stub screenshots-



```
class Server():
    server_in_use=0

# PORT NUMBERS
def update_server():
    global port1, proxy1
    port1 = Server.server_in_use
    proxy1 = xmlrpclib.ServerProxy("http://localhost:" + str(port1) + "/", allow_none=True)

def check_server_status():
    try:
        return_value = ""
        update_server()
        return_value = proxy1.status()
    except:
        pass
    if return_value == "Alive":
        return 0
    return -1

class client_stub():
    # REQUEST TO BOOT THE FILE SYSTEM
    def Initialize_My_FileSystem():
        try:
            print("File System Initializing.....")
            time.sleep(2)
```

IDE and Plugin Updates  
PyCharm is ready to update

```
FileSystem.py × MemoryInterface.py × server1.py × InteractiveWindow_ref.py × client_stub.py ×
52 | exit(0)
53 |
54 | def addr_inode_table(self):
55 |     try:
56 |         return self.proxy.addr_inode_table()
57 |     except Exception as err_:
58 |         print('connection error')
59 |         return -1
60 |
61 | def get_data_block(block_number, server_number):
62 |     try:
63 |         retVal = self.proxy.get_data_block(pickle.dumps(block_number))
64 |     except Exception as err_:
65 |         print('connection error')
66 |         return -1
67 |
68 |     retVal = pickle.loads(retVal)
69 |     return retVal
70 |
71 | def get_valid_data_block(server_number):
72 |     try:
73 |         retVal = self.proxy.get_valid_data_block()
74 |     except Exception as err_:
75 |         print('connection error')
76 |         return -1
77 |     retVal = pickle.loads(retVal)
78 |     return retVal
79 |
80 | def free_data_block(self, block_number, server_number):
```

## MemoryInterface.py screenshots-

```
class Configure_server():
    @staticmethod
    def get_active_servers(port1):

        proxy1 = xmlrpcclib.ServerProxy("http://localhost:" + str(port1) + "/", allow_none=True)
        proxy2 = xmlrpcclib.ServerProxy("http://localhost:" + str(port1 + 1) + "/", allow_none=True)
        proxy3 = xmlrpcclib.ServerProxy("http://localhost:" + str(port1 + 2) + "/", allow_none=True)
        proxy4 = xmlrpcclib.ServerProxy("http://localhost:" + str(port1 + 3) + "/", allow_none=True)

        active_servers = []
        active_ports = []

        return1 = ""
        return2 = ""
        return3 = ""
        return4 = ""

        try:
            return1 = client_stub.check_status()
        except:
            pass

        if return1 == "Alive":
            active_servers.append("server01")
            active_ports.append(port1)

        try:
            return2 = proxy2.check_status()
        except:
            pass
```

## Server.py screenshot-

```
def check_status():
    return "Alive"

global portNumber

filesystem = Memory.Operations()

state = True

def main():
    try:
        portno=0
        if len(sys.argv)==2:
            if sys.argv[1].isdigit():
                portno=int(sys.argv[1])
            else:
                print ("Usage: python filename portno")
                exit(0)
        else:
            print ("Usage: python filename portno")
            exit(0)
    except:
        print("Error occurred")

def configure():
    configuration = [config.TOTAL_NO_OF_BLOCKS, config.BLOCK_SIZE, config.MAX_NUM_TNODES, config.TNODE_SIZE, config.MAX_STL
```

## Client Interactive Window-

- This acts a user interactive layer above file system layer in which we have mentioned in the code about what commands should be given by the user in the terminal to make the connection and run the file system.
- The way the code works is that the number of servers and the port number to be used are given by the user in the terminal and below mentioned functions are called from the below layers to make directory, create file, write, read, remove and move data to print on the screen-
  - the dollar sign "\$ " (with space after it) is used as a prompt
  - Once started, the client takes one command at a time (one per line), until entering the "exit" command, which terminates it.
  - Implemented commands: mkdir, create, mv, read, write, status, rm(all lowercase)

```
if cmd[0]=="exit":
    exit(0)
if cmd[0]=="mkdir":
    if len(cmd)<3 and len(cmd)>1:
        obj.mkdir(cmd[1])
    else:
        print("Usage:mkdir path")
elif cmd[0]=="status":
    if len(cmd)<2:
        obj.status()
    else:
        print("Usage:status")
elif cmd[0]=="create":
    if len(cmd)<3 and len(cmd)>1:
        obj.create(cmd[1])
    else:
        print("Usage:create path")
elif cmd[0]=="rm":
    if len(cmd)<3 and len(cmd)>1:
        obj.rm(cmd[1])
    else:
        print("Usage:remove path")
```

## RAID-5 Implementation-

- RAID-5 is a redundant array of independent disks configuration that uses disk striping with parity. In this project, we have used RAID-5 configuration such that while data is transferred from client to any of the server, a parity block is added in each server to verify if the sent data is same as the received data.
- We have used integers to identify your servers, and configured the system so that there is a total of N=4 servers.

## Virtual Block Number-

The upper layers of the file system are in the client side and the memory and configuration is given in the server side, we have allocated virtual block numbers to be used in the layers of the client side as the server number to be used is not pre defined and hence the physical block number of its memory cannot be used in upper layers until the connection is made with that server.

## Mapping-

- Servers play the role of disks and we have mapped the servers with the physical block numbers by creating a matrix in the code, which has virtual block numbers stored in it.
- In this matrix, the columns represent the servers and rows represent the physical block numbers in each server. Therefore, if we give the virtual block numbers as an input, we can get the server number and the physical block number which we need to refer to.
- We have used functions `check_mapping()` and `addr_translate(Virtual Block number)` in `Memory_Interface.py` for this functionality.
- We have given virtual block number and server number as an argument in the functions of memory interface.

```
# Create matrix with zero value - re-turns i and j
def check_mapping():
    global active_servers
    Mapping = []
    for i in [client_stub.configure.TOTAL_NO_OF_BLOCKS]:
        for j in [active_servers]:
            Mapping[i].append(j)
            Mapping[i][j]=0
        return i,j

# Use above matrix with input virtual block number - returns physical block number and server number
def addr_translate(Virtual_BK):
    global active_servers
    for i in [client_stub.configure.TOTAL_NO_OF_BLOCKS]:
        for j in [active_servers]:
            check_mapping.Mapping[i][j] = Virtual_BK
            i = i +12
        return i,j

#REQUEST TO FETCH THE INODE FROM INODE NUMBER FROM SERVER
def inode_number_to_inode(inode_number):
    global server_number, global active_servers
```


```

#REQUESTS THE VALID BLOCK NUMBER FROM THE SERVER
def get_valid_data_block():
    global Virtual_BK, global active_servers, global server_number
    for i in range(len(active_servers)):
        (block_number, active_servers[i]) = check_mapping()
        active_servers[i] = server_number
        check_block_number = client_stub.get_valid_data_block(server_number)
        Virtual_BK = check_mapping.Mapping[active_servers[i]][block_number]
        Virtual_BK += 12
    return Virtual_BK

#REQUEST THE DATA FROM THE SERVER
def get_data_block(Virtual_BK):
    global server_number, global active_servers
    server_number, block_number = addr_translate(Virtual_BK)
    retVal = ''.join(client_stub.get_data_block(block_number, server_number))
    print('get_data_block')
    print(retVal)
    return retVal

#REQUEST TO MAKE BLOCKS RESUABLE AGAIN FROM SERVER
def free_data_block(Virtual_BK):
    global server_number,
    global active_servers
    server_number, block_number = addr_translate(Virtual_BK)
    retVal = client_stub.free_data_block((block_number, server_number))
    print('free_data_block')
    print(retVal)
addr_translate() > for i in [client_stub.configure... > for j in [active_servers]

```

 IDE and Plugin Updates  
 PyCharm is ready to update

## Parity-

- We have distributed data and parity information across servers, at the granularity of the block size from your configuration file.
- Function parity\_map() is used to keep fixed blocks for the parity data in each server.
- In memory\_inface.py, function XOR() has been created to calculate the data by taking the XOR of the rest of the blocks.

## Functions Read() and Write()-

### Read()-

- In case of no failures, the code allows for load-balancing by distributing requests across the servers, holding data for different blocks – i.e., for a large file consisting of B blocks, there are on average “No. of blocks/No. of servers requests” handled by each server.
- For reading data from the server side, the read function is defined in filesystem.py which first finds the active server and read data from it from the offset.

### Write()-

- For writing data from to the server, the write function is defined in filesystem.py which first finds the active server and read data from it from the offset.
- Write function updates both the data and parity block. If acknowledgments from both replicas return, write is considered to be completed successfully and the client returns. If only one acknowledgment is received, the code registers the server that did not respond as failed, and continue future operations (reads/writes) using only the remaining, non-faulty servers.



## RAID-1-

- RAID-1 configuration should have same data in all the servers, hence mirroring is implemented while writing data to all the servers
- Reading from server is the same as it is done in RAID-5

```
#WRITE TO FILE
def write(self, path, data, offset=0):
    global wait_time
    servers=[]
    server_ports=[]
    init=0
    for each in map_list:
        if each.filename==path:
            #Server port,Server.server in use=each.portno
            servers.append(each.server)
            server_ports.append(each.portno)
            #interface.write(path, offset, data)
            init=1

    if init==1:
        print "write will be performed on"
    for j in servers:
        print j+".py"
    msg="waiting to write in another replica"
    for i in range(len(servers)):
        if i==1 and msg!="":
            print msg
            time.sleep(wait_time)
            Mapping.Server.server_in_use=server_ports[i]
            value=check_server_status()
            if value==-1:
                msg=""
                continue
```

```
#READ
def read(self, path, offset=0, size=-1):
    data_read=""
    servers=[]
    server_ports=[]
    read_flags=[]
    flag=0
    for each in map_list:
        if each.filename==path:
            servers.append(each.server)
            server_ports.append(each.portno)
            read_flags.append(each.read_flag)
            flag=1

    if flag==1:
        print "data resides on"
    for j in servers:
        print j+".py"
    all_1=[1,1]
    if read_flags==all_1:
        read_flags=[0,0]
    for each in map_list:
        if each.filename==path:
            each.read_flag=0
    time.sleep(wait_time)
    ignore=0
    for i in range(len(servers)):
        Mapping.Server.server_in_use=server_ports[i]
        value=check_server_status()
        if value==-1:
```

```
if value==-1:
    ignore=1
    continue
if read_flags[i]==1 and ignore==0:
    continue
print "reading from "+servers[i]+".py"

read_buffer = interface.read(path, offset, size)
if read_buffer != -1:
    print(path + " : " + read_buffer)
for each in map_list:
    if each.filename==path and each.server==servers[i]:
        each.read_flag=1
data_read="done"
break
if data_read=="":
    for i in range(len(servers)):
        Mapping.Server.server_in_use=server_ports[i]
        value=check_server_status()
        if value==-1:
            continue

        print "reading from "+servers[i]+".py"

        read_buffer = interface.read(path, offset, size)
        if read_buffer != -1:
            print(path + " : " + read_buffer)
        for each in map_list:
            if each.filename==path and each.server==servers[i]:
```

FileSystemOperations > read() > for i in range(len(servers))

IDE and Plugin Updates  
PyCharm is ready to [update](#).

## How to Run Code-

- Open Interactive window.py, server\_1.py, server\_2.py, server\_3.py, server\_4.py in Linux terminal
- Run Interactive window.py with no arguments, and run rest of the server files with argument of port number 8000, 8001, 8002 and 8003
- Enter commands in client interactive window terminal like mkdir ("/A"), create ("/A/1.txt"), write("/A/1.txt", strjoin, 0) , read("/A/1.txt", 7, 9) along with \$ sign

## Testing-

Some of the test scenarios are given below:

- Changing the values of sizes in config.py
- Changing the length of input data to be transferred from client to server
- Testing each layer separately by giving input accordingly